



POLITECNICO
MILANO 1863

MACHINE LEARNING

COURSE NOTES

Author: ALESSIO BRAY

ACADEMIC YEAR 2020/2021

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 5 |
| 1.1 | What is Machine Learning? | 5 |
| 1.2 | Traditional Programming vs ML | 5 |
| 1.3 | Why Machine Learning? | 6 |
| 1.4 | What is ML useful for? | 7 |
| 1.5 | Machine Learning Models | 7 |
| 1.5.1 | Supervised Learning | 8 |
| 1.5.2 | Unsupervised Learning | 10 |
| 1.5.3 | Reinforcement Learning | 11 |
| 2 | Overview of Supervised Learning | 12 |
| 3 | Linear models for regression | 19 |
| 3.1 | Linear Basis Function Models | 21 |
| 3.1.1 | Linear regression | 21 |
| 3.1.2 | Linear regression with non-linear basis functions | 22 |
| 3.2 | Discriminative vs generative approaches | 28 |
| 3.2.1 | Loss functions | 29 |
| 3.3 | Least square minimization | 31 |
| 3.3.1 | Ordinary Least Squares (Closed Form) | 33 |
| 3.3.2 | Gradient Optimization (Open form) | 36 |
| 3.3.3 | Maximum Likelihood ML (Closed Form) | 37 |
| 3.3.4 | Gradient optimization (Open form) | 42 |
| 3.3.5 | Underfitting - Overfitting | 42 |
| 3.4 | Regularization | 46 |
| 3.4.1 | Ridge regression | 47 |
| 3.4.2 | Lasso | 51 |
| 3.5 | Bayesian Linear regression | 54 |
| 3.5.1 | Predictive distribution | 72 |
| 4 | Linear models for classification | 73 |
| 4.1 | Linear classification | 73 |
| 4.1.1 | Geometric interpretation | 75 |
| 4.1.2 | Multiple outputs | 78 |
| 4.2 | Least square for classification | 80 |
| 4.2.1 | Least squares problems | 80 |
| 4.2.2 | Fixed Basis functions | 83 |
| 4.3 | The Perceptron algorithm | 85 |
| 4.3.1 | Perceptron algorithm | 87 |
| 4.4 | Probabilistic Discriminative Models: Logistic regression | 89 |
| 4.4.1 | Maximum Likelihood for logistic regression | 91 |

| | | |
|----------|---|------------|
| 4.4.2 | Multiclass logistic regression | 94 |
| 4.4.3 | Connection between Logistic regression and Perceptron Algorithm | 96 |
| 5 | Bias-Variance and Model Selection | 98 |
| 5.1 | “No Free Lunch” Theorems | 98 |
| 5.2 | Bias-Variance trade-off | 100 |
| 5.2.1 | Bias-Variance decomposition | 101 |
| 5.2.2 | Training-test error | 109 |
| 5.3 | Model selection | 118 |
| 5.3.1 | Curse of dimensionality | 118 |
| 5.3.2 | Feature selection | 120 |
| 5.3.3 | Choosing the optimal model | 122 |
| 5.3.4 | Regularization | 132 |
| 5.3.5 | Dimension reduction | 135 |
| 5.4 | Model Ensembles | 143 |
| 5.4.1 | Bagging | 143 |
| 5.4.2 | Boosting | 146 |
| 6 | PAC-Learning and VC-Dimension | 149 |
| 6.1 | PAC-Learning | 150 |
| 6.2 | VC Dimension | 157 |
| 7 | Kernel methods | 164 |
| 7.1 | Kernels | 166 |
| 7.1.1 | Kernel functions | 167 |
| 7.1.2 | Dual representation | 169 |
| 7.2 | Advantage of dual representation | 176 |
| 7.2.1 | Kernel construction | 176 |
| 7.3 | Kernels for symbolic data | 180 |
| 7.4 | Radial Basis Function Networks | 181 |
| 7.5 | Gaussian processes | 184 |
| 7.5.1 | Prediction | 190 |
| 8 | Support Vector Machines | 197 |
| 8.1 | Learning phase | 199 |
| 8.2 | Dual representation | 207 |
| 8.3 | Prediction | 210 |
| 8.4 | Curse of dimensionality | 213 |
| 8.5 | Handling Noisy data | 214 |
| 9 | Markov Decision Processes | 218 |
| 9.1 | Reinforcement Learning | 218 |
| 9.2 | Sequential Decision Making | 219 |

| | | |
|-----------|--|------------|
| 9.3 | MDP | 227 |
| 9.3.1 | MDP model | 227 |
| 9.4 | Goals and rewards | 230 |
| 9.4.1 | Return | 231 |
| 9.4.2 | Policies | 235 |
| 9.4.3 | Value functions | 237 |
| 9.5 | Optimal Value function | 245 |
| 9.6 | Solving Markov Decision Processes: Dynamic programming | 250 |
| 9.6.1 | Dynamic programming: Policy iteration | 252 |
| 9.6.2 | Value iteration | 260 |
| 9.7 | Linear Programming | 263 |
| 10 | RL in finite domains | 266 |
| 10.1 | Model-free prediction | 269 |
| 10.1.1 | Monte-Carlo reinforcement learning (TD(1)) | 269 |
| 10.1.2 | Temporal difference reinforcement learning (TD(0)) | 278 |
| 10.1.3 | TD(λ) | 289 |
| 10.2 | Model-free control | 297 |
| 10.2.1 | On-Policy-Monte-Carlo control | 297 |
| 10.2.2 | On-policy Temporal-Difference control | 303 |
| 10.2.3 | Off-Policy learning | 309 |
| 11 | Multi Armed Bandit | 313 |
| 11.1 | MAB problem | 314 |
| 11.1.1 | Stochastic MAB | 315 |
| 11.1.2 | Adversarial MAB | 319 |
| 11.2 | Generalized MAB problem | 321 |

1 Introduction

1.1 What is Machine Learning?

What does it mean learning?

Machine Learning. A computer program is said to learn from an Experience E with respect to some class of Tasks T and performance measure P , improves with Experience E (Mitchell 1997).

One of the key elements is **experience**, so machine learning is an **inference process that starts from experience**, from observing facts, data. The task is the goal we want to reach, by leveraging the experience, but we also need to evaluate our performance over the task. In general the learning happens if the performance improve with more data (experience), that means that we are **able to extract more knowledge from data getting better performances**.

ML is a sub-field of AI where the **knowledge comes from**:

- **Experience**
- **Induction:** inference process starting from observation trying to generalize from this observation by learning, inferring general rules (opposite of deductive paradigm that starting from rules and hypothesis generates new knowledge; used at the beginning of AI).

An inductive approach is prone to failure since anything that I don't see in data is unknown to me: I know only what I see, so if I don't see enough I don't know enough (from this reason comes the need of combining induction paradigm with deductive one based on models and not on data). Hence ML is not magic. **ML extract information from data and do NOT create information**, so if data do not contain information ML cannot extract anything and it does not succeed. **Furthermore, ML has an efficiency so only part of the information could be extracted from data**.

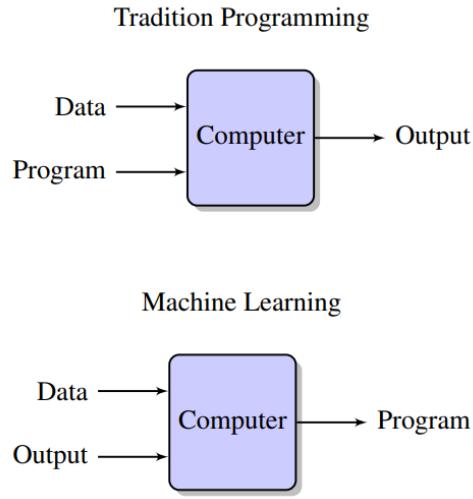
Is important to keep in mind the following aspects:

- Know how it works
- Know how to use it
- **Do NOT create information but only extract part of it (due to efficiency).**

1.2 Traditional Programming vs ML

Traditional programming involves a program that elaborates data to obtain an output. Instead in ML provides to the computer the data and the desired output obtaining the program,

i.e. the lines of code are produced by the computer, not by the developer that need only to provide the output. Therefore, ML could be used only if we know what is the desired output, but is important to remember that **ML goal** is not to remember by memory the output presented but **to learn a general model able to answer even for new data not showed to the computer**, hence a very large set of data is needed.



Example - classification of cat and dogs pictures

- Traditional programming: the program distinguishes the feature of a cat from the one of a dog, they are coded as parameters.
- ML: provide images and the ground truth (output), then the program able to answer to the question is produced.

Human brain is able to distinguish cat and dogs, and is much more difficult to code a program able to distinguish it, so ML approach result easier.

1.3 Why Machine Learning?

We need computers to make informed decisions on new unseen data, but often is too difficult to design a set of rules 'by hand': **machine learning allows to automatically extract relevant information from data** applying it to analyze new data, hence **finding a model able to generalize**. In this way machine learning is automating automation (programs) i.e. getting computer to program themselves, learning how to solve task autonomously: whenever writing the software is the bottleneck with ML you can let the data do the work instead.

1.4 What is ML useful for?

ML is becoming widespread:

- Computer vision and robotics
- Speech recognition
- Biology and medicine
- Finance
- Information retrieval, Web search, ...
- Video gaming
- Space exploration
- Many application and many jobs...

NOTE AI contains all methodologies that tries to mimic human brain. ML is a field of AI that learn in an inductive way. Data science instead can be considered an application of ML techniques and so focuses on the technologies to make machine learning work.

NOTE in ML data quality is very critical, and data mining focus precisely on this task: obtain good data and dataset.

NOTE Deep Learning is a technique of ML.

NOTE Is important to know the problem I am working on? Having a very large amount of data ML models can be used as black boxes, but having a low amount of data may require the knowledge. In particular a trade-off is needed: the use of the knowledge about the problem should be limited since otherwise we are learning something that we already know, hence in data scarcity situations the problem knowledge is needed but must be contained.

1.5 Machine Learning Models

Machine learning is classified in three main sub-fields:

- **Supervised learning** i.e. *learn the model*
- **Unsupervised learning** i.e. *learn the representation*
- **Reinforcement learning** i.e. *learn the control*

this classification is not exhaustive but these are the main classes.

1.5.1 Supervised Learning

The objective of this type of models is to **learn the model**, i.e. to **find the relationship between the data observed as input and the ground truth**, the output I want to predict. Hence the goal of supervised learning is to learn a **function that maps** (relate) the **input space x to some target space t** , i.e. maps the data to the answer.

- *Goal*

- **Estimating the unknown model that maps known inputs to known outputs.**
- Training set:

$$D = \{< x, t >\} \rightarrow t = f(x)$$

where x is the input and t is the output and f is the model that maps input to outputs i.e. given a set of pairs we obtain the model able to map a new input to its corresponding output.

- *Problems*

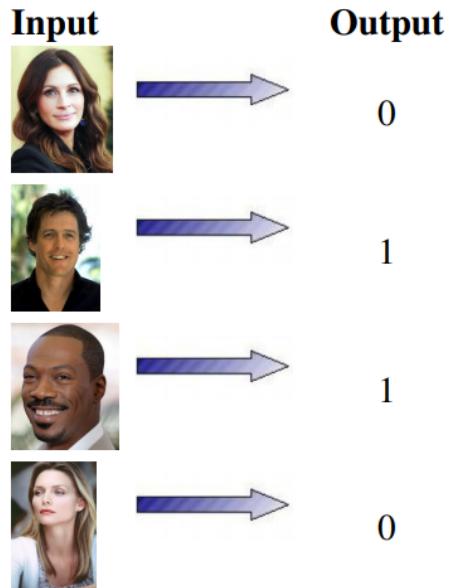
- **Classification** (e.g. predict if the image contains a cat or dog)
- **Regression**, where the target is not a label with few values but it is a continue variable (e.g. prediction of the price of an house t based on the feature of the house x , such as number of bathroom, m^2 , ..., hence find a relationship between the feature of the house and the price).
- **Probability estimation**, similar to regression with the limitation that the output of the model is a probability, so its value is limited between 0 and 1.

- *Techniques*

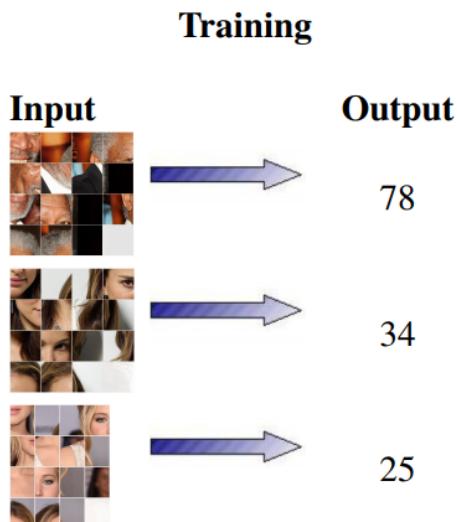
- Artificial Neural Networks
- Support Vector Machines
- Decision trees
- Etc.

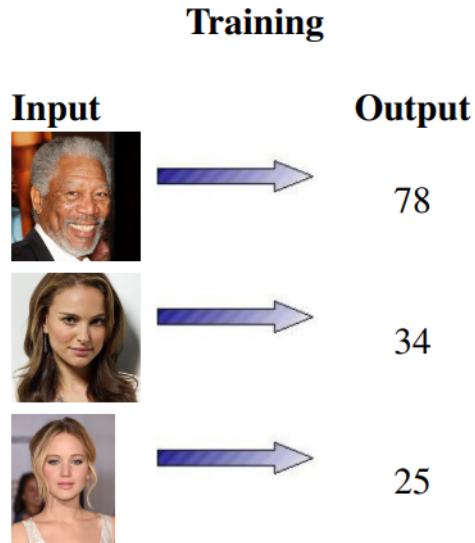
Humans are used and can learn with very few data, automatically filtering out not relevant information, hence we can limit the number of input dimensions, but instead a machine learning algorithm needs a lot of example to extract meaningful information. For example if we have a small set of photo of men and women we are able to say that the goal is to distinguish if the photo contains a man or a woman since we are filtering out useless features like colours or background, instead a machine would see all the features not being able to understand with few samples the goal.

Training



Furthermore, machines see pixel. Differently from humans there is no difference between images divided in pieces and then shifted:





Human brain performs very well with some tasks but badly with others, and this is the same in machine learning: **to learn with few samples assumptions are required, because working with infinite hypothesis a lot of samples are needed.**

1.5.2 Unsupervised Learning

The objective of this type of models is to **learn a better representation of the data** i.e. **there is no supervision, no ground truth is given to the model**, and you want to **find structure of the input data in order to improve the representation** and then elaborate the data so that they can be used for other things such as supervised learning or reinforcement learning. For this reason this type of models **can be seen as data pre-process**. For example if data is divided into clusters we may want to identify them since they may contain some information. Another important usage of unsupervised learning is for **dimensionality reduction** of data described by many variables but some are redundant, hence a more **compact representation of data** containing more or less the same information is achievable (remember that **learning is easier if the input dimension is lower**, hence is good to avoid redundancy in the representation of data).

- *Goal*
 - Learning a more efficient representation of a set of unknown inputs.
 - Training set:
- $$D = \{x\} \rightarrow ? = f(x)$$
- *Problems*

- **Compression i.e. dimensionality reduction:** find a new representation that allow to have the same information or to lose small amount of information using less variables, indeed may happen that some variables are used to describe particular things that are not important in our case.

- **Clustering**

- *Techniques*

- K-means
- Self-organizing maps
- Principal Component Analysis
- Etc.

Example - compression meaning and importance What is the probability of generating an image of a face by generating an image at random? is almost 0 since we consider all the possible images in the world. So if the task is recognizing faces working in the space of all images is useless, so is better to find a new representation where I have more or less only faces that correspond exactly to the reduction of the input space.

Example - reduction of input space needed If the points in the space are represented with 4 coordinates but by looking at them they all belong to the same plane, hence the point can be represented with only 2 coordinates by simply applying a transformation to the 4 (rotation and translation).

1.5.3 Reinforcement Learning

The objective of this type of models is to **learn control, learning to make decisions and take actions** i.e. finding the **best action to perform to increase the utility**, hence learn how to behave using experience. This type of models are the one closer to our idea of learning (e.g. learning how to walk without falling).

- *Goal*

- **Learning the optimal policy.**
- Training set:

$$D = \{< x, u, x', r >\} \rightarrow \pi^*(x) = \text{argmax}_u\{Q^*(x, u)\}$$

where $Q^*(x, u)$ must be estimated. The input meaning is the following: x is the state, i.e. the situation we are in, u is the action chosen to be performed, x' is the new state obtained starting from x and applying the action u and r is the reward, telling if the action being performed on x (i.e. u) is good (> 0) or not (< 0). The goal is to find the optimal policy, i.e. the optimal action to choose in each state x in order to maximize the future rewards, i.e. the cumulative utility.

- *Problems*
 - Markov Decision Process (MDP)
 - Partially Observable MDP (POMDP)
 - Stochastic Games (SG)
- Techniques
 - Q-learning
 - SARSA
 - Fitted Q-iteration
 - Etc.

NOTE Reinforcement learning is correlated but different from planning (Min-Max search) since in the latter the model is known and the starting state is a specific one (the root of the tree), instead in reinforcement learning the model is not known and the problem must be solved starting from any state (node of the tree).

2 Overview of Supervised Learning

Supervised (**inductive**) learning is the largest, most mature, most widely used sub-field of machine learning:

- *Given*: training data set including desired outputs: $D = \{< x, t >\}$ from some unknown function f .
- *Find*: A good **approximation of f** that **generalizes well on test data not seen during training** i.e. predict the answer in new situations.

Input variables x are also called **features** (, predictors, attributes). Output variables t are also called **targets** (, responses, labels). Furthermore:

- If t is **discrete**: *classification*.
- If t is **continuous**: *regression*.
- If t is the **probability of x** : *probability estimation*.

When is useful to apply supervised learning?

- There is *no human expert* (e.g. DNA analysis: nature provide the input and the output hence we can use data to find the relationship between genes and problems such as cancer).

- Humans *can perform the task but cannot explain how*, hence is difficult for them to write a program able to do it (e.g. character recognition), is much easier to give samples and let ML to solve the problem.
- *Desired function changes frequently*, hence this would require to constantly change the program that relates the input to the output, and for this reason we would prefer a ML algorithm that automatically updates the parameters of the function in order to follow the variation in the process (e.g. prediction of stock prices based on recent trading data, hence since the market is non stationary the process would change continuously and the program should adapt to it to make good predictions).
- *Each user need a customized function f* , hence is hard to write a specific program for each user (e.g. email filtering try to predict if an incoming mail is meaningful or not for a particular user, indeed depending on the user the concept of spam could be different).

We want to **approximate a specific function f** , given a data set D (**set of examples**), inside the space F of all possible functions. Any **supervised learning** algorithms requires 3 ingredients to be defined:

- **Loss function (L)**: is a function that **measure the distance between any function and the desired function f** (i.e. the one we are trying to approximate). This give us a measure of how bad or good is the candidate solution, like a score.

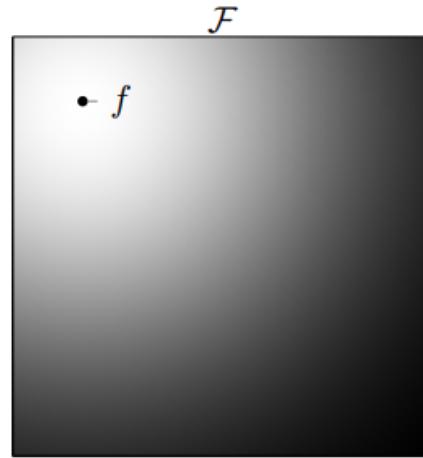
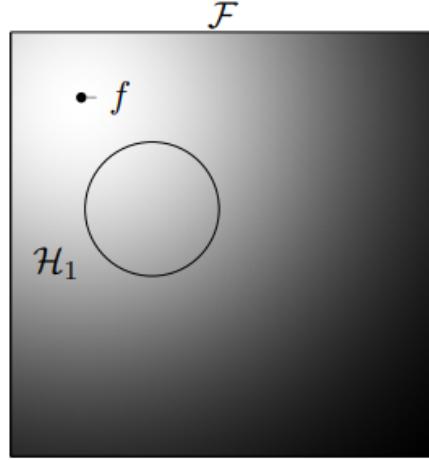


Figure 1: Loss function (color-map)

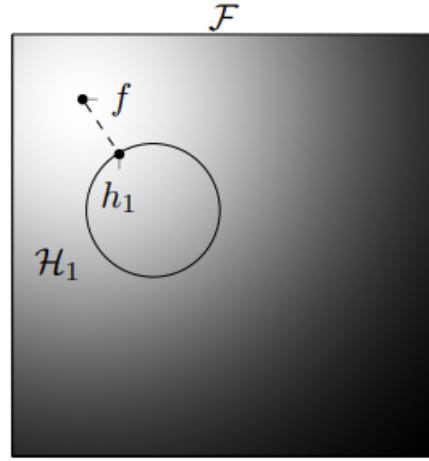
Hence, we should **minimize the loss function**, since greater the values greater the difference between the considered function and the desired one.

- **Hypothesis space (H)**: **subset of function we restrict our search to** (e.g. only linear models,...), i.e. we are approximating the function considering only the ones belonging to the hypothesis space.



Obviously considering all the possible function that maps the input x to the output t the candidate are too many and to learn in such a **large hypothesis space requires a lot of samples**, that makes the solution unfeasible in most of the cases. Hence we restrict our attention to a sub-space.

- **Optimization function (h):** optimization procedure that looks for the best hypothesis inside the hypotheses space, i.e. searching the parameters of the model (e.g. of the neural network, decision tree, ...) that minimize the loss function (over the hypothesis space). This means that it is searching in the hypotheses space which is a space of functions.



In the latter figure h_1 is the best model to approximate f inside the hypothesis space H_1 .

Example - regression: house pricing We want to predict the price of a house given only the square size (m^2):

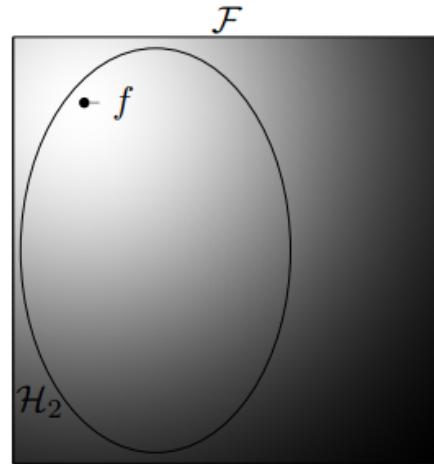
- F : all functions that maps the square size of a house into a price (e.g. linear, discontinuous, gaussian, ...)
- H : hypothesis space (e.g. only linear functions).

In this setup through supervised learning I will look to the function that minimize the loss function, considering only linear ones, i.e. the one in the hypothesis space.

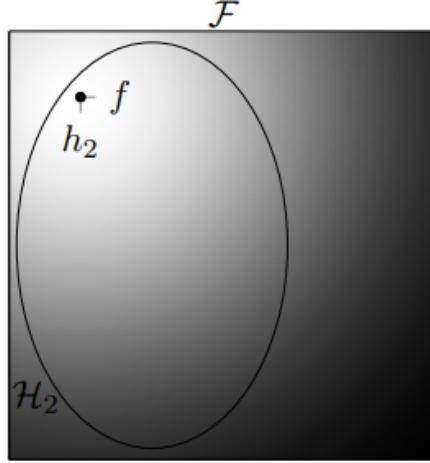
The hypothesis space is the main place where you can put the knowledge about the problem you are facing. Of course **smaller and closer to the desired function is the hypothesis space** is better our resulting model would be. The **objective** is to obtain an hypothesis space that is small enough to be learn with the data you have. Indeed, the **problem is that the size of the hypothesis space is defined by the amount of data used for training**.

NOTE Obviously if the hypothesis space contains the desired function and is very small that means that the knowledge about the problem is such that the problem is already solved, so ML could be useless.

What happens if we enlarge the hypothesis space?



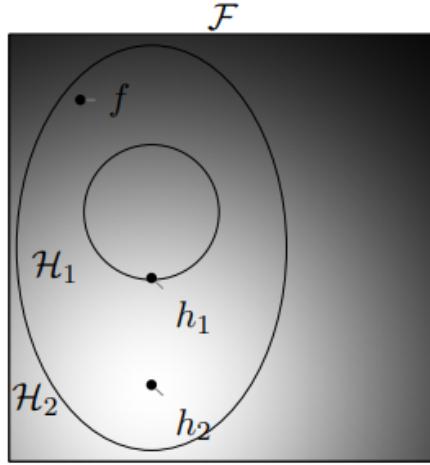
Considering H_2 as hypothesis space then we would find f .



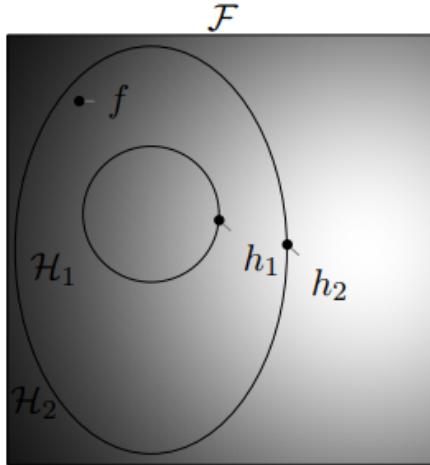
So, why considering a space smaller than the whole space? Indeed the whole space contains f and we would find it through the optimization of the loss function. The reason is that what we are doing here is still not supervised learning. The picture above represent function approximation and the problem is that we **do not know** f , and this is needed in function approximation. **In supervised learning f is unknown, the only thing is known are the samples, hence the loss function is not computed over f , but is computed over the training samples which is like computing the loss function over a discrete number of samples taken from it.** So the **loss function optimized is not the real (continuous) one** (like is shown in the above picture, that would lead to the real function) **but an empirical (discrete) loss function, that could be seen as a noisy measure of the true loss function.** This means that the **more the samples in the training set the less the noise, and better the estimation of the lost function.** Having infinite samples in the training set would mean having the true loss function, hence we could consider the hypothesis space as the whole set of functions. But having few samples the noise of the empirical loss function with respect to the true loss function, may be very large.

In our case we **do NOT know f and we have a finite number of samples.**

So the risk with a finite number of samples is that the minimum h_2 we would find over a larger hypothesis space is no more in f but is far away: **considering a larger hypothesis space the minimum could be very far from f due to the noise in the empirical loss function, instead reducing the hypothesis space dimension H_1 , would lead to a reduced error, since the boundaries of the hypothesis space limit the distance from f .**



Changing the set of samples, that has the same dimension of the previous one, the empirical loss function would be different.



Of course more the example you have better would be the approximation of the loss function and closer the solution to f even with larger hypothesis space, but what these two example tell us is that having few samples is better not to search in a larger hypothesis space since the hypothesis you learn (i.e. the model learnt) is subject to this noise, hence the risk is to obtain a model very far to the true solution you would like to learn.

IMPORTANT Therefore, in supervised learning, since you do not know f , you cannot learn in an arbitrary large hypothesis space, but the dimension of the hypothesis space should be chosen considering the number of samples in the training dataset: having few samples smaller spaces are needed otherwise due to noise the solution would not be robust enough.

The true loss function is never optimized in supervised learning since it is achievable with infinite number of samples, so the loss function we are optimizing could be really different from the loss function true loss function. **The risk is that the optimization of a (bad) empirical function on a larger hypotheses space could lead to learn very bad solutions.**

This **problem of the dimension of the hypotheses space** is correlated to **overfitting** (i.e. taking a hypotheses space too big considering the number of samples we have) and **underfitting** (i.e. taking a hypotheses space too small considering the number of samples we have). Indeed, overfit leads to a bad model that learn ("memorize") the data presented to it, hence the fact to not being able to generalize tell that the optimization of the loss function went wrong and the model is far from the desired function (like we saw in the figures where the empirical function is optimized).

NOTE The colour of the background in the images is the value of the loss: the lighter smaller the loss (i.e. better the solution), the darker higher the loss (i.e. worse the solution). Beware that in figures may change the loss function computed, the true loss function or an empirical one, given a certain amount of data. From this colour scheme, by comparing the figure where the true function is optimized and the one where an empirical function is optimized, is possible to see how bad the solution found using the empirical function is, looking at the darkness of the same point in the other figure.

NOTE The size of the hypotheses space is chosen by the engineer, and the **optimal size of the hypotheses space** is related to the **number of samples** we have in the dataset: **greater the number of samples greater is the dimension of the hypothesis space we can afford.**

NOTE Its all **matter of statistical robustness**: with a great number of samples we can learn complex models; trying to learn complex models with few data we would learn garbage.

NOTE Understanding the proper size of the hypotheses space is the main request in supervised learning: you cannot hope to find it at the first try hence you have to compare different techniques.

NOTE The samples of my dataset are assumed to be taken from the function f we are trying to learn. So the function f is considered to exist by definition. In the course we will **focus on stationary functions**, i.e. the functions considered do not change through time.

NOTE We are **almost never able to find the desired function f** , and always find some kind of approximation, and **the goodness of the approximation depends on the amount of data and the dimension of the hypothesis space**. Furthermore, for some

combination of hypotheses space and loss function we are guaranteed to find the best model inside our hypothesis space, but outside this combination there is no guarantee.

Example Let's consider again the problem of learning the function that maps the square size of a house to its price. Restricting the hypotheses space to the linear models we could learn the function even with a low number of samples. But noticing that the correlation between the two quantities is not linear we would like to try different models like cubic ones, non-linear models like neural networks (and so on), which means trying different hypotheses spaces. Otherwise we could notice that the number of features used (square size) are not enough so we may want to add more features (e.g. number of bathrooms, number of floors, ...), this means that I had to add new parameters increasing the dimension of the hypotheses space. Both of the latter solutions can be applied having enough data.

NOTE Optimization means that we are looking for the function that minimize the loss function, searching the function inside the hypotheses space. For each function in the hypothesis space the loss is computed and then the minimum one is selected.

NOTE From the example the hypothesis h_2 will give worse result with respect to h_1 testing on new data, i.e. worse generalization.

NOTE Supervised learning problem is a little bit **ill-posed** since we want to optimize a function but we do not really have that function but only an approximation of it, whose precision increase with the number of data.

3 Linear models for regression

Regression problems are supervised learning problem where the target of our function is continuous.

The goal of regression is to predict the value of one or more continuous target variables t given the value of a D-dimensional vector x of input variables. The function that maps x to t is unknown but we have some example taken from this function. Data comes from f plus some noise, so there is some noise in the measuring of the values. What we would like to find is the model i.e. the function f that explains the relationship between x and t .

Examples of regression

- Predict stock market price
- Predict age of a web user
- Predict effect of an actuation in robotics

- Predict the value of a house
- Predict the temperature in a building

Many real processes can be approximated with **linear models**. There are many reasons to study linear models:

- **Linear problems can be solved analytically**, i.e. using as hypotheses space a linear combination of the input and some specific loss function (e.g. square error) the regression problem could be **solved exactly, in closed form**, hence you can find the hypothesis in the hypotheses space that minimize the loss function (the optimal value). **The optimization problem can be found with a closed formula**, not requiring very complex algorithm. There are **no local optima**, but **only global ones** that can be computed in closed form. This is a very important point, indeed, using linear models and the solution found do not satisfy you, you are sure that the problem is in the hypotheses space and not in the optimization algorithm (e.g. in neural networks, that are non-linear model, an unsatisfactory result could come from both the hypotheses space, i.e. the hyperparameters of the network, or from the optimization of the network itself).
- Linear prediction provides an introduction to many of the core concepts of machine learning, indeed many models such as neural networks and SVMs can be seen as a generalization of linear models.
- **Linear models augmented with kernels, can model non-linear relationships.**

The simplest form of linear regression models are also linear functions of the input variables. However, we can obtain a much more useful class of functions by **taking linear combinations of a fixed set of non-linear functions of the input variables**, known as **basis functions**. Such models are **linear functions of the parameters**, which gives them simple analytical properties, and yet can be **non-linear with respect to the input variables**. So the "linear" part in the title **refers to the linearity respect to the parameters**. So even though the **model we are building is linear in the parameter space, it can estimate non-linear models in input/output space**.

3.1 Linear Basis Function Models

3.1.1 Linear regression

The simplest linear model for regression is one that involves a linear combination of the input variables. Given a set comprising $D - 1$ input x_j , where $j = 1, \dots, D - 1$ we have,

$$y(x, w) = w_0 + w_1 x_1 + \dots + w_{D-1} x_{D-1} = w_0 + \sum_{j=1}^{D-1} (w_j x_j) = w^T x,$$

where $w^T = [w_0 \ w_1 \ \dots \ w_{D-1}]$, $x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_{D-1} \end{bmatrix}$

(1)

w_0 is the offset of the linear function, furthermore by fixing w^T values we define a certain linear model that maps $x \rightarrow t$: a line if $w \in R^2$ (w_0 is the offset and w_1 is the slope of the model), plane if $w \in R^3$, and hyperplane if $w \in R^k$ with $k \geq 4$. Changing w values you are changing your hypothesis that explain your data i.e. the model. This is often simply known as **linear regression**. The key property of this model is that it is a linear function of the parameters w_0, \dots, w_{D-1} . It is also, however, a linear function of the input variables x_i , and this imposes significant limitations on the model.

In ML there is a distinction between:

- **parametric methods**: we define the size of the hypotheses space independently of the number of samples in the training dataset (e.g. neural networks). You define the **number of parameters** and the parameters **define the size of the hypotheses space**.
- **non-parametric methods**: the size of the hypotheses space change accordingly to the size of the samples of the training dataset.

Linear models are parametric methods, indeed the number of samples does not appear in the function: once the parameters w are learned, the function is fixed independently of how many samples were used to estimate w , so once the parameters are learnt you can forget about the number of samples and use only the model.

Example We want to estimate the weight of a person based on his age, height and gender. So our inputs variables would be

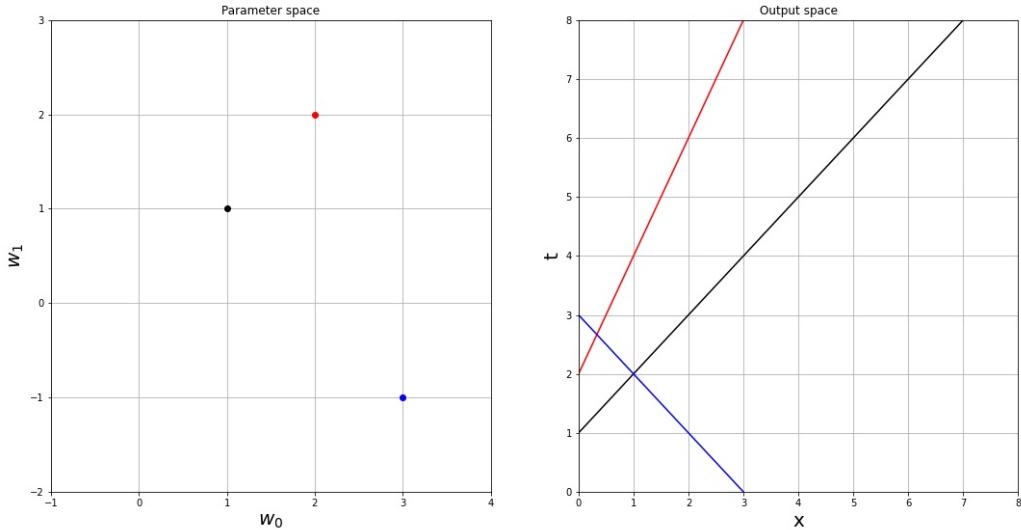
$$\begin{array}{ll} \text{age}, & x_a \in \mathbb{N} \\ \text{height}, & x_h \in \mathbb{N} \end{array}$$

¹Note that we added a dummy sample $x_0 = 1$ to include w_0 in $w^T x$.

$$\begin{array}{ll} gender, & x_g \in \mathbb{N} \\ bias, & 1 \end{array}$$

$$y(x, w) = [w_0 \ w_a \ w_h \ w_g] \begin{bmatrix} 1 \\ x_a \\ x_h \\ x_g \end{bmatrix}$$

NOTE In linear regression we can distinguish two spaces. **Hypotheses space (Parameters space)** and **Output space (Input space)**



3.1.2 Linear regression with non-linear basis functions

Talking about linear model $\mathbf{y}(x, \mathbf{w})$ we are talking about models that are not linear in the input variables x but in the parameters w . Indeed, **input variables can be transformed in new variables with non-linear transformations**. Indeed, in some cases using function linear in the input is bad.

Example Is unlikely to have a linear relationship between the weight and the height of a person, indeed the relationship is about $weight = height^{2.4}$, then the relationship between $weight$ and $height^{2.4}$ is linear.

To consider non-linear functions (of the input) we can use **non-linear basis function**. Hence, **to find a good model we have to find a feature space**, a set of features $\phi_j(x)$ that have a linear relationship with the target value. Then by **linearly combine**

them we have a good model since in the new feature space the relation between transformed input and target is linear so it can be described by a linear model.

We can extend the class of models by considering linear combinations of **fixed non-linear functions** of the input variables, of the form:

$$y(x, w) = w_0 + \sum_{j=1}^{M-1} (w_j \Phi_j(x)) = w^T \Phi(x),$$

where $\Phi(x) = \begin{bmatrix} 1 \\ \Phi_1(x) \\ \vdots \\ \Phi_{M-1}(x) \end{bmatrix}$

(2)

Starting from $D-1$ input variables, usually, we produce $M-1$ features, which means that the **data in $D-1$ dimension** and we **project them in a feature space of $M-1$ dimension**, with $M >$ or $<$ than D (usually $M > D$, but is not always good to increase the features).

By using non-linear basis functions², we allow the function $y(x, w)$ to be a non-linear function of the input vector x . Functions of the form (2) are called linear models because they are linear in w . It is this **linearity in the parameters that will greatly simplify the analysis of this class of models**. From a geometric point of view the task of basis functions is to linearize input space "bending" the point into a straight line.

Some examples of non-linear features usually employed are:

- Polynomial:

$$\phi_j(x) = x^j$$

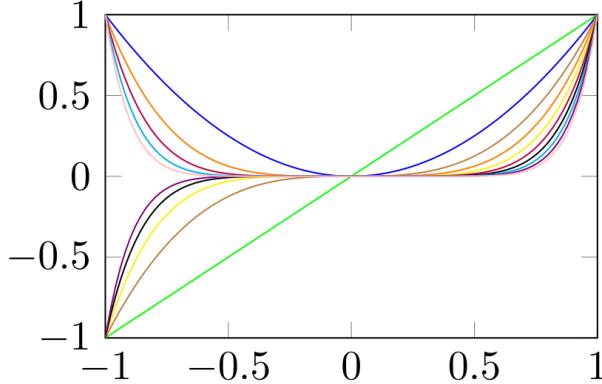


Figure 2: Polynomial features ϕ changing the degree.

²A basis function is an element of a particular basis for a function space. Every function in the function space can be represented as a linear combination of basis functions, just as every vector in a vector space can be represented as a linear combination of basis vectors.

- Gaussian:

$$\phi_j(x) = \exp\left(-\frac{(x - \mu_j)^2}{2\sigma^2}\right)$$

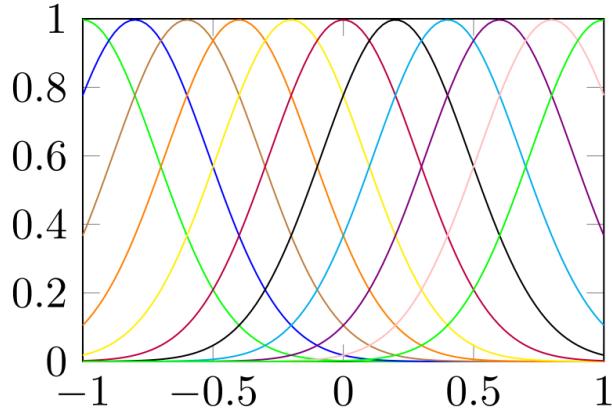


Figure 3: Gaussian features ϕ changing the mean μ_j .

- Sigmoidal:

$$\phi_j(x) = \frac{1}{1 + \exp(\frac{\mu_j - x}{\sigma})}$$

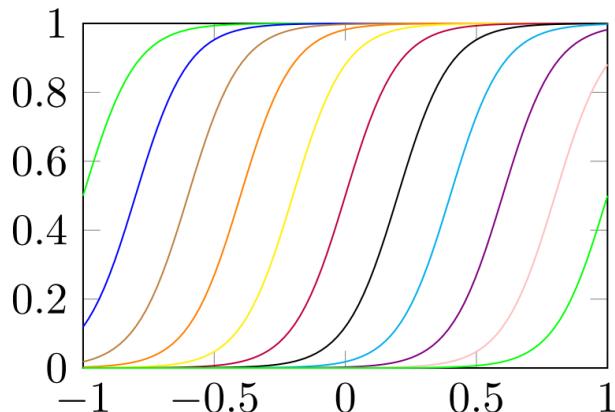


Figure 4: Sigmoidal features ϕ changing the "center" μ_j .

this are some example of functions used to transform input data in new features, that you hope that are linearly related with respect to the target values.

VERY IMPORTANT By the means of this non-linear transformation we are able to represent with the linear combination of the parameter any kind of function provided that you have enough features. Hence, with linear models you can learn arbitrarily complex non-linear functions that relates the input to the target. The problem is to find $\phi_j(x)$, i.e. have good features.

NOTE We want to preserve the linearity of the parameters w , not of the input variables x : using for example $\phi_j(x) = x^2$ we are using a function of grade 2, but the model is still linear since it is a parametric model.

NOTE If x and t have a non-linear relationship I can try to find a new feature space ϕ derived from x that have a linear relationship with t . The good thing is that the **non-linear functions ϕ can be as complex as we want, still having that the optimization problem is the one of a linear model**, so is not affected by the complexity of those functions (even discontinuous). Hence, finding the right ϕ we can have a linear model for any type of function. The utopian case is to know f , which means that the feature is unique and $\phi_1 = f$, $w_1 = 1$

NOTE Changing the features then, in linear models, weights are computed optimally in closed form, so they are automatically adjusted with respect to the new features.

NOTE The features are the place to put your knowledge about the problem, since you can shape the features to solve the problem, which means obtaining features that are linear in t so that we obtain a linear model. Without knowledge instead we should start with a great number of different features, looking for the good ones.

NOTE The model is still linear, even applying non-linear transformation to the input, since this is a parametric model (linearity in x is NOT required), the model $y(x, w)$ is linear in the parameters w , and the features are linearly combined.

NOTE The hypotheses space is the space obtained by changing the parameters in all the possible ways.

NOTE The parameters are only w and ϕ_i are features, non-linear transformation of input variables.

NOTE There is a big difference in considering linear models or **non-linear models** (always referring to the parameters), like neural networks. Indeed there is **no more a closed form solution and the optimization could lead to local optima**. Non-linear model have the **advantage to be more flexible since they can learn the features**, i.e. due to their non-linear structure they can somehow learn ϕ_i , but of course this **comes with**

the cost of local optima and to learn features a lot of data is needed. Indeed the learning problem is much more difficult since in linear model you simply need to learn how to combine the features $\phi_{i,j}$ (weight of each feature) and do not need to learn the features themselves:

- **Linear model:** easier learning, **closed expression** but **less flexible, need of trying several functions** as features.
- Non-linear model: learn the features (more flexible), but more complex learning, more data needed.

Suppose we have found the best linear model for a set of samples (i.e. the best set of parameters that define the red line):

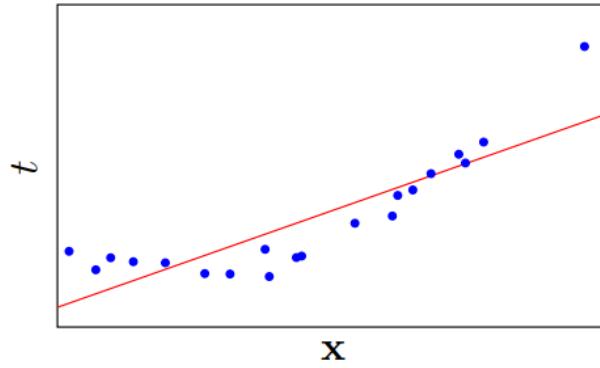


Figure 5: Linear model with features x ($y(w, x) = w_0 + w_1x$)

but actually the best model is something non-linear in x , so cannot be learned by a linear model in x :

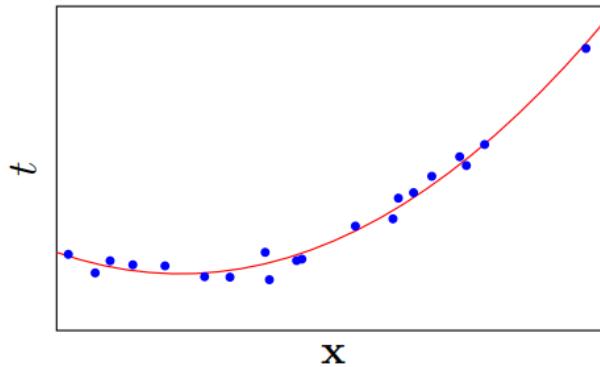


Figure 6: Non-linear model (with respect to x) with features x ($y(w, x) = w_0 + w_1x + w_2x^2$)

Hence, to reach it we could change the features with a non-linear function, for example x^2 (take the input but squared, e.g. squared weight), we are searching the solution in the input space x^2 :

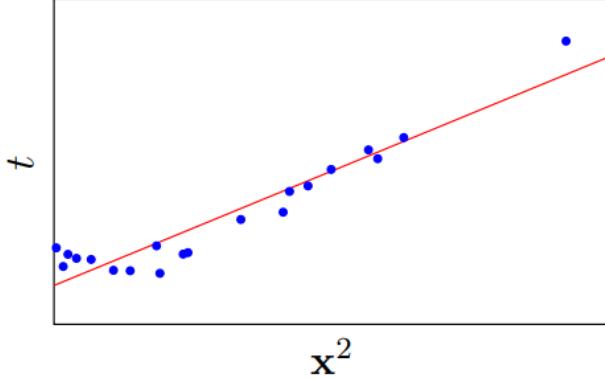


Figure 7: Linear model with feature x^2 ($y(w, x) = w_0 + w_1x^2$)

The model represent in a better way the samples, but it does not completely match, so we decide to use two features with both x and x^2 we obtain the following:

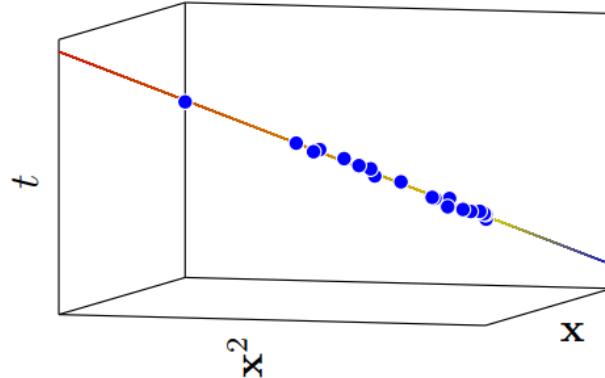


Figure 8: Linear model with features x and x^2 (the plot is a plane but is oriented in such a way to show that samples belong to the plane) ($y(w, x) = w_0 + w_1x + w_2x^2$)

the samples lie on a plane, and this means that t are in a linear relationship with both x and x^2 , so we have found the best model f . This shows that the input variable is only x but I can build more features to be able to capture the relationship between t and x , in this case a second order polynomial, indeed having enough data we could find the parameters that define the plane.

Obviously this is an example, we usually look at the error of the model and if we are not satisfied with it we change features or hypotheses space increasing the number of features, being cautious to not **overfit data i.e. using more parameters than the number**

of samples you have allow you to use, that is equivalent to increase too much the hypotheses space. But this example shows how a certain number of non-linear features of x , could transform a problem not solvable with a linear model in w (a non-linear problem in w) into one that could be solved with a linear problem in w (a linear problem in w).

NOTE The best model in (t, x) space is a parabolic function, instead in the (t, x, x^2) space is a plane.

NOTE The dimension of the hypotheses space is determined by the number of parameters. Hence in the last figure the hypotheses space is enlarged. Therefore, learning in that case require more data.

NOTE The transformation $\phi_{i,j}$ can be any transformation that transform the input vector x into one of the feature of the feature vector ϕ_i . For example x could contain weight and height and $\phi_{i,j}$ could combine weight and height in some way). Is important to remember that **the input x is a vector**.

3.2 Discriminative vs generative approaches

There are different approaches in supervised learning:

- **Generative:** try to model directly the joint probability density:

$$p(x, t) = p(x|t)p(t)$$

and then infer conditional density $p(t|x)$:

$$p(t|x) = \frac{p(x, t)}{p(x)}$$

Finally, **marginalize** the conditional density to **find the conditional mean**:

$$E[t|x] = \int tp(t|x)dt$$

- **Discriminative:** try to model the conditional density

$$p(t|x)$$

Then, **marginalize** the conditional density to **find the conditional mean**:

$$E[t|x] = \int tp(t|x)dt$$

- **Direct:** is not a statistical approach (so it won't try to model $p(x,t)$ or $p(t|x)$) but it searches directly for the model $y(x)$ that minimizes the loss function, directly from training data.

NOTE Discriminative and generative approaches are statistical approaches.

3.2.1 Loss functions

To evaluate which model, and so parameters, is the best, we need to quantify what it means to do well or poorly on a task, i.e. the goodness of a model for the considered task. To do so we need to define a loss (error) function, that consider the ground truth t and the output of the model $y(x)$. First of all we define the **true loss function**, i.e. the loss function **we would like to optimize but we cannot since we do not really know it**, both on a single sample x and over the whole set:

$$\begin{aligned} L(t, y(x)) && && && \text{(true) Loss function} \\ E[L] = \int \int L(t, y(x)) p(x, t) dx dt && && && \text{Average (true) loss function} \end{aligned}$$

where dx and dt means respectively over the input and output space, and $p(x, t)$ is the probability of observing the pair (x, t) as pair of input-output in the training set. **This definition cannot be used**, indeed the problem is that $p(x, t)$ is **unknown**, hence it is **an ideal loss function**. Indeed, **knowing $p(x, t)$ is equivalent to knowing the function**, because it encodes the joint probability of having an input output pair: if you want to know the target value given an input you take p , and by fixing \hat{x} we obtain the conditional distribution of $p(t|\hat{x})$, hence this could be done for any input and this means knowing the true function f . So the latter formula is the one we want to compute but it is not accessible to us. Note that **the process $p(x, t)$ from which data x is picked is stationary so the distribution won't change over time**. Furthermore we are assuming that training data comes from the distribution $p(x, t)$, and to compute the true loss function we would need the distribution p in all the space and not only in the samples we have.

Our goal is to find the model $y(x)$ that minimize the loss function $L(t, y(x))$. The loss function $L(t, y(x))$ must be chosen. If we take the **Minkowsky** loss:

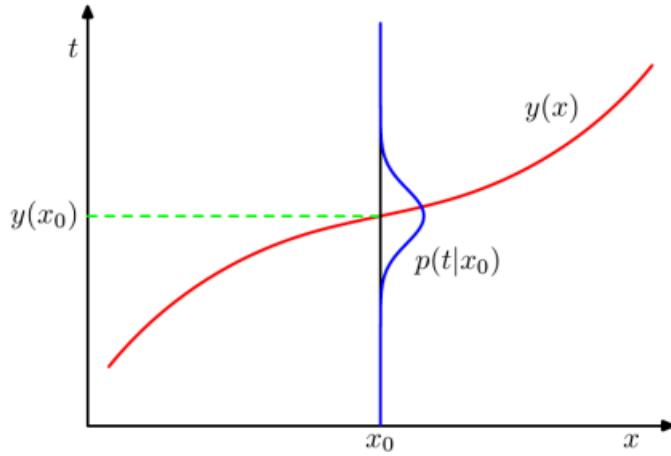
$$E[L] = \int \int (t - y(x))^q p(t, x) dt dx \quad (3)$$

where $y(x)$ is the output produced by the model. Based on q the **model $y(x)$ that minimize $E[L]$** (i.e. the minimum of $E[L]$ is given by), **assuming we know $p(x, t)$ (i.e. utopic situation)**, is:

- q=2:

$$y(x) = E[t|x]$$

where $E[t|x] = \int t \cdot p(t|x) dt$, hence **minimizing $E[L]$ the goal is to learn the conditional average, i.e. the model predict for each point x the expected value**. So the **optimal solution is the conditional average**.



i.e. for a certain value x_0 what we get as prediction is $y(x_0)$, that correspond to $E[t|x_0]$ the expected value of the conditional probability of seeing t given x_0 ($p(t|x)$ i.e. the probability distribution of t given x_0).

- $q=1$:

$$y(x) = \text{median}(t|x)$$

learn the conditional median.

- $q \rightarrow 0$:

$$y(x) = \text{mode}(t|x)$$

learn the conditional mode. As the q is closer to 0 the model obtained from optimization would get closer to the mode and further to the median. Of course $q = 0$ cannot be done.

Which means that finding the minimum of the average loss function $E[L]$ we find those values, hence **changing the loss function we learn different concepts about training data**, i.e. it **extract different information from the data**. Furthermore, changing loss function obviously mean to change the easiness of the learning, i.e. of the optimization (with $q = 1$ and $q \rightarrow 0$ the loss function is harder to optimize). Increasing $q > 2$ the learned concept are not as interpretable as the one we saw, indeed they give more importance to the tails of the distribution, so the risk is to become very sensitive to the outliers (i.e. data far from the mean), and this lead to unstable models. That's why losses for $q > 2$ are not used.

The **choice of the loss function** is guided by two main reasons:

- The **quality of the solution**
- The **easiness of the optimization** of the function.

NOTE For $q=2$ we have the **squared loss function**. This is the most used one, since it has **nice properties**.

NOTE In this sub-section we are talking about $y(x)$ which is not a linear function, but a general one, in particular the one that minimize the true average error (so it could be not linear).

NOTE In a conditional distribution the point with the largest probability is not the mean but the mode, hence for $q = 2$ (squared loss function) we are not learning the $y(x)$.

NOTE What we can do is to replace the probability $p(t, x)$ is replaced with a Montecarlo approximation of the integral: instead of computing the integral with respect to all the possible pair (x, t) we simplify it by simply focusing on the pair available in the training set, which means that instead of computing the integral over the whole pair space we will simply sum the error over the samples present in the training set. This will be our **empirical loss function, a rough approximation** (like approximating the expected value of a distribution with an empirical mean: if the samples are a lot then the empirical mean will converge to the expected value).

3.3 Least square minimization

Since the true loss function is unknown the we **replace the integral with a summation over the samples**.

The method of least squares is a standard approach in regression analysis to **approximate a model by minimizing the sum of the squares of the residuals** (errors). Given N samples, we consider the following loss (error) function:

$$L(w) = \frac{1}{2} \sum_{n=1}^N (y(\underset{n^{th} input}{x_n}, w) - \underset{n^{th} target}{t_n})^2 \quad (4)$$

hence we are computing an **empirical loss function**. This loss function is called **SSE (Squared Sum of Errors)** or half-RSS (Residual Sum of Squares). It can be also written as:

$$RSS(w) = \|\epsilon\|_2^2 \underset{l_2 norm}{\text{square}}, \quad \text{where } \epsilon = \begin{bmatrix} y(x_1, w) - t_1 \\ \vdots \\ y(x_N, w) - t_N \end{bmatrix} \quad (5)$$

with $\|\epsilon\|_2^2 = \sum_{i=1}^N \epsilon_i^2$, that is the sum of the l_2 -norm of the vector of the residual errors.

NOTE Given $x = [x_1 \ x_2 \ \dots \ x_N]$, the l_2 -norm of x corresponds to the euclidean norm i.e.:

$$l_2\text{norm}(x) = \|x\|_2 = \sqrt{\sum_{n=1}^N x_n^2} \quad (6)$$

indeed the SSE is a squared l_2 -norm.

NOTE We **cannot minimize the true loss function** but we **can minimize** the latter loss function (**SSE**) since it is computed **over the samples** of the function we have.

3.3.1 Ordinary Least Squares (Closed Form)

Let's construct the loss function using the matrix notation. Given N samples and M parameters, we construct the **target vector**:

$$t = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix}$$

and what is called the **design matrix**:

$$\Phi = \begin{bmatrix} \Phi(x_1) \\ \vdots \\ \Phi(x_N) \end{bmatrix}$$

The design matrix has a **number of columns equal to the features** (M) and a **number of rows equal to the number of samples of the dataset** (N), indeed $\Phi(x_i)$ (i.e. the i -th row) is a vector with

$$\phi(x_i) = [\phi_1(x_i) \dots \phi_M(x_i)]$$

Instead the vector w has, obviously, size M , one for each feature:

$$w = [w_1 \dots w_M]$$

We can rewrite the SSE in matrix notation

$$L(w) = \frac{1}{2} RSS(w) = \frac{1}{2}(t - \Phi w)^T(t - \Phi w) \quad (7)$$

which is:

$$\frac{1}{2}([Nx1] - [NxM][Mx1])^T([Nx1] - [NxM][Mx1]) = \frac{1}{2}([1xN])([Nx1]) = [1x1]$$

In this formulation Φw represent the prediction of the model for each point in the training set. Hence $(t - \Phi w)$ is the error between the prediction of the model and the target.

NOTE $\Phi[NxM]$, $w[Mx1]$, $t[Nx1]$ where the [] next to the symbol indicates their dimensions.

NOTE $(t - \Phi w)^T(t - \Phi w) \equiv (y(x_n, w) - t_n)^2$

To minimize $L(w)$, i.e. finding w that minimize L , we have to compute the gradient of the function with reference to w (i.e. $[M \times 1]$), which means the first derivative, to find the stationary points, and the Hessian of the function, which means the second derivative, to find which of the stationary point is the minimum (**due to the fact that the model is linear in w the minimum would be global**):

$$\frac{\partial L(w)}{\partial w} = \frac{\partial(\frac{1}{2}(t - \Phi w)^T(t - \Phi w))}{\partial w} = -\Phi^T(t - \Phi w) \quad [M \times 1] \quad (8)$$

$$\frac{\partial^2 L(w)}{\partial w \partial w^T} = \Phi^T \Phi \quad [M \times M] \quad (9)$$

NOTE $\frac{\partial L(w)}{\partial w}$ is a gradient so the partial derivative is computed in the directions of every w_i .

NOTE The $1/2$ in the error computation is put there to obtain a scalar coefficient of 1 in the gradient.

NOTE $\Phi^T \Phi$ is like taking the matrix " Φ^2 " (i.e. squared matrix) which means that **all eigenvalues ≥ 0** . It also means that the matrix is **positive semi-definite matrix**³ for, that from a graphical point of view means that the curvature of the function is upward (positive), hence the stationary point we find from $\frac{\partial L(w)}{\partial w} = 0$ is a minimum of $L(w)$.

Now we have to find the minimizing w imposing the first derivative to zero, that **assuming $\Phi^T \Phi$ non-singular** can be computed as:

$$\begin{aligned} \frac{\partial L(w)}{\partial w} &= 0 \\ -\Phi^T(t - \Phi w) &= 0 \\ -\Phi^T t + \Phi^T \Phi w &= 0 \\ \Phi^T \Phi w &= \Phi^T t \\ \hat{w} &= (\Phi^T \Phi)^{-1} \Phi^T t \quad [M \times 1] \end{aligned} \quad (10)$$

Since the Hessian is equivalent to a squared matrix, its eigenvalues are all positive (not equal to zero since we are assuming it to be not singular: having even one eigenvalue equal to zero would mean $\det = \prod \lambda = 0$) so **the stationary point** we have found **localize the global minimum of the function**, i.e. **the optimal solution**. We should remark that **the optimal solution is in closed form, and it can be applied directly**.

Does this solution always exist (i.e. is this solution always computable)? There is a **problem**

³The matrix A is positive semi-definite if $x^T A x \geq 0$, $\forall x \in \mathbb{R} \setminus \emptyset$, so $x^T \Phi^T \Phi x \geq 0$, $\forall x \in \mathbb{R} \setminus \emptyset$

coming from the **invertibility**: the solution can be computed if the **assumption of $\Phi^T\Phi$ non-singular, i.e. invertible, holds**. This **assumption does not hold** anymore if:

- **Two features are linearly dependent** (e.g. height, 2*height: two columns of Φ matrix will be linearly dependent) this means that there are infinite combination of the two dependent features that provide the same result, i.e. an infinite number of w that give the same result. **NEVER BUILD FEATURES THAT ARE LINEARLY DEPENDENT**, is not only useless but it creates problems. Indeed the linear dependency of two columns would imply that the rank of the matrix $\Phi^T\Phi$ is not maximum, so the invertibility of the matrix does not hold anymore.
- **When there are more features than samples**:

$$M > N$$

to have $\Phi^T\Phi$ invertible Φ should be full row rank, being the number of rows of Φ lower than its number of columns. The matrix row rank is not full, thus being not invertible due to the fact that being not full rank means that has some zero eigenvalues.

So the **Hessian (9)** give us some information about features (basis functions) choice importance.

NOTE The meaning of the requirement of the non-singularity of the matrix is the non existence of a closed form solution, i.e. the existence of infinite solution to the minimization problem that are all optima, instead of only one given in closed form.

NOTE In general we want to avoid the situation of having linear features since we are wasting resources with two equivalent features. Instead in some cases we would want to have more features than samples because we hope that in the big set of features we are able to distinguish the most useful one (looking at the weights found, if close to zero they contribute only in a really small part to the solution), like in very complex problem where we do not have the idea of which are the good features, letting the algorithm to understand that.

NOTE Linear dependency could be easily spotted computing the ratio between the two functions, and obtaining a constant.

From a **computational** point of view the **matrix inversion** is a **very complex operation, cubic in the dimension of the matrix**. In fact, the temporal complexity of OLS (Ordinary Least Square) is:

$$O(NM^2 + M^3)$$

Having **many number of features**, since Φ is [NxM] could lead to **very complex computations** (alternative are Cholesky algorithm $O(NM^2/2 + M^3)$ or QR algorithm $O(NM^2)$ for computing the inverse)

3.3.2 Gradient Optimization (Open form)

Closed form solutions are not practical with big data due to complexity of matrix inversion, having many features required by the big amount and correlation of data. Hence we can to apply gradient optimization, that would find the global optima since the function is convex and has not local optima (you cannot be stuck in a local optima like in neural networks). Gradient optimization is **done trough sequential (online) updates** i.e. **updating the parameter sample by sample**, instead of using all the information in one time as in closed form (in a batch way). One way to do it is **gradient descent**:

$$L(x) = \sum_i L(x_i)$$

$$w^{(k+1)} = w^{(k)} - \alpha^{(k)} \nabla L(x_i)$$

where k is the iteration number and α is the **learning rate**. Furthermore, ∇L for the $i-th$ sample is the $i-th$ line of the matrix $\partial L / \partial w = -\Phi^T(t - \Phi w)$:

$$w^{(k+1)} = w^{(k)} - \alpha^{(k)} (w^{(k)T} \Phi(x_i) - t_i) \Phi(x_i)$$

This is done by **scanning** (iterate over) the **samples**, updating each time w and **keeping decreasing the learning rate**. By scanning the training set many times the algorithm would converge to the optimal value of w that is the same one we could compute with the closed form $\hat{w} = (\Phi^T \Phi)^{-1} \Phi^T t$. The advantage of this is that the **complexity of the computation is not cubic in the number of features but is something between quadratic and cubic in the number of features, according to the choice of the learning rate** (but closer to the quadratic).

For **convergence** of this method the **learning rate has to satisfy the stochastic convergence conditions**:

$$\sum_{k=0}^{\infty} \alpha^{(k)} = +\infty$$

$$\sum_{k=0}^{\infty} \alpha^{(k)2} < +\infty$$

which mean that the series of learning rate needs to be divergent and the series of the square of the learning rate needs to be convergent. Their meaning are the following ones:

- $\sum_{k=0}^{\infty} \alpha^{(k)} = +\infty$: the learning rate **should not decrease too fast**.
- $\sum_{k=0}^{\infty} \alpha^{(k)2} < +\infty$: the learning rate **should not decrease too slow**.

hence they **define a range in which the learning rate schedule should be constrained in**. For instance $1/k$ is a series that satisfies this two conditions.

NOTE Usually the gradient is decreased at each iteration, but it is not necessary.

Geometric interpretation We can give a geometric interpretation of the problem. Given the vector of the predicted output over the dataset samples $\hat{t} = [y(x_1, w) \dots y(x_N, w)]^T$, from the previous calculation we can say that \hat{t} is a linear combination of the column of Φ . So \hat{t} lies in a M-subspace S (the column are M) and since \hat{t} minimize $L(w)$ with respect to t , it represents the projection of t in S

$$\hat{t} = \underbrace{\Phi(\Phi^T\Phi)^{-1}\Phi^T}_{\text{Hat matrix } H} t = Ht, \quad H \text{ projects } t \text{ on } S$$

3.3.3 Maximum Likelihood ML (Closed Form)

The ordinary least squared method and the gradient optimization are both direct methods since they try to compute directly the optimal hypothesis. Now we will apply a discriminative approach, that, obviously, arrives at the same conclusion.

We assume a probabilistic model in which the target variable t is given by a deterministic function $f(x)$ (unknown) with additive Gaussian noise ϵ so that:

$$t = f(x) + \epsilon$$

and we want to approximate $f(x)$ with $y(x, t)$. Furthermore, we assume:

$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

Given N i.i.d. (Independent and Identically Distributed random variables) samples, with input $X = \{x_1, \dots, x_N\}$ and outputs $t = [t_1 \dots t_N]^T$ the likelihood function is:

$$p(t|X, w, \sigma^2) = \prod_{n=1}^N \mathcal{N}(t_n | w^T \Phi(x_n), \sigma^2)$$

that is the probability of observing the target t given the input X , the parameters w and the variance of the noise σ^2 . This conditional distribution $p(t|X, w, \sigma^2)$ is simply the product over the samples in the training set where: t_n is the point where the distribution is evaluated, $w^T \Phi(x_n)$ (i.e. the prediction of our model) is the mean of the distribution and σ^2 is the variance, hence each term of the product is the probability of observing t_n in a gaussian distribution with the mean centered in the prediction of our linear model and variance equal to the one of the noise. This probability is called Likelihood. If our model always predict something that is very close to t_n the likelihood would be very high, so this is the desired behaviour, indeed is equivalent to a correct prediction since $t_n \sim w^T \Phi(x_n)$. Instead if t_n is far from the mean of the gaussian it means that our model gives low probability to the value of t_n that is equivalent of a wrong prediction, since the right one is exactly t_n .

NOTE The likelihood expression obtained, comes from the assumption of $\epsilon \sim \mathcal{N}(0, \sigma^2)$, changing this would change the expression.

NOTE The product comes from the fact that the samples are i.i.d.

At this point is clear that we want to find the **parameters w that maximize the likelihood**, that is **equivalent to say that we want t_n to be the value with the most probability, i.e. the mean of the gaussian distribution**:

$$t_i \sim w^T \Phi(x_i) \quad (11)$$

For the properties of the Gaussian distribution we can write the following

$$\begin{aligned} p(t|X, w, \sigma^2) &= \prod_{n=1}^N \mathcal{N}(t_n|w^T \Phi(x_n), \sigma^2) \\ p(t|X, w, \sigma^2) &= \prod_{n=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2} \frac{(t_n - w^T \Phi(x_n))^2}{\sigma^2}} \\ \Downarrow &\quad \text{Transition to ln to simplify calculus.} \\ \text{Min \& Max remain the same (log is monotonic)} \end{aligned}$$

$$\begin{aligned} \ln(p(t|X, w, \sigma^2)) &= \ln \prod_{n=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2} \frac{(t_n - w^T \Phi(x_n))^2}{\sigma^2}} \\ &= \sum_{n=1}^N \ln \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2} \frac{(t_n - w^T \Phi(x_n))^2}{\sigma^2}} \right) \\ &= \sum_{n=1}^N \ln \left(\frac{1}{\sqrt{2\pi}\sigma} \right) + \ln \left(e^{-\frac{1}{2} \frac{(t_n - w^T \Phi(x_n))^2}{\sigma^2}} \right) \\ &= \sum_{n=1}^N \ln \left(\frac{1}{\sqrt{2\pi}\sigma} \right) + \sum_{n=1}^N \ln \left(e^{-\frac{1}{2} \frac{(t_n - w^T \Phi(x_n))^2}{\sigma^2}} \right) \\ &= \sum_{n=1}^N \ln \left(\frac{1}{\sqrt{2\pi}\sigma} \right) + \sum_{n=1}^N -\frac{1}{2} \frac{(t_n - w^T \Phi(x_n))^2}{\sigma^2} \\ &= \sum_{n=1}^N \ln \left(\frac{1}{\sqrt{2\pi}\sigma} \right) - \frac{1}{2\sigma^2} \sum_{n=1}^N (t_n - w^T \Phi(x_n))^2 \\ &= \sum_{n=1}^N \ln \left((2\pi\sigma^2)^{-\frac{1}{2}} \right) - \frac{1}{2\sigma^2} \sum_{n=1}^N (t_n - w^T \Phi(x_n))^2 \\ &= -\frac{N}{2} \ln(2\pi\sigma^2) - \underbrace{\frac{1}{2\sigma^2} \sum_{n=1}^N (t_n - w^T \Phi(x_n))^2}_{RSS} \end{aligned} \quad (12)$$

where the first term is constant since it is independent from w and the second term contain the square loss we have seen and optimized before. Computing the derivative of this expression with respect to w the constant term would be eliminated and the multiplicative constant of the second term is not meaningful for the stationary points ($= 0$). Since to find the maximum likelihood the gradient of $\ln(p(t|X, w, \sigma^2)) = 0$ must be equal to zero the solution we have found is the same as before:

$$\begin{aligned}
\stackrel{w}{\nabla} \ln(p(t|X, w, \sigma^2)) &= \stackrel{w}{\nabla} \left(-\frac{1}{2\sigma^2} \sum_{n=1}^N (t_n - w^T \Phi(x_n))^2 \right) \\
&= -\frac{1}{2\sigma^2} \sum_{n=1}^N 2(t_n - w^T \Phi(x_n))(-\Phi^T(x_n)) \\
&= -\frac{1}{\sigma^2} \sum_{n=1}^N t_n \Phi^T(x_n) - w^T \sum_{n=1}^N \Phi(x_n) \Phi^T(x_n) \\
0 &= \sum_{n=1}^N t_n \Phi^T(x_n) - w^T \sum_{n=1}^N \Phi(x_n) \Phi^T(x_n) \\
w^T &= \sum_{n=1}^N t_n \Phi^T(x_n) \left(\sum_{n=1}^N \Phi(x_n) \Phi^T(x_n) \right)^{-1} \\
w &= \left(\sum_{n=1}^N t_n \Phi^T(x_n) \left(\sum_{n=1}^N \Phi(x_n) \Phi^T(x_n) \right)^{-1} \right)^T \\
&= \left(\left(\sum_{n=1}^N \Phi(x_n) \Phi^T(x_n) \right)^{-1} \right)^T \left(\sum_{n=1}^N t_n \Phi^T(x_n) \right)^T \\
&= \left(\sum_{n=1}^N \Phi(x_n) \Phi^T(x_n) \right)^{-1} \sum_{n=1}^N \Phi(x_n) t_n \quad \text{having } \Phi = \begin{bmatrix} [\Phi^T(x_1)] \\ \vdots \\ [\Phi^T(x_N)] \end{bmatrix} \\
&= (\Phi^T \Phi)^{-1} \Phi^T t \quad [Mx1]
\end{aligned} \tag{13}$$

thus we have shown that the ordinary least square lead to the same solution that optimizes the likelihood of a gaussian model, so **implicitly the ordinary least square is assuming that the dataset has target values affected to gaussian noise**. We can also notice how the solution is independent from σ^2 , so independently from the variance of the noise the optimal model is the same.

NOTE The latter shows another interpretation of the direct method of OLS, indeed it can be seen as a discriminative approach under the assumption of the gaussian noise over the targets t . I.e. **remember that using the OLS you are making the implicit assumption of gaussian noise over the target samples**.

NOTE Assuming the noise not gaussian we would obtain different solutions.

As we can see both OLS and ML give the same result (13).

W variance estimation Once we have computed the parameters we should be **interested** to know **how much I can trust the value of the parameter**. Assuming that:

- the **observations t_i are uncorrelated and have constant variance σ^2**
- x_i are **fixed (non-random)**

the variance-covariance matrix of least-squares estimates is:

$$Var(\hat{w}_{OLS}) = Var(\hat{w}_{ML}) = (\Phi^T \Phi)^{-1} \sigma^2 \quad (14)$$

Therefore, **looking at the variance we can know which parameter are pretty sure and which are very uncertain**. The only problem of this formula is that σ^2 is **usually unknown**, so we need to **estimate σ^2** using the following formula:

$$\sigma^2 = \frac{1}{N - M} \sum_{n=1}^N (t_n - \hat{w}^T \Phi(x_n))^2$$

As we have seen before $\Phi^T \Phi$ **matrix can give us some insights on features importance**. If a **features is relevant its eigenvalue is high**. An effect of this is the **reduction of parameters variance**. In fact the $\Phi^T \Phi$ matrix is inverted (same as divided) and multiplied to error variance σ , so high eigenvalues reduce variance. Furthermore:

Theorem (Gauss-Markov) **The least squares estimate of w has the smallest variance among all linear unbiased estimates.**

It follows that least squares estimator has the lowest MSE of all linear estimator with no bias, so it **does not exist any other unbiased estimator of w with smaller variance**. Telling us that for unbiased solution it doesn't make sense of considering other kind of formula than LS. However **having an unbiased solution is not always desirable**, indeed there **may exist a biased estimator with smaller MSE**. As we will see in future there are always two "forces":

- the **bias of the solutions**. The bias of an estimator is the difference between this estimator's expected value and the true value of the parameter being estimated.

$$Bias(\theta, \hat{\theta}) = E[\hat{\theta}] - \theta = E[\hat{\theta} - \theta]$$

An estimator or decision rule with zero bias is called unbiased.

- the **variance of the solution**

and the **error is the sum of the bias and the variance**. Using an unbiased estimator means that the bias is 0, but usually this imply a high variance hence the error is high. For this reason in many cases we prefer to introduce bias to reduce the variance. From this comes the problem of supervised learning of balancing bias and variance. **So the LSE gives the minimum variance for an unbiased solution but if the variance should be reduced further then may exist another estimator that even if it is biased the variance is reduced so much that is beneficial.** This is true especially with few samples.

NOTE If a estimation is biased then even with infinite samples does not learn the right solution (the variance does to zero but the bias remains and the error is given by the sum of the bias and the variance). Instead, unbiased means that it could learn the right solution with a finite number of samples.

We can **achieve variance reduction** by gathering more samples for our estimation. Furthermore, **if the parameter is small and highly uncertain (i.e. high variance) probably that feature is not meaningful for our model, so we could remove in order to avoid the overfitting.**

NOTE An example of small parameter and highly uncertain can be seen in MIDA 1 course when the data generation system is overmodeled.

Assuming that the model is linear in the features $\Phi_1(), \dots, \Phi_M()$ and that the noise is additive and Gaussian:

$$\hat{w} \sim \mathcal{N}(w, (\Phi^T \Phi)^{-1} \sigma^2)$$

$$(N - M)\sigma^2 \sim \sigma^2 X_{N-M}^2$$

and such properties can be used to form test hypotheses and confidence intervals.

NOTE Libraries often return the parameters with the **p-value**, that is the **probability of that parameter of being close to 0 i.e. useless**: if the p-value is high probably the parameter is useless (only noise for our solution), vice-versa closer to zero more useful seems to be the parameter.

Multiple outputs To solve a regression problem with multiple outputs we **could use different sets of basis function for each output**, having **independent regression problems**. Hence having k outputs, we could solve them with k **independent regression problems**. Usually, a single set of basis function is used:

$$\hat{W}_{ML} = \underbrace{(\Phi^T \Phi)^{-1} \Phi^T}_{{\Phi}^\dagger \text{Pseudo-inverse}} T \quad [M \times K], \quad \text{where} \quad T \quad [N \times K] \quad (15)$$

obtaining for each output t_k the set of parameters \hat{w}_k (**decoupled solutions, i.e. computed independently**):

$$\hat{w}_k = (\Phi^T \Phi)^{-1} \Phi^T t_k$$

where t_k is an N-dimensional column vector. The **advantages lay in the fact that the matrix $(\Phi^T \Phi)^{-1} \Phi^T$ is equal for each model, so it can be pre-computed one time and used for each model** (that is good since it involves the inversion of a matrix, that we know is computationally expansive).

NOTE With multiple outputs the input are the same for each model, so the features are usually the same, but obviously can be considered also different ones.

3.3.4 Gradient optimization (Open form)

Closed form solutions are not practical with large datasets because they are computationally too complex. To overcome this problem we can use iterative approaches which make sequential (online) updates of the parameters instead of calculating them directly. An example is stochastic⁴gradient descent. If the loss function can be represented as a sum over samples ($L(x) = \sum_n L(x_n)$) this iterative approach is applicable. This is the case for least squares.

$$w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)}}{\text{learning rate}} \nabla L(x_n) \quad (16)$$

The learning rate α is an important hyperparameter⁵of the model. It defines the length of each iteration step. A big step makes the iterative process converge faster, but is less precise because it can "jump" the minimum that we are looking for. Similarly if we take small steps we are more precise but we could get stuck in local minima and the convergence is slow. To be sure of algorithm convergence the learning rate must have the following properties

$$\sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)}} = \infty \quad \wedge \quad \sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)2}} = M, \quad M \in \mathbb{R} \quad (17)$$

3.3.5 Underfitting - Overfitting

Model complexity play a very important role for the success of an algorithm. A simple definition of "model complexity" can be define as the **number of parameters and features of a model**. The complexity **influence model capability to generalize**. An optimal model is one that generalize well the problem and **generalization is the model's ability to give sensible outputs to sets of input that it has never seen before**. We can have either **two cases for bad generalization**.

⁴Stochastic means that we won't use all data at once to update the parameters, but we can update them from smaller subsets of the dataset. This is done to reduce the complexity.

⁵In machine learning, a hyperparameter is a parameter whose value is used to control the learning process. By contrast, the values of other parameters are derived via training.

- **Underfitting** The model is **too simple and it generalize too much**. In practice we don't have enough parameter to estimate the true model.
- **Overfitting** The model is **too complex and it relies too much on the given dataset**. The model will behave very well on our dataset, but will perform poorly on other ones. In practice the model **tries to interpolate the dataset** learning the true model and the noise of our starting dataset because it has too many parameters. Having too many degree of freedom the model will minimize the error over the training samples, which means that it will try to interpolate that data, obtaining the interpolation of data noise.

A **naive way to measure the performance of our model is to look at the empirical loss function value**. A lower value corresponds to a better performance. This is true, but **loss function value lacks of valuable information regarding model complexity**. Indeed, as we had said before, an overfitting model tries to interpolate the dataset, this will produce a loss function value that tends to zero but the model true performance is far from optimal.

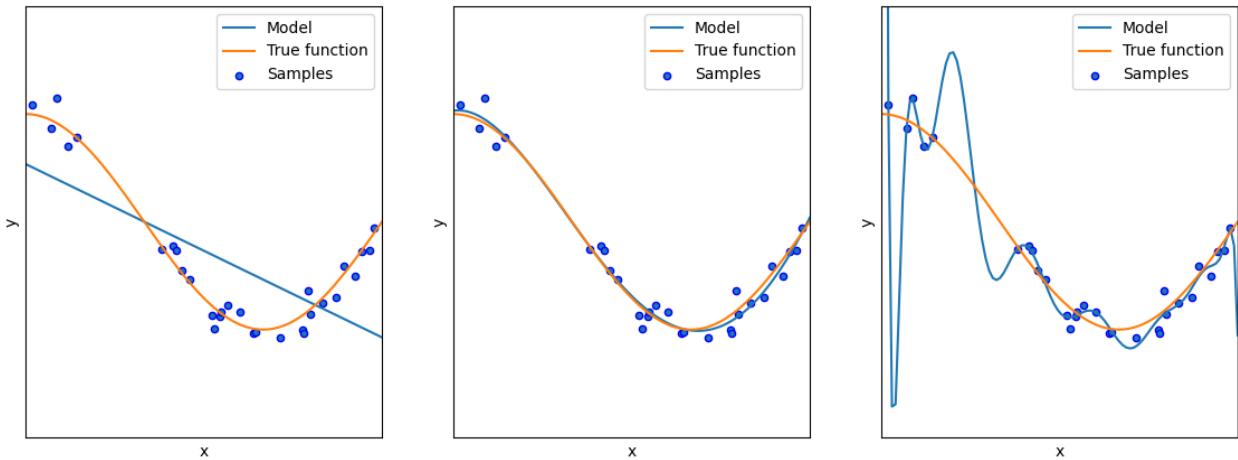


Figure 9: Underfitting, perfect fit, overfit

This happens since we are minimizing the empirical loss function (computed over few points) that is **only a rough approximation of the true loss function (computed on infinite points, i.e. the real function)**, so **increasing the hypotheses space there is the risk that the function we obtain is much distant (poor generalization) with the one that generated the data (f)**. Hence the hypotheses space is to large for the number of samples we have: **more samples would mean a better approximation of the real loss function increasing the possible hypotheses space indeed more samples would need a model of greater complexity to be interpolated, i.e. to reach overfitting**. Furthermore, the empirical loss function is not a good way to measure the performance of the model since it is a bias estimate: the parameters where found using

the same data we are using for performance evaluation, so it badly represent the capability of generalization of the model, as we can notice considering the loss of an overfitting model that is close to zero, but has poor generalization capabilities.

The problems, in general, are the following:

- With **low-order polynomials** we have **underfitting**, i.e. the model is **not flexible enough to represent our data**.
- With **high-order polynomials** we could represent the true function but we are not able to do this since the samples our model could see lead the solution over that has an excellent fit over training data but a poor representation of the true function. The degree of the polynomial to reach this situation depends on the number of samples we have, **greater the number of samples**, greater the polynomial should be to interpolate the samples, which **means that greater the hypotheses space** (which is connected to the number of parameters, connected to the number of features, connected to the polynomial degree) should be to reach overfitting, i.e. greater polynomials degree is needed to obtain overfitting, **indeed the empirical loss function is a better approximation the real loss function**.

Our goal is good generalization, but reducing the empirical loss function do not necessarily means we are decreasing the true loss function, i.e. better generalization.

NOTE When the hypotheses space is very small there is a strong correlation between the empirical loss function and the true loss function. As long as the hypotheses space is increased the correlation between the two vanishes, so even reducing the empirical one, the true one may be very different. The point in which the two function becomes uncorrelated, depends on the number of samples. Having a lot of samples the uncorrelation happens with very complex model, and having few samples happens for simpler models.

There are **many ways to avoid overfitting**. One of them is **model selection** (we will see this later) i.e. **finding the proper size of the hypotheses space**. We cannot say in general that a certain hypotheses space is small or large is always in function of the number of samples: **higher the number of samples higher the dimension of the hypotheses space accepted** (e.g. a model with 1M parameters could be not too large if we have 1M of samples).

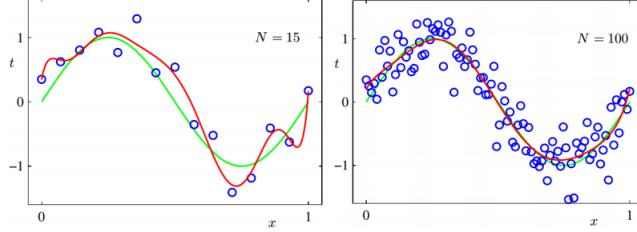


Figure 10: Increasing the number of samples the same hypothesis space gives a good function approximation, whereas with a lower number of samples it suffers a little overfitting.

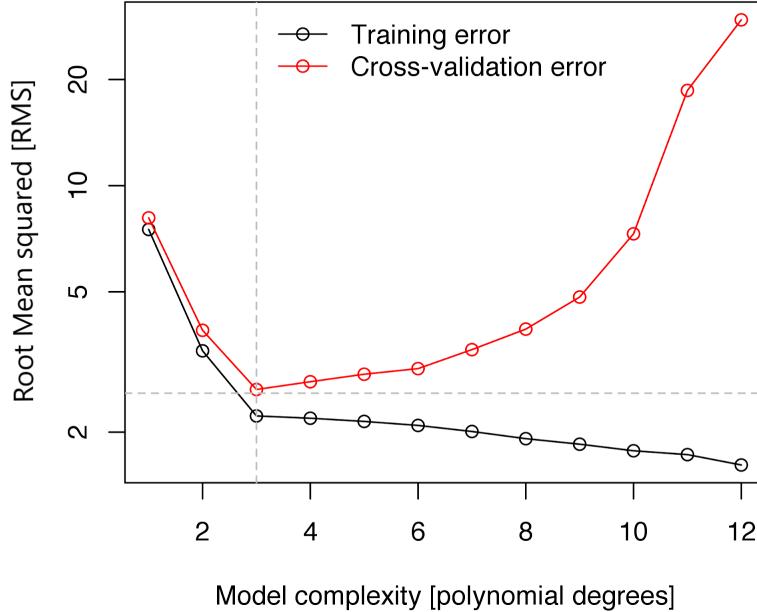
To overcome the under/over-fitting problem we can **split the dataset in two subset**,

- **Training set:** set is used to learn the model's parameters.
- **Test set:** set is used to test model performance on unseen data. This is **very helpful for complexity selection**.

A **good error function for testing the complexity** is:

$$E_{RMS} = \sqrt{\frac{2RSS(\hat{w})}{N}} \quad (18)$$

Differently from the loss function used at training time E_{RMS} is **not monotonically decreasing with model complexity**. It has a **U shape** and the **minimum** corresponds to the **optimal model complexity**.



But what happens to the parameters when the model gets more complex?

| | $M = 0$ | $M = 1$ | $M = 3$ | $M = 9$ |
|-------------|---------|---------|---------|-------------|
| \hat{w}_0 | 0.19 | 0.82 | 0.31 | 0.35 |
| \hat{w}_1 | | -1.27 | 7.99 | 232.37 |
| \hat{w}_2 | | | -25.43 | -5321.83 |
| \hat{w}_3 | | | | 48568.31 |
| \hat{w}_4 | | | | -231639.30 |
| \hat{w}_5 | | | | 640042.26 |
| \hat{w}_6 | | | | -1061800.52 |
| \hat{w}_7 | | | | 1042400.18 |
| \hat{w}_8 | | | | -557682.99 |
| \hat{w}_9 | | | | 125201.43 |

We can notice that **increasing the number of parameters**, their value increases significantly, since the **parameters struggle to shape the function in order to interpolate the samples of the training set**, and more the parameters are more they try to increase the frequency of the function in changing values. This is an **evidence of overfitting**. This phenomenon is very typical and is a sign of the model trying to overfit data. The **idea** that comes from this analysis is to use many parameter but **do not let their value to increase over a certain threshold**. Indeed, in this way, **even with many parameters the function will be smooth**, since are the **large values** of parameters to cause the function to **oscillate very rapidly with very high slopes**. More regular functions are obtained with smaller values of parameters. The **idea** is not to simply **minimize the empirical loss function**, but I want to **do it with a regular function that is smoothly changing**. And this process is called **Regularization**.

NOTE Hence, another indicator for poor generalization is parameters absolute value. If we have very large parameters it means that the model oscillate very rapidly.

3.4 Regularization

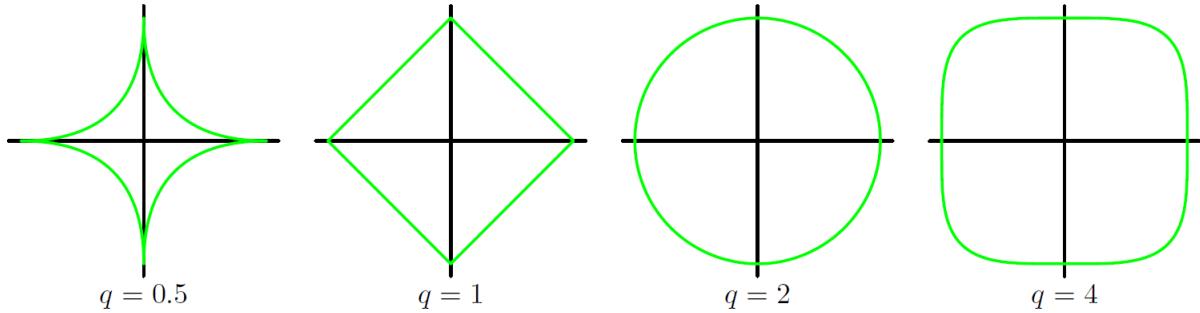
One of the major aspects of training your machine learning model is **avoiding overfitting**. The model will have a **low accuracy if it is overfitting**. This happens because your model is trying too hard to capture the information in the dataset **ending up in capturing the noise in your training dataset**. One way to avoid overfitting is using **cross validation**. Another way to reduce overfitting is **constrain/regularize or shrink the parameters estimates towards zero**. To do so, we can **change the loss function**

$$L(w) = \underbrace{L_D(w)}_{\text{error on data}} + \underbrace{\lambda L_W(w)}_{\text{loss on complexity}} \quad (19)$$

For tractability we take

$$L_D(w) = \frac{1}{2} RSS \quad L_W(w) = \frac{1}{2} \sum_{j=1}^M |w_j|^q, \quad q \in \mathbb{N}^+$$

As we can see L_W depends on parameter absolute value. This loss function component discourages high parameters values. So a parameters have to justify its high value by giving a very good contribution to L_D , otherwise the model is penalized. The **penalization** of L_W can be **controlled with λ (regularization coefficient)** and q . Furthermore we can interpret L_W as a constraint. If we consider the parameters space, L_w is **bounding the parameters at a certain distance from the origin**. The **distance is imposed by λ and q** . In particular, q **modifies the shape of the boundary** (constraint) in the parameters space. The most used value for q are 1 and 2.



Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity. However, the problem of determining the optimal model complexity is then shifted from one of finding the appropriate number of basis functions to one of determining a suitable value of the regularization coefficient λ .

3.4.1 Ridge regression

The idea of ridge regression (or weight decay) is to regularize the loss function. Instead of simply minimizing the empirical loss we change the loss function **adding a term that penalizes solution in which the weights have large values**:

$$L(w) = L_D(w) + \lambda L_W(w)$$

where $L_D(w)$ is the error on data (e.g. RSS), L_W is something that **consider the model complexity (e.g. the magnitude of the weights)**. In particular ridge regression has:

$$L_W(w) = \frac{1}{2} w^T w = \frac{\lambda}{2} \|w\|_2^2$$

where the l_2 – norm squared is simply the sum of the squared component. Hence, the loss function minimized is:

$$L(w) = \frac{1}{2} \sum_{i=1}^N (t_i - w^T \Phi(x_i))^2 + \frac{\lambda}{2} \|w\|_2^2$$

in this way we are saying: find the best solution **good in explaining the data** ($L_D(w) = \frac{1}{2} \sum_{i=1}^N (t_i - w^T \Phi(x_i))^2$) **but that has low values** ($L_W(w) = \frac{\lambda}{2} \|w\|_2^2$) since the L_W term is a penalizing one. Furthermore λ is an **hyperparameter of the model**, higher the λ **higher is the penalty** given to the higher values.

What happens **considering very large values of λ** ? All the parameters will be **zeros**, since the minimization of the weights will become the main objective since the **dominant term** in the loss function would be L_W , so minimizing that would mean to **learn the most regularized function possible**, i.e. the planar function. Hence, **lambda** express how much I want the estimated function to be **regular**:

- Increasing λ : the function becomes regularized becoming always nearest to the one with weights at 0.
- Decreasing λ : the function is allowed to explain the data.

This is called ridge regression (or weight decay) and it is a regularization (or parameter shrinkage) method. The **loss function is still quadratic in the in w and it has a closed form solution**:

$$w_{ridge} = (\lambda I + \Phi^T \Phi)^{-1} \Phi^T t$$

always found putting the gradient of the former loss function equal to zero. The solution is similar to the one of ordinary least square, indeed the **only one difference is that the inverted matrix this time is summed to the matrix λI before being inverted**. Furthermore, as it was for the loss function, having $\lambda = 0$ we would recover the ordinary square solution, and **increasing really much lambda the dominant term would be λI** . The interesting point of the closed form solution obtained is that **the matrix $\lambda I + \Phi^T \Phi$ is always invertible**, i.e. always non-singular (**differently for $\Phi^T \Phi$, that must be checked**). Indeed $\Phi^T \Phi$, as we have seen, is a square positive semi-definite matrix, so some eigenvalues may be zero, but we are summing up a diagonal matrix with all eigenvalues equal to λ , hence from a theorem we know that the eigenvalues of the sum of the two matrix are at least equal to λ ($eig \geq \lambda$), which means that **choosing $\lambda > 0$ we couldn't have eigenvalues equal to 0 even when you have more features than samples**. This fact is very interesting since it **allow you to consider a very large hypothesis space** (i.e. many features), **avoiding the consequent phenomenon** (due to hypotheses space too large) **overfitting thanks to the regularization**, finding a compromise by properly setting λ , between having the function properly explaining the data avoiding very irregular and quickly changing ("oscillating") models. **Is true that you can have more features than samples**, having a closed form solution, **but we have to pay attention to the**

computational complexity remembering that the **inversion of the matrix is cubic in the number of parameters/features**. Again, having problem of inversion complexity we could resort to **stochastic gradient descent** always keeping in mind that the **complexity is more than quadratic** in the number of features.

NOTE Ridge regression combines the sum of square errors with the penalization for large values.

NOTE The use of regularization comes from the fact that overfitting could be associated with large parameters.

NOTE Ridge regularization allows to add a large number of features. λ can be then used as a trade off between explaining the data and overfitting data. **Choosing λ too large would lead to underfitting, instead with very low value the risk is overfitting**, hence the choice of model selection in this case is to elect the right λ for the problem.

Example Zero regularization, i.e. $\lambda = 0$, we have overfitting; increasing regularization overfitting is reduced and can be noticed by the value of $\|w\|_2^2$ that is regularized: higher lambda higher regularization, so we can reach a situation in which we don't have overfitting with the same number of parameter than before. **Exaggerating the value of lambda is not good, since the function learned would be always closer to the zero one**. For the same polynomial of degree 15 the trend of the parameters w , changing λ is the following one:

NOTE In general would be better to have no regularization and a low number of features, since **regularization introduce a bias**, but the **feasibility depends** on:

- how many samples you have: with a **low number of samples is better to use regularization**
- the knowledge about the problem: **to have a low number of features you have to have idea of what are the features good for the problem**. Not having idea of them is better to use a lot of them and then regularize. Instead having the knowledge is better to have a small set of features and no regularization (having a good knowledge of the problem may mean that the problem is simple, and maybe ML is not necessary; ML is used to search complex function in complex feature spaces).

NOTE In ridge regression being $\lambda I + \Phi^T \Phi$ always non singular there is **always a single global optima, never multiple (global) optimal solutions and no local optima**.

NOTE The **advantage** of ridge regression is the **existence of a closed form solution**. Different functions have different properties.

Let's see how regularization changes the error of the learned function, where the test error E_{RMS} is used to find the optimal complexity, i.e. is considered an approximation of the one of the true function:

- For low values of λ the **training error is very small, instead the test error is high**, so the training error is **completely uncorrelated** to the test error, **meaning that the learned function is overfitting data**. Indeed, the error over the training data is almost null, and the one over test data, i.e. the **generalization error is high**. (**Zero bias, high variance**). [0]
- **Regularization**, i.e. higher value of λ the training error get worse, indeed the loss function is not considering anymore only data but also parameters magnitude. So the **bias introduced by the regularization let the test error to reduce and the training error to become closer to the latter, hence the bias is beneficial**. (**Increasing bias, decreasing variance**). [0, -25]
- **Too much regularization** cause an **increase the training and test error becomes more and more correlated**, i.e. the regularization leads to a too smooth model that is **underfitting data when both train error and test error are increasing** (underfitting more and more as λ increases). (**High bias, Low variance**). [-25, -20]

Remember that ideally we do not want to minimize the training error but the test error, that approximate the error of the true function, in other words the minimization should be done on the real loss function (unknown) and not on the empirical one.

NOTE The variance is index of the generalization capability of the model, indeed having high variance would mean very different prediction over different data.

NOTE Increasing regularization (i.e. λ) is like **making the hypothesis space simpler**, since increasing lambda **penalize solution with high parameters values**, hence is like the learning process won't (this solution are really penalized so they are very scarcely considered) consider those solution, that is like having a smaller hypothesis space. As already said ridge regression is imposing a boundary that is more and more strict as λ increases.

NOTE We could use as much features as we want with ridge regression (for the positive definiteness property) but we must take into account the complexity of the solution and the fact that a good model has low test error, that is not so easy to achieve.

NOTE The fact that test error and training error are close means that the variance of the error of the model is low, and we know that a low variance of the loss means that the model is good so the empirical training loss was a good estimation of the real loss. When the variance is high it means that the empirical loss was a bad approximation of the true loss. The empirical training loss can be used to measure the bias (0 loss means 0 bias) and the gap between test and training error can be used to measure the variance (far means high variance, near means low variance).

NOTE Recalling the general loss function that consider also model complexity considering $q = 2$ we have Ridge regression.

$$\begin{aligned} L_W(w) &= \frac{1}{2} w^T w = \frac{1}{2} \|w\|_2^2 \\ L(w) &= \frac{1}{2} \sum_{j=1}^N (t_i - w^T \Phi(x_i))^2 + \frac{\lambda}{2} w^T w \\ &= \frac{1}{2} (t - \Phi w)^T (t - \Phi w) + \frac{\lambda}{2} w^T w \end{aligned} \quad (20)$$

NOTE The **main advantage** of ridge regression is that the **loss function is still quadratic in w** , so we can **still** derive a **closed form solution**.

$$\hat{w}_{ridge} = (\lambda I + \Phi^T \Phi)^{-1} \Phi^T t \quad (21)$$

NOTE The eigenvalues of $(\lambda I + \Phi^T \Phi)$ are still greater or equal to zero because $\Phi^T \Phi$ is positive semi-definite and λI simply imposes a positive lower bound to the eigenvalues ($eig \geq \lambda$). $(\lambda I + \Phi^T \Phi)$ is still positive semi-definite and so invertible.

3.4.2 Lasso

Another popular regularization method is lasso:

$$L_W(w) = \frac{1}{2} \sum_{i=1}^N (t_i - w^T \Phi(x_i))^2 + \frac{\lambda}{2} \|w\|_1$$

where $\|w\|_1 = \sum_{j=1}^M |w_j|$, i.e. w.r.t. ridge regression it uses the $l_1 - norm$, instead of the $l_2 - norm$, that is simply the sum of the absolute values of the weights. **Again it penalize the models with high values of parameters.** The **disadvantage** of this approach is that differently from ridge regression, **lasso is non-linear in the t_i and no closed-form solution exists but the problem is still a quadratic programming problem**, so through quadratic programming the solution could still be found the optimal solution. There are still **no local optima**, but **to find the unique global optima we must resort to iterative**

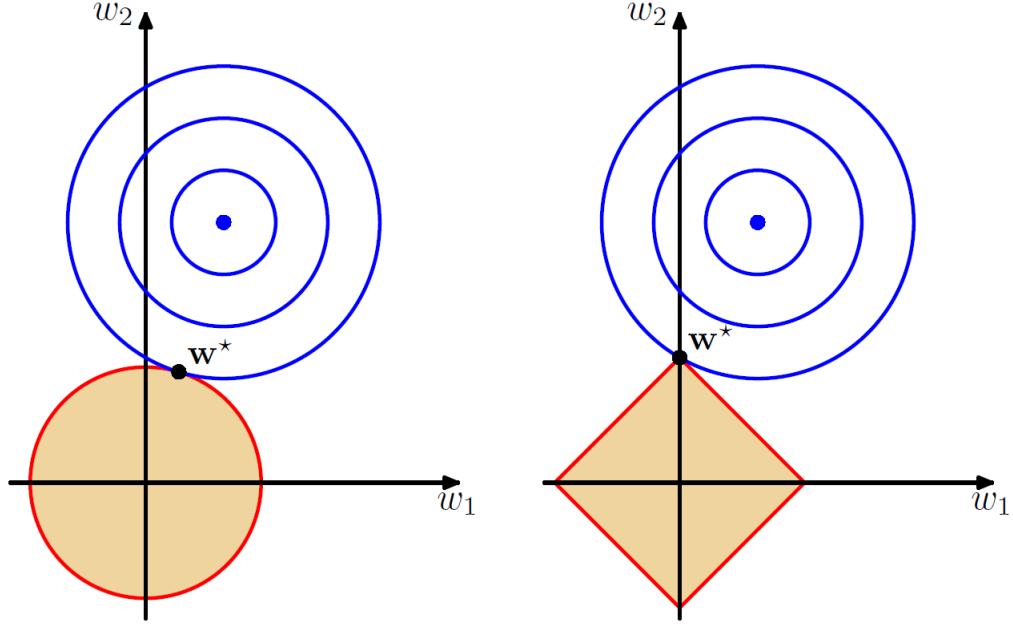


Figure 11: The red lines in the figure represents $\sum_{j=1}^M |w_j|^q$ respectively for $q = 2$ and $q = 1$. The blue lines are the unregularized error functions.

approaches to find it, and not a closed-form solution. Nonetheless, w.r.t. ridge regression it has the advantage of making some weights equal to zero for values of λ sufficiently large, this creates sparse models, with parameters to 0 associated to not useful features: differently from ridge, where parameters decrease to 0, some parameters are set exactly to 0 without decreasing (only some are decreased to 0). This characteristic is very useful since it allow you to make feature selection: having $w_i = 0$ means that the corresponding features are useless.

Lasso tends to generate sparser solutions than quadratic regularizer (i.e. ridge regression). In the latter figure, which represent the space of parameters in a linear model: blue lines represent the loss function we want to minimize (imagine it as a quadratic function, with vertex in the point) where the blue dot is the solution using ordinary least square, and the circle are isolines that show how the function increase quadratically.

Ridge and Lasso could be seen as restricted optimization problems: in a constrained optimization problem you can take the constraint and then put them in the objective function using lagrangian multipliers. Looking at ridge and lasso the regularization term is like a lagrangian multiplier ($\lambda/2$, with $\|w\|_2^2$ and $\|w\|_1$ the respective constraint), so minimizing the loss is like asking, for a certain value of lambda, that the weights respect the constraint:

- Lasso:

$$\|w\|_1 < \text{constant}_\lambda$$

- Ridge:

$$\|w\|_2^2 < \text{constant}_\lambda$$

where the constant changes according to λ . These constraints in the parameter space can be seen as (in a 2 dimensional parameter space):

- Lasso:

$$\|w_0\|_1 + \|w_1\|_1 < \text{constant}$$

the sum of absolute values define a **rhomoidal shape** whose radius varies with λ^{-1} , hence the **radius increases as lambda decreases**.

- Ridge:

$$\|w_0\|_2^2 + \|w_1\|_2^2 < \text{constant}$$

hyper-sphere centered in 0 (circle in 2 dimensional space) whose radius varies with λ^{-1} , hence the **radius increases as lambda decreases**.

Inside this constrained area the solution w^* is searched and it is the one that minimize the loss function, i.e. the **point closest to the local optimum** (if the areas are infinite both will learn the blue point). In general, the solution learnt is something between the unconstrained solution (blue point) and the origin of the system. **In the lasso method the solution may end up on a vertex of the rhomoidal shape, so this means that some parameters will be null.** So the shape of the lasso regularization will make the parameters equal to 0, that is **more difficult to obtain with ridge regularization** without increasing λ very much (the blue dot must be aligned with one of the axes).

NOTE The constraint are around the origin, since regularization is asking to limit the magnitude of the parameters. In particular **higher lambda smaller the area** indeed **higher lambda means more regularization**.

NOTE Considering the ordinary least square optimization function and the constraint of l_1 or l_2 norm, by solving them with the Lagrange multipliers we obtain the expression of lasso and ridge methods.

Between the two methods **there isn't one more effective, in generalb**, against overfitting. But you may prefer lasso if you want to apply feature selection, and ridge to have a closed-form solution. Usually, after we have done feature selection with lasso we pass to other approaches. For what concerns **performance they are more or less equivalent**, there is not one superior to the other.

NOTE Recalling the general loss function that consider also model complexity considering $q = 1$ we have the so called Lasso.

$$\begin{aligned}
 L_W(w) &= \frac{1}{2} \sum_{j=1}^M |w_j| = \frac{1}{2} \|w\|_1 \\
 L(w) &= \frac{1}{2} \sum_{j=1}^N (t_i - w^T \Phi(x_i))^2 + \frac{\lambda}{2} \sum_{j=1}^M |w_j| \\
 &= \frac{1}{2} \sum_{j=1}^N (t_i - w^T \Phi(x_i))^2 + \frac{\lambda}{2} \|w\|_1
 \end{aligned} \tag{22}$$

Differently from ridge regression, lasso is not linear. No closed form solution exists because we have the absolute value operator in L_W . In contrast, a very good advantage is the capability to make some weights equal to zero for values of λ large enough. This means that lasso leads to sparser models (A model is sparse if some parameters tend to zero, eliminating some features from the model)

NOTE Regularization represent the complexity of the model in the loss function: **small weights imply that the function will be more flat** (so it **cannot overfit data**, with the risk of **underfit data**), instead with **high weights the function will have great slopes**, and change quickly being able to **overfit data**.

3.5 Bayesian Linear regression

Up to now what we have seen was a **frequentist approach**. A frequentist approach we try to maximize the likelihood of the data you are observing, indeed as we have seen the OLS (ordinary least square) solution can be seen as the maximization of the likelihood of the data under the Gaussian model, so it simply tries to find the best model to explain the training data, i.e. **only relaying on data and nothing else**.

The Bayesian approach, that instead can be considered as the **opposite of the frequentist approach**, starts from another perspective: **introduces a kind of prior information about the solution of the problem inside the estimation**. This is a **probabilistic approach**, i.e. **the knowledge about the world is expressed in a probabilistic way**.

We are explicitly uncertain of the parameters of the model: while in the frequentist approach the optimal weights are computed as the one that maximize the likelihood, in the Bayesian approach what is **computed is a distribution of probability of the weights, not just a value**. Hence the model expresses the knowledge qualitatively, with unknown parameters, and the **assumptions about the unknown parameters are specified by the prior distribution over those parameters before seeing the data**. The prior distribution will have high probability to model reasonable for the problem considered and low probability for parameters that are assumed to be not reasonable for the problem. For exam-

ple having **no knowledge** about the problem, will **imply a uniform prior distribution**, so that all the models (set of parameters) are **equally likely**.

Using a uniform prior distribution the bayesian and the frequentist approach are the same, in the bayesian setting you have the possibility to introduce more information if you have them.

NOTE Prior distribution represent (encodes) your knowledge before seeing the data.

Once the prior has been defined you observe the data and you will combine your prior knowledge and the information inside the data, i.e. updating the prior distribution using observed data, obtaining the posterior probability distribution for the parameters.

NOTE The posterior distribution is combination of the prior and the likelihood of the data.

The **advantage** is that once you have the posterior distribution:

- Make predictions by averaging over the posterior distribution.
- Examine/Account for uncertainty in the parameter values: unlike before we can asses the uncertainty because we have the distribution over all the possible models, each of them will produce a different prediction. So having the different prediction of each model, and having different probability for each prediction to be true, we can have the distribution of the prediction i.e. we can measure the uncertainty of the model prediction.
- Make decisions by minimizing expected posterior loss: we can use the uncertainty measure to make decisions.

All of this comes with a cost, indeed the disadvantage is that computationally speaking the bayesian approach is more expansive than the frequentist one.

The posterior distribution of the model parameters can be found by combining the prior with the likelihood for the parameters given data. This is accomplished using the Bayes' rule:

$$P(\text{parameters}|\text{data}) = \frac{P(\text{data}|\text{parameters})P(\text{parameters})}{P(\text{data})}$$

$$\implies p(w|\mathcal{D}) = \frac{p(\mathcal{D}|w)P(w)}{P(\mathcal{D})}$$

- $p(w|D)$ is the **posterior probability** of parameters w given training data D

- $p(D|w)$ is the probability (**likelihood**) of observing D given w
- $P(w)$ is the **prior probability** over the parameters
- $P(D)$ is the **marginal likelihood (normalizing constant)**:

$$P(D) = \int p(D|w)P(w)dw$$

What we want to estimate is the probability that our parameters explain the model, given the observed data. In the Bayes' rule $P(D|w)$ is the likelihood, so how much the data are likely to be generated by those parameters. Instead, $P(w)$ represent the prior knowledge of the problem, (i.e. the hypotheses space). The term $P(D)$ is a normalization term to have the integral of $P(w|D)$ equal to one, being a probability density function.

Stating the Bayes' rule in words: the posterior probability is \propto likelihood [$P(D|w)$] x prior [$P(w)$].

Since we want the **most probable value of w given the data** we take the **maximum a posteriori (MAP)**, i.e. the value w that has the maximum posterior probability $P(w|D)$, i.e. the **mode of the posterior**. So MAP maximize $P(w|D)$. Considering the frequentist approach instead we maximize only the likelihood term $P(D|w)$, when we use OLS assuming Gaussian noise. The difference is that the bayesian approach consider also the prior $P(w)$, but considering a uniform prior i.e. each vector w has the same probability, maximizing the likelihood or the posterior probability is the same, hence the two approaches in that case become the same.

As we can imagine Bayesian regression thanks to the **prior allow to contrast the problem of overfitting**. As we have seen OLS, has the risk of overfitting, so we have to use other techniques (ridge and lasso) that introduce other terms in the loss function. Instead in **Bayesian linear regression the regularization is not needed**: having knowledge about the problem and considering that inside the prior distribution has the outcome of regularization, i.e. the prior distribution acts as a regularizer.

Hence another approach to avoid the over-fitting problem of ML is to use **Bayesian linear regression**. In the Bayesian approach the **parameters** of the model are considered as **drawn** from some distribution, the **distribution of probability over the space of parameters**. Let's assume a specific distribution, one of the most widely used is the **Gaussian distribution**. So, assuming **Gaussian likelihood model**, the **conjugate prior is Gaussian** too:

$$p(w) = \mathcal{N}(w|w_0, S_0)$$

i.e. a Gaussian over the vector w , where w_0 is the mean of the distribution and S_0 is the covariance function of the distribution. The reason of using a Gaussian is the following one: considering that the posterior distribution is obtained by (without considering the normalizing term) the multiplication of two probabilities, after we have computed the posterior

$P(w|D_1)$ over the data D_1 , considering new data D_2 , the posterior we have computed on D_1 can be used as prior for the posterior $P(w|D_2)$ over D_2 ; this can be iterated if the data is collected in a sequential way. One desired property is that the **product** of this distribution $p(D|w)P(w)$ produces a new distribution that is the **same family of the prior**, in other words I would like the posterior distribution belong to the same family of distribution of the prior, indeed if this is true collecting new data from the process, I can take the posterior I have to use it as a prior (e.g. if the prior is a gaussian and the posterior is a gaussian too). When the **distribution of the posterior is the same as the distribution of the prior**, we say that the **distribution of the prior and the distribution of the likelihood are conjugate**, i.e. their product generates a probability distribution of the same family as the one of the prior ($A * B = A$, with A and B types of distributions, e.g. prior gaussian, likelihood gaussian then the product of two gaussian is still a gaussian; e.g. prior beta distribution, likelihood bernoulli distribution then the posterior is beta).

NOTE Supposing not having all the training set at the beginning you receive the data in a sequential way (e.g. working on a time series): you start with a prior and compute the posterior, then this distribution takes the place of the prior when new data is collected, indeed the posterior distribution is an update over the prior distribution, combined with the information from data i.e. the likelihood of having a certain model that produces that data.

NOTE The iterative model update can be done also for frequentist approaches, but usually in those cases all the data are collected together and the model is retrained, or you use the gradient approach to adjust the values. But **in bayesian approach the iterative procedure is more straightforward**.

NOTE More the data that you observe less important will be the importance of the prior, the prior is very important with very few data. Indeed, the prior act as a regularizer reducing the variance, i.e. the uncertainty when you have few data (increasing the bias the variance decreases). Furthermore, if at the beginning you have a lot of data, you do not want to add much bias by choosing a (maybe) wrong prior, is better to choose a prior that is almost uniform. **Having enough data to explain the process the frequentist approach should be preferred**, since the knowledge in the prior would be not needed and would introduce a bias. The problem for frequentist approach comes when you have not enough data for searching in a very large hypotheses space, so the bayesian approach could help (variance cannot be reduced with data but increasing the bias).

So, having the likelihood Gaussian is interesting case since the conjugate prior is Gaussian too and given the data D , the posterior is still Gaussian:

$$p(w|t, \Phi, \sigma^2) \propto \underbrace{\mathcal{N}(t|\Phi w, \sigma^2 I)}_{likelihood} \underbrace{\mathcal{N}(w|w_0, S_0)}_{prior} = \mathcal{N}(w|w_N, S_N)$$

$$w_N = S_N \left(S_0^{-1} W_0 + \frac{\Phi^T t}{\sigma^2} \right)$$

$$S_N^{-1} = S_0^{-1} + \frac{\Phi^T \Phi}{\sigma^2}$$

i.e. we can **compute the posterior distribution in the closed form**, where w_N and S_N are the mean and the variance matrix of the Gaussian of the posterior.

NOTE If you want a closed form solution that generates the posterior probability over a set of training data $\{D_i\}$ then, **the prior and the posterior should belong to the same distribution class, otherwise the scheme cannot be iterated**.

NOTE If the posterior distribution is of the same family of the one of the prior the two distributions are called conjugate, and the prior is called conjugate prior.

Is important to notice that **with the prior you can make a huge damage**: using a **very strong prior**, i.e. a prior with very low variance, in a bad region, you will introduce a **very bad bias** that needs a lot of samples to delete.

NOTE **With a lot of data** is usually better to perform **OLS**, since is **computationally cheaper**. The quantity of "a lot of data" depends on the sizes of the hypotheses space, i.e. on the number of features considered. But having a very good prior could encourage you to use the bayesian approach.

NOTE **OLS relays only on data**, so if the data is enough it overfits, so we must add other element (regularization). On the other hand the **bayesian approach relays both on data and the prior knowledge**, that **works as a regularizer** introducing a bias that could be good or bad.

NOTE The choice between a frequentist approach and the bayesian one is made considering the following things:

- If you need to have a distribution of probability over the prediction and not just a point prediction.
- If you have knowledge about the problem.
- Computational resources.

is not just related to the accuracy since, as we will see, we deal with that in other way.

NOTE The **prior** $p(w)$ acts as a **regularization** since you are **putting a bias leading the model in a certain direction, i.e. where your knowledge knows that is more probable to have the model**. Is not like bounding the hypotheses space but, similarly to ridge and lasso, using a prior we are **penalizing some models**, by saying that they have a low probability to be the one we are looking for. With the prior we are saying that, instead of giving the same probability to all the hypotheses space, some part of the hypotheses space are more likely and some other are less likely.

NOTE If the prior is totally wrong the solution would be totally screwed up. One **way to understand that the prior is wrong is to see if the prior is trying to change a lot w.r.t.** the prior since data information (i.e. likelihood) are transforming the prior in something really different; **of course to be sure of this you would need a lot of data.**

NOTE The difficulty for the establishment of the prior depends on the problem. With the prior you can tell the procedure to look for the model in a certain zone of the parameter space, since you may know is not far from there. For example, you may know that some parameters cannot have negative values, so with the prior you can put this knowledge inside the algorithm.

Let's analyze the **Gaussian distribution formula for the posterior**:

$$p(w|t, \Phi, \sigma^2) \propto \mathcal{N}(w|w_0, S_0) \mathcal{N}(t|\Phi w, \sigma^2 I) = \mathcal{N}(w|w_N, S_N)$$

i.e. is proportional to the product of the prior and the **likelihood whose mean is given by the model prediction and the variance matrix is diagonal with values σ^2 , i.e. is a white noise (σ^2 is assumed to be known)**.

$$w_N = S_N \left(S_0^{-1} w_0 + \frac{\Phi^T t}{\sigma^2} \right)$$

$$S_N^{-1} = S_0^{-1} + \frac{\Phi^T \Phi}{\sigma^2}$$

where the inverse S_N^{-1} of the variance matrix is equal to the inverse of the covariance of the prior plus a term that we already know since it is the quadratic matrix of our dataset divided by σ^2 that is the noise that we have in the likelihood. By looking at the latter expression and substituting its inverse in the former we will find something we could expect.

Example - uniform prior Supposing that the prior is uniform then its variance goes to ∞ , so S_0^{-1} goes to 0 since we are inverting something that is infinite. As expected the prior term inside both w_N and S_N^{-1} disappears obtaining:

$$S_N^{-1} = \frac{\Phi^T \Phi}{\sigma^2}$$

$$w_N = S_N \frac{\Phi^T t}{\sigma^2} = \sigma^2 (\Phi^T \Phi)^{-1} \frac{\Phi^T t}{\sigma^2} = (\Phi^T \Phi)^{-1} \Phi^T t$$

that is the closed form solution of OLS. So, as expected, using an uniform prior i.e. a prior with infinite variance you go back to the Maximum Likelihood solution.

NOTE Saying that the prior variance is ∞ means that we do not have any prior knowledge about the model, so the probability of having a certain model w (i.e. the prior $p(w)$) is equal for each model, i.e. all the model are equally likely to be the one we are looking for.

From the latter example we deduce that **having a prior with variance lower than ∞** , we obtain a **solution that deviates from the one of maximum likelihood** and it **would be a kind of mixing of the OLS solution represented by the term $\Phi^T t / \sigma^2$ with the mean of the prior w_0** . The latter **combination between the OLS solution and the mean of the prior would be regulated by the prior variances S_0** , present even in S_N .

The MAP is the mode of the posterior. Furthermore, in the Gaussian distribution the mode coincides with the mean of the distribution, so the MAP will coincide with the mean of the distribution w_N . Hence, w_N is the **MAP (maximum a posteriori) estimator**. If the prior has infinite variance, as we have seen, w_N reduces to the maximum likelihood estimator, which means that the MAP estimator will coincide with the maximum likelihood estimator.

Furthermore, (still with a Gaussian likelihood) using a **Gaussian prior with $w_0 = 0$ and $S_0 = \tau^2 I$ (i.e. the prior is a Gaussian white noise)**, then w_N reduces to the **ridge estimate**, where the parameter of ridge regression is given by:

$$\lambda = \frac{\sigma^2}{\tau^2}$$

Indeed **using zero mean prior we are forcing with our prior that the model is around 0**, and how much we are enforcing this knowledge is regulated by the covariance matrix: larger the variance less the regularization, indeed τ is at the denominator of the λ parameter of ridge regression, hence **having τ^2 very close to zero is like saying in the prior that the model is very close to 0, so the regularization is very strong**. In other words, the regularization is inversely proportional to τ^2 , i.e. to the variance of the gaussian prior, indeed having a small variance of the prior with mean 0 we are saying that we are pretty sure that the model is close to zero, i.e. the value of the parameters are mainly distributed around zero, **giving less relevance to the parameters with higher values, like we do in ridge regression**. Furthermore, since a small variance τ^2 implies a high regularization, it also means that the importance given to the likelihood of data is small that could be seen as being more focused in obtaining a smooth function (regularization term) than reducing the error over the observed data (equivalent to the OLS term).

NOTE Using a small variance imply a greater interest in regularization than in the data likelihood. This could be done if we do not trust the data since they are too low with respect to the dimension of the hypotheses space.

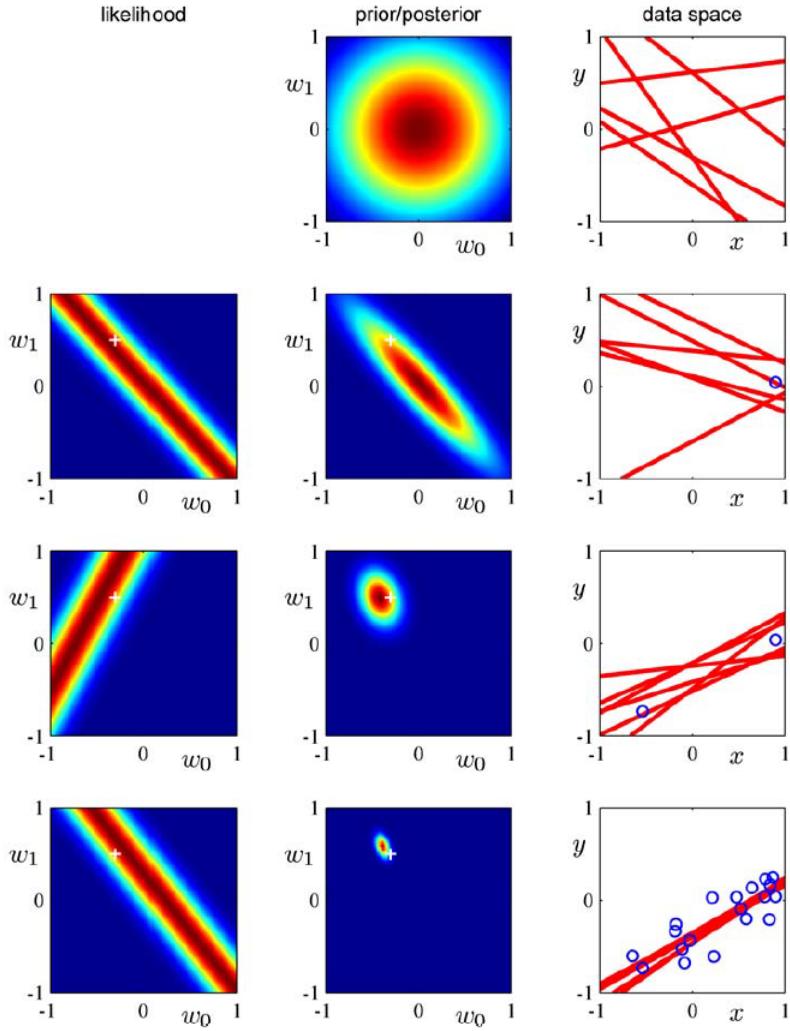
Example (1D) Data are generated from the (unknown) function:

$$t(x) = -0.3 + 0.5x + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, 0.04)$. The x values are generated uniformly over $[-1, 1]$. So we want to learn a model, and we consider the following:

$$y(x, w) = w_0 + w_1 x$$

Actually the hypotheses space of the models contains the true function, hence with infinite samples we could learn the true value of the true model. We assume to know that the noise of the true function has variance $\sigma^2 = 0.04$ and we consider the prior a gaussian distribution with 0 mean and $\tau^2 = 0.5$ as the parameter of the covariance matrix ($S_0 = \tau^2 I$). Hence we are applying ridge regression to the problem with parameter $\lambda = \sigma^2 / \tau^2 = 0.04 / 0.5 = 0.08$. The prior can be represented as the following: where the different colors represent magnitude of the probability given to that points in the parameter space by the distribution: from red (high) to blue (low). The probability decreases as we go far from the origin, and since the covariance function is diagonal the level sets of the gaussian are circles. Sampling some model from the red zone of the distribution, i.e. a point in the parameter space, I obtain a line in the input output space t-x: Now let's see what happens once I look at the data:



where the blue circles represent the data observed and the white cross represent the true model. The likelihood images show the probability of seeing the data observed for each point in the parameter space, in this case for each pair (w_0, w_1) . The posterior, is computed by the multiplication of the prior and the likelihood, but since are both gaussian also the posterior would be gaussian so it can be used (as we have seen) as the prior when new data is collected and observed. So in the figure the likelihood of one row, i.e. on the new data, is multiplied for the prior/posterior one the previous line, i.e. on the data observed before. At each step the update of the posterior generate a distribution that converges towards the true function, since it is contained in the hypotheses space. The convergence behaviour could be noticed even in the image with red lines, that shows some sampled models from the red zone of the posterior distribution, indeed as the **posterior distribution reduces its variance** the red line are closer to each other. An interesting thing to point out is that if I want to do a prediction for a certain input x , I could obtain a prediction from all the models weighted by the probability that each model is the true models (assuming that the number of model is finite).

NOTE The prior introduces a bias, indeed is like ridge regression.

NOTE Obviously if the real function is linear it should be detected with only two points as only one line passes from two points, but this happens in absence of noise. **The function, that represent a phenomenon we are trying to study, is affected by noise, that's why an infinite number of data would be needed to look for the true function in an hypotheses space that contains it.**

A nice thing is that in the Gaussian case also the **posterior predictive distribution** i.e. **the distribution of the target values given a certain posterior** (and input obviously), is a **Gaussian**:

$$\begin{aligned} p(t|x, D, \sigma^2) &= \int \mathcal{N}(t|w^T\phi(x), \sigma^2) \mathcal{N}(w|w_N, S_N) dw \\ &= \mathcal{N}(t|w_N^T\phi, \sigma_N^2) \\ \sigma_N^2 &= \sigma^2 + \phi(x)^T S_N \phi(x) \end{aligned}$$

It tells the **probability distribution of a target** is the **expected value** (\int) **considering all the models'** (dw) **prediction** $\mathcal{N}(t|w^T\phi(x), \sigma^2)$ (i.e. gaussian distribution with mean the prediction of the model and variance equal to the data noise) **proportionally (weighted) to the probability of that model to be the real one in the posterior distribution** $\mathcal{N}(w|w_N, S_N)$ (i.e. the posterior Gaussian distribution). In particular the posterior predictive distribution is a **Gaussian** of **mean** given by the **prediction of the MAP** ($w_N\phi(x)$) and of **variance**:

$$\sigma_N^2(x) = \underbrace{\sigma^2}_{\text{noise in the target values}} + \underbrace{\Phi(x)^T S_N \Phi(x)}_{\substack{\text{Uncertainty associated} \\ \text{with parameters values (model)}}} \quad (23)$$

i.e. the sum of the intrinsic noise in the data and the uncertainty in the model estimation (i.e. uncertainty of the parameters) gives the uncertainty of the prediction.

NOTE The prediction of the models is weighted by the posterior probability of the models, i.e. how likely the model is the true one.

NOTE The **model considered to find the prediction distribution** are infinite, since actually the integral is over an infinite number of elements. But **luckily if the posterior is Gaussian, then there is the closed form solution**, hence the **integral calculation can be simplified using directly the formula** $\mathcal{N}(t|w_N^T\phi(x), \sigma_N^2)$, that is the value you would obtain computing the integral over infinite models i.e. the **closed form with infinite models**.

Hence **having a Gaussian prior and a Gaussian likelihood, you have a Gaussian**

posterior distribution and a Gaussian posterior prediction distribution. Furthermore the distribution of the prediction have a closed form with:

- **MEAN:** prediction form MAP model.
- **COVARIANCE matrix:** sum of data uncertainty and model uncertainty.

As the number of samples goes to infinite ($N \rightarrow \infty$) the uncertainty of the model goes to 0, so the noise of the prediction will tend to the noise of the target values, that cannot be eliminated since it is intrinsic to data. This behaviour can be figured in the upper posterior plot, indeed in case of infinite samples the posterior distribution will tend to the real function, becoming a delta of dirac in the point corresponding to the weights of the real function.

NOTE The noise in the target values, i.e. the intrinsic noise of data, if not known is estimated from the data, using empirical formulas.

NOTE Having a Gaussian likelihood is like assuming you have a Gaussian error.

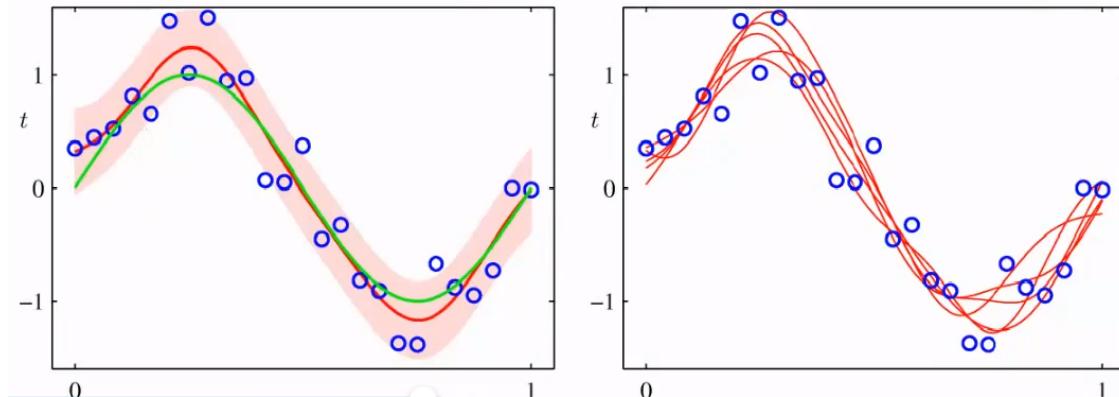
NOTE The MAP model is the mean of the posteriori distribution.

NOTE We are more interested in the predictive distribution, than the correct model that does not exist. What exist is the knowledge about the prediction and this is never a point in the parameter space, i.e. the correct model. Hence you should never provide a point estimate, since if you provide a point estimate, means that you are sure of your prediction but you are never sure of your prediction, so is much more meaningful to produce as output a distribution (in many cases not so effective) or at least a confidence interval. Indeed, you have to encode your uncertainty in some way, you have to make evident that you are not sure about the prediction and is important to know how confident it is a prediction. So what we are looking for is not the correct model but what would be a reasonable range of values for the prediction. If the maximum likelihood model produces only a single solution and not distribution, does it means that it is never used? No, there is also a way to have not proper distribution but confidence level also for maximum likelihood.

NOTE The posterior distribution is the probability distribution of the model given a set of data, i.e. given a set of data it associates to each point of the hypotheses space a probability of the corresponding parameters value to be the one that describe the true function. The prediction distribution instead is the probability distribution of the target values given a certain input and the posterior distribution i.e. given an input and a posterior associate to each value in the target space a probability to be the real target, combining the prediction coming from each model with the posterior value of each model, that is the probability of each model to be the true function, that

could be **seen** as the probability of the **reliability of that model**, i.e. giving different importance to the prediction of different models producing a distribution over predictions (e.g. $p = 0$ if none of the models predict that or if only model with probability 0 in the posterior predict that).

Example - predictive distribution In the following picture is showed a predictive distribution over the input output space: where the green line is the true function, the red line is the MAP and the pink area is the predictive distribution. We can notice how the **variance of the prediction distribution is much lower where we have a point of observed data** (blue circle) In the following picture instead you can see in red many samples, i.e. models, from the posterior distribution: noticing how in the zone with reduced variance the models are closer. Increasing the number of observed data the MAP, and the prediction distribution change, having always a reduced variance in the points where we have some observed data. The latter **phenomenon is due to the information coming from the likelihood that help to shape the prior obtaining the posterior that reduces the variance (i.e. the uncertainty) over observed data**. Observing enough data the MAP will be quite close to the real model, but the uncertainty (i.e. the variance) of the prediction ($\sigma_N^2 = \sigma_{data}^2 + \sigma_{model}^2$) will be never reduced under the uncertainty of the of the data since the noise in the data is intrinsic:



What are the **disadvantages** with the Bayesian approach?

- **MODELING CHALLENGES:** The first challenge is in **specifying suitable model (features**, as it was for the maximum likelihood) **and suitable prior distribution**.
 - A suitable model should admit all the possibilities that thought to be at all likely.
 - A suitable **prior** should be **informative, i.e. with low variance**, but the prior should be **in the correct region**, indeed not being able to place in a correct region would mean to enlarge the variance. Furthermore, **should avoid giving zero or very small probabilities to possible events** indeed in this case you would need a very large amount of samples to forget the prior, but

should also avoid spreading out the probability over all possibilities. To avoid uninformative priors (i.e. with high variance) we may need to point out the relationship/dependencies between the parameters of the model, restricting the hypotheses space to represent more reasonable models. [One strategy is to introduce latent variables into the model and hyperparameters into the prior. Both of these represent the ways of modeling dependencies in a tractable way.]

- **COMPUTATIONAL CHALLENGES:** The other big challenge is **computing the posterior distribution**. There are several approaches:

- **Analytical integration:** If we use “conjugate priors” (like with Gaussian distributions, beta-Bernoulli,...), the posterior distribution can be **computed analytically i.e. with a closed form solution**. Only works for **simple models**, not with more complex models where we may want to combine different distributions to represent the posterior distribution.
- **Gaussian (Laplace) approximation:** even if the problem is not Gaussian, you approximate the posterior distribution with a Gaussian i.e. **starting from a posterior that is not Gaussian you try to find the best Gaussian that explains the product of the likelihood and the prior**. Works well when there **a lot of data compared to the model complexity**, so that the huge amount of data could compensate the complexity of the model.
- **Monte Carlo integration:** Once we have a sample from the posterior distribution, we can do many things. Currently, the common approach is **Markov Chain Monte Carlo (MCMC)**, that consists in **simulating a Markov chain that converges to the posterior distribution**. You perform a random walk in the space of parameters (models) in a particular way that guarantee you that the walk will converge to the posterior distribution from which you want to sample. In this way the **distribution obtained** is not a continuous one but you will have a **discrete** one composed by some samples, that **can be used then to compute your predictions** (the **integral will become a sum over the models sampled**). So instead of having infinite models you will have a **finite number of models, extracted from the posterior distribution**, i.e. we are discretizing the posterior distribution, **approximating the posterior prediction distribution**.
- **Variational approximation:** A cleverer way of **approximating the posterior**. It is **usually faster than MCMC**, but it is **less general**.

NOTE If you want to perform bayesian regression **without having a conjugate prior**, you **cannot compute the posterior distribution in a closed form** so you would need **the methods cited above** (Gaussian approximation, MCMC, Variational approximation, ...). And they have a **computational cost** that is **quite high, struggling in scaling for very large problems**.

Summarizing, **the pros and cons of working with fixed basis functions**, as we do in linear regression models are:

- **Advantages:**

- The problem is convex, so there are only global optima and not local optima in which we could get stuck. (***)
- Closed-form solution (***)
- Tractable Bayesian treatment, if we have analytical solution for the posterior, i.e. having a conjugate prior.
- Choosing the proper basis functions we can model arbitrary non-linear relationship between the input variables and the target variables.

- **Limitations:**

- The **feature selection may be hard**, since we could not know which one should be selected, ending up with a huge number of features. The features should then be selected.
- **Basis functions are chosen independently from the training set.**
- **Curse of dimensionality:** the more the features I consider the larger is the number of samples (observed data) that I need to estimate the values of the parameters (w) associated to the features.

In our discussion of maximum likelihood for setting the parameters of a linear regression model, we have seen that the effective model complexity, governed by the number of basis functions, needs to be controlled according to the size of the data set. Adding a regularization term to the log likelihood function means the effective model complexity can then be controlled by the value of the regularization coefficient, although the choice of the number and form of the basis functions is of course still important in determining the overall behaviour of the model. This leaves the issue of deciding the appropriate model complexity for the particular problem, which cannot be decided simply by maximizing the likelihood function, because this always leads to excessively complex models and over-fitting. We therefore turn to a Bayesian treatment of linear regression, which will avoid the over-fitting problem of maximum likelihood, and which will also lead to automatic methods of determining model complexity using the training data alone.

NOTE Bayes' theorem is obviously the heart of this type of regression. As a reminder, Bayes theorem states

$$\underbrace{P(A|B)}_{Posterior} = \frac{\overbrace{P(B|A)}^{Likelihood} \overbrace{P(A)}^{Prior}}{\underbrace{P(B)}_{Marginalization}} \quad (24)$$

Example We can estimate the probability of getting head or tail with a coin flip. We don't know if the coin is tricked or not. We know that a coin flip follow a Bernoulli distribution

$$P(r) = \begin{cases} p, & \text{if } r = \text{Head} \\ q = 1 - p, & \text{if } r = \text{Tail} \end{cases}$$

- **Prior** $P(r)$ we assume a regular coin so $P(r) = p = \frac{1}{2}$ ($P(\text{Head}) = \frac{1}{2}$ and $P(\text{Tail}) = \frac{1}{2}$)
- **Posterior** $P(r|D)$ Probability of the coin having $p = \frac{1}{2}$ given the Data.
- **Likelihood** $P(D|r)$ Probability of observing the Data given that the coin have $P = \frac{1}{2}$
- **Marginalization** $P(D)$ Probability of observing the Data

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

So the Bayesian approach formulate our knowledge as follow,

1. We formulate our knowledge about the world in a probabilistic way
 - (a) We define the model that expresses our knowledge qualitatively
 - (b) We capture our assumptions about unknown parameters by specifying the prior distribution over those parameters before seeing the data
2. We observe the data
3. We compute the posterior probability distribution for the parameters, given observed data
4. We use the posterior distribution to make predictions or take decisions

As the example we have made before we have

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

- **Prior** $P(w)$ probability distribution of the parameters
- **Posterior** $P(w|D)$ Probability of w given training data D
- **Likelihood** $P(D|w)$ Probability of observing the Data given parameters w
- **Marginalization** $P(D)$ Probability of observing the Data $P(D) = \int P(D|w)P(w)dw$

Our objective is to find the most probable value of w given the data maximum a posteriori (MAP), which is the mode of the posterior $P(w|D)$.

Note An advantage of the Bayesian approach is the introduction of the prior distribution. This greatly reduce our hypothesis space(parameter space) reducing overfitting. Assuming a Gaussian likelihood model we can take a Gaussian prior. The Gaussian family is conjugate to itself (or self-conjugate) with respect to a Gaussian likelihood function: if the likelihood function is Gaussian, choosing a Gaussian prior over the mean will ensure that the posterior distribution is also Gaussian.

$$P(w) \sim \mathcal{N}(w|w_0, S_0) \quad (25)$$

- w_0 [MX1] Mean of the distribution. We guess that w is equal to w_0 in the parameter space.
- S_0 [MxM] Covariance matrix of the distribution. The matrix is diagonal because we assume i.i.d. parameters.

So $P(w)$ is a multivariate⁶Gaussian. As we have said before, the posterior will be Gaussian

$$\begin{aligned} P(w|t, \Phi, \sigma^2) &\propto \mathcal{N}(w|w_0, S_0)\mathcal{N}(t|\Phi w, \sigma^2 I_N) = \mathcal{N}(w_N, S_N) \\ w_N &= S_N \left(S_0^{-1} w_0 + \frac{\Phi^T t}{\sigma^2} \right) \\ S_N^{-1} &= S_0^{-1} + \frac{\Phi^T \Phi}{\sigma^2} \end{aligned}$$

⁶In probability theory and statistics, the multivariate normal distribution or multivariate Gaussian distribution is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions.

For two particular prior distribution cases, Bayesian regression estimate reduces to already known regressions,

- $S_0 = \infty I$ In this case we have no prior knowledge of the parameters distribution. If we substitute S_0 in w_N definition we find,

$$\begin{aligned}
S_N^{-1} &= S_0^{-1} + \frac{\Phi^T \Phi}{\sigma^2} = \frac{\Phi^T \Phi}{\sigma^2} \\
S_N &= \sigma^2 (\Phi^T \Phi)^{-1} \\
w_N &= S_N \left(S_0^{-1} w_0 + \frac{\Phi^T t}{\sigma^2} \right) \\
&= \sigma^2 (\Phi^T \Phi)^{-1} \frac{\Phi^T t}{\sigma^2} \\
&= (\Phi^T \Phi)^{-1} \Phi^T t
\end{aligned} \tag{26}$$

As we can see (26) is equal to the ML estimator. So Bayesian regression reduces to the maximum likelihood case.

- $w_0 = 0, S_0 = \tau^2 I, \quad \tau \in \mathbb{R}$

$$\begin{aligned}
S_N^{-1} &= S_0^{-1} + \frac{\Phi^T \Phi}{\sigma^2} \\
&= \frac{1}{\tau^2} I + \frac{\Phi^T \Phi}{\sigma^2} \\
w_N &= S_N \left(S_0^{-1} w_0 + \frac{\Phi^T t}{\sigma^2} \right) \\
&= S_N \frac{\Phi^T t}{\sigma^2} \\
&= \left(\frac{1}{\tau^2} I + \frac{\Phi^T \Phi}{\sigma^2} \right)^{-1} \frac{\Phi^T t}{\sigma^2} \\
&= \left(\frac{\sigma^2}{\tau^2} I + \Phi^T \Phi \right)^{-1} \Phi^T t
\end{aligned} \tag{27}$$

We can notice that (27) is equal to Ridge regression with $\lambda = \frac{\sigma^2}{\tau^2}$. Small values of S_0 corresponds to high values of λ and viceversa.

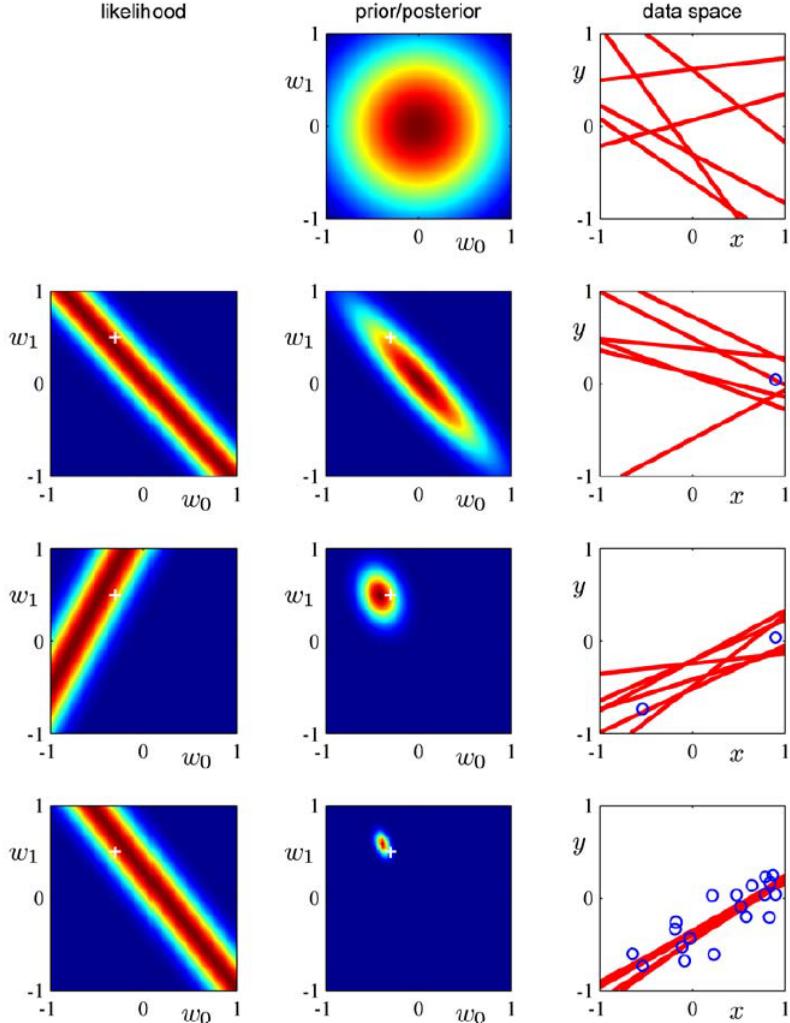
Example - Bayesian sequential learning We generate some data from

$$t(x) = -0.3 + 0.5x + \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, 0.04)$$

As a model we take

$$y(x, w) = w_0 + w_1 x, \quad \sigma^2 = 0.04, \quad \tau^2 = 0.5$$

To find the posterior distribution we use an iterative approach as follow. We start from a multivariate Gaussian prior $P(w) \sim \mathcal{N}(0, \tau^2 I)$ (prior). Then we take one data point and we find the parameters that make the model pass through that point, also considering data noise σ^2 (likelihood). The next step is to multiply the prior with the likelihood to find a posterior distribution in parameters space. The new posterior can be used as a prior for the next iteration. We take a new point, we find the parameters that make the model pass through that point and again we multiply together prior and likelihood. Note that at each iteration, the likelihood distribution considers only one point at the time.



3.5.1 Predictive distribution

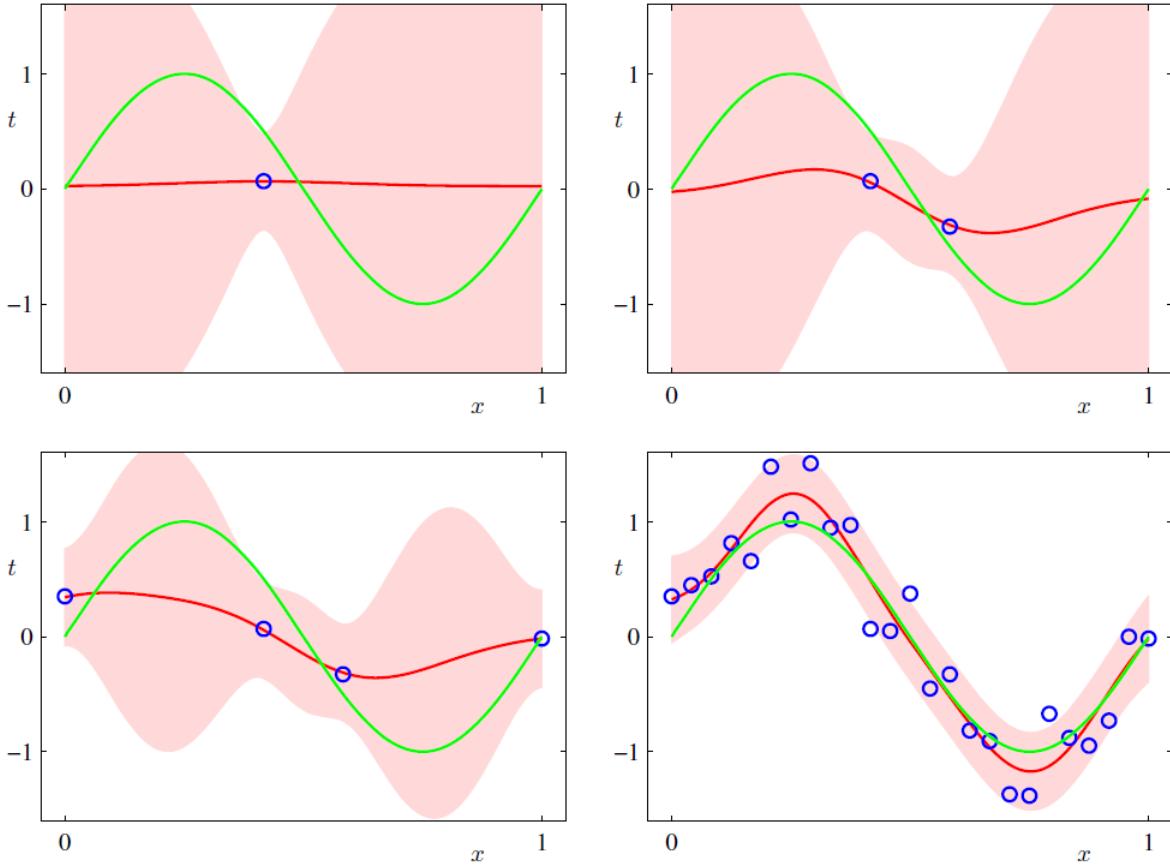
In practice, we are not usually interested in the value of w itself but rather in making predictions of t for new values of x . This requires that we evaluate the posterior predictive distribution defined by,

$$p(t|x, D, \sigma^2) = \int \mathcal{N}(t|w^T \Phi(x), \sigma^2) \mathcal{N}(w|w_N, S_N) dw = \mathcal{N}(t|w_N^T \Phi(x), \sigma_N^2(x)), \quad (28)$$

$$\sigma_N^2(x) = \underbrace{\sigma^2}_{\text{noise in the target values}} + \underbrace{\Phi(x)^T S_N \Phi(x)}_{\text{Uncertainty associated with parameters values}} \quad (29)$$

σ^2 is also called irreducible noise, in fact for $N \rightarrow \infty$, the second term of $\sigma_N^2(x)$ goes to zero, but σ^2 remain constant. In the figure below we can observe,

- **Green line** True model
- **Blue dots** Samples
- **Red line** Mean of the Gaussian predictive distribution
- **Red area** Predictive distribution spanning one standard deviation either side of the mean.



Theorem[section]

4 Linear models for classification

The goal in classification is to take an input vector x and to assign it to one of K discrete classes C_k where $k = 1, \dots, K$. In the most common scenario, the **classes are taken to be disjoint**, so that each input is assigned to one and only one class.

Example The input vector x is the set of pixel intensities, and the output the variable t will represent the presence of cancer (class C_1) or absence of cancer (class C_2).

So it means that the **input space can be divided into K disjointed regions each one associated to a specific class**. The input space is thereby divided into decision regions whose boundaries are called **decision boundaries** or decision surfaces. We will consider **linear models for classification**, by which we mean that the **decision surfaces** are defined by **($D - 1$)-dimensional hyperplanes within the D -dimensional input space**. Datasets whose classes can be separated exactly by linear decision surfaces are said to be **linearly separable**.

4.1 Linear classification

We will consider linear models for classification. In the linear regression case, the model is linear in parameters:

$$y(x, w) = \sum_{j=0}^{D-1} w_j x_j = x^T w$$

To have a simpler notation in future steps we explicit w_0 from w

$$= w_0 + \sum_{j=1}^{D-1} w_j x_j = w_0 + x^T w \quad (30)$$

For classification, we need to **predict discrete class labels, or posterior probabilities that lie in the range of $(0, 1)$** , so we **use a non-linear function** (discriminant function) to remap the (unbounded) input space to the (bounded) output space.

$$y(x, w) = \underbrace{f(w_0 + x^T w)}_{\text{activation function}}$$

Indeed using the linear regression model the output is unbounded since the input is unbounded and the linear model does not saturate, hence to obtain the bounding of the output in classification we must use a **non-linear function that takes the output of the linear regression model and remap them in the correct range** (e.g. $(0,1)$ in case of binary classification).

NOTE The remapping function cannot be linear since linear function like lines and are unbounded, so a bounded function, for example between $(0,1)$, must be non-linear (e.g. Heaviside function i.e. step function).

A naive way to perform classification with two output classes is to choose an arbitrary **activation function** f and for output less than 0.5 we have class 0 and vice-versa class 1 (i.e. we are setting a threshold). We **could be tricked into thinking that this classifier can predict non-linear boundaries, but it's not** the case. In fact, considering f invertible, taken the boundary

$$\begin{aligned} y(x, w) &= f(w_0 + x^T w) = 0.5 \\ f^{-1}(f(w_0 + x^T w)) &= f^{-1}(0.5) \\ w_0 + x^T w &= f^{-1}(0.5) \\ w_0^* + x^T w &= 0, \\ \text{with } w_0^* &= w_0 - f^{-1}(0.5) \end{aligned} \tag{31}$$

As we can see (31) represents an hyperplane, hence the **decision surfaces are linear functions of x , even if the activation function is non-linear**. So this models are called **generalized linear models, even if they are no more linear in the parameters w** , because what is linear are the decision surfaces, hence using this type of models we are partitioning the input space using the hyper-planes. Dealing with **this kind of models is more complex both analytically and computationally with respect to regression**. As in regression we can use fixed non-linear basis function to make the input space linearly separable (i.e. we are not obliged to use input variables but also features without changing the characteristics of the problem). The bad thing about this approach is that the **model is no longer linear in the parameters, so no closed form solution exists** (***)).

NOTE The decision surfaces correspond to $y(x, w) = \text{const}$, where the constant is, for binary classification, the threshold we decide to consider.

NOTE The function f is any (non-linear) function that gets a real value and maps it, for binary models, in a $(0, 1)$ range. Some examples are: hyperbolic tangent (\tanh), sigmoid, ...

NOTATION In **two-class problem**, we have binary (real) target value $t \in \{0, 1\}$, such that $t = 1$ is the positive class and $t = 0$ is the negative class. We can **interpret the value of t as the probability of the positive class**, so the output of the model can be represented as the probability that the model assigns to the positive class. Instead if there are K **classes**, we can use a **1-of-K encoding scheme (one-hot encoding)**. Instead of having a scalar value for the target class we produce a vector t of length K and contains a single **1 for the correct class and 0 elsewhere** (e.g. if $K=5$ then $C_2 \rightarrow t = (0, 1, 0, 0, 0)^T$), and also in this case we can interpret t as a vector of class probability.

NOTE The target of observed data is totally sure of the answer indeed they can only have as values 0 or 1.

NOTE In the case of K classes, if the model is not constrained, through f , to output a probability vector that sum up to 1 it won't be able to do it, but then you can normalize the output obtaining that characteristic.

As we have discussed for linear regression, also for linear classification we can have the three categories of approaches:

- **Discriminant function:** build a function that directly maps each input to a specific class. Hence it simply tries to place a separating boundary (that in this case is linear) in a way that minimizes the loss function, i.e. the miss-classified instances of the dataset. Hence this is a **direct method**.
- **Probabilistic approach:** model the conditional probability distribution $p(C_k|x)\forall k$ (i.e. the distribution of an input to belong to a certain class for all the classes) and use it to make optimal decisions. There are in this case two alternatives:
 - **Probabilistic discriminative approach:** model $p(C_k|x)$ directly, for instance using parametric models (e.g. logistic regression).
 - **Probabilistic generative approach:** model class conditional densities $p(x|C_k)$ together with prior probabilities $p(C_k)$ for the classes. Infer the posterior using Bayes' rule:

$$P(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)}$$

So we are learning not only the distribution of probability of the class given the input $p(C_k|x)$ but also the probability of seeing a certain input for a certain class $p(x|C_k)$ and the probability of seeing a certain class $p(C_k)$ and so the joint distribution $p(x,C_k) = p(x|C_k)p(C_k)$ of classes and inputs. Instead in the discriminative approach we directly compute the probabilistic distribution $p(C_k|x)$. Since you have learnt the distribution over the input, this type of models can also be used to generate new data. [we won't see algorithms of this class].

4.1.1 Geometric interpretation

Let's see what happens if we decide to use a linear model, the one used for linear regression. The same thing in binary classification would be:

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

and then assigning a point x to C_1 is $y(x) \geq 0$ and class C_2 otherwise ($y(x) < 0$), choosing the threshold equal to 0. In this settings the decision boundary would be when the prediction

of the model is 0, i.e. the decision boundary correspond to $y(x) = 0$. One interesting thing is that by taking two point of the decision surface:

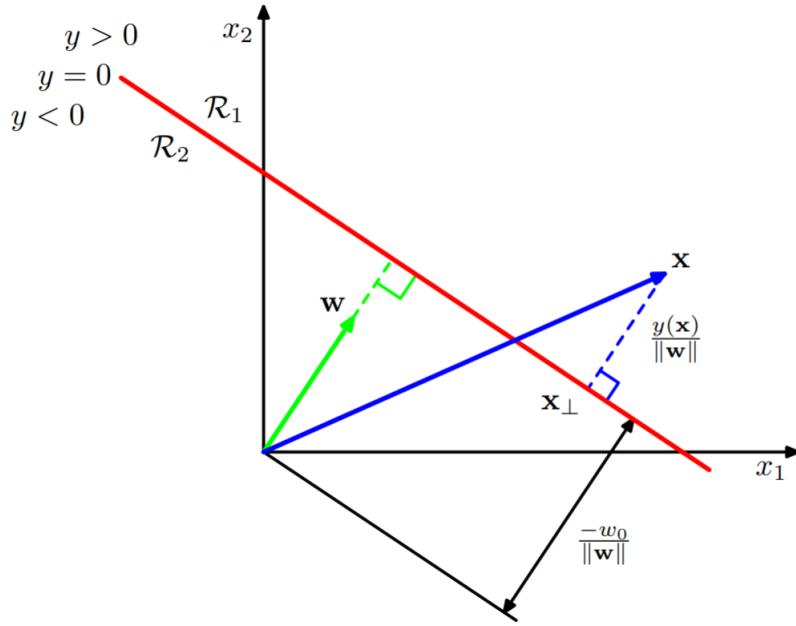
$$y(x_a) = y(x_b) = 0$$

I get immediately the equation of the hyperplane that represent the decision boundary (indeed we know that a plane is identified by two points):

$$w^T x_a + w_0 = w^T x_b + w_0$$

$$\implies w^T(x_a - x_b) = 0$$

From the last equation we could notice that w is orthogonal to the vector $x_a - x_b$ that lays on the plane itself, so the **parameters w^T learned by the model represent a vector that is orthogonal to the decision surface**:



Instead, **the role of w_0 is to specify where the plane is located, the shift with respect to the origin in the direction orthogonal to the plane**. Indeed calling x the decision surface:

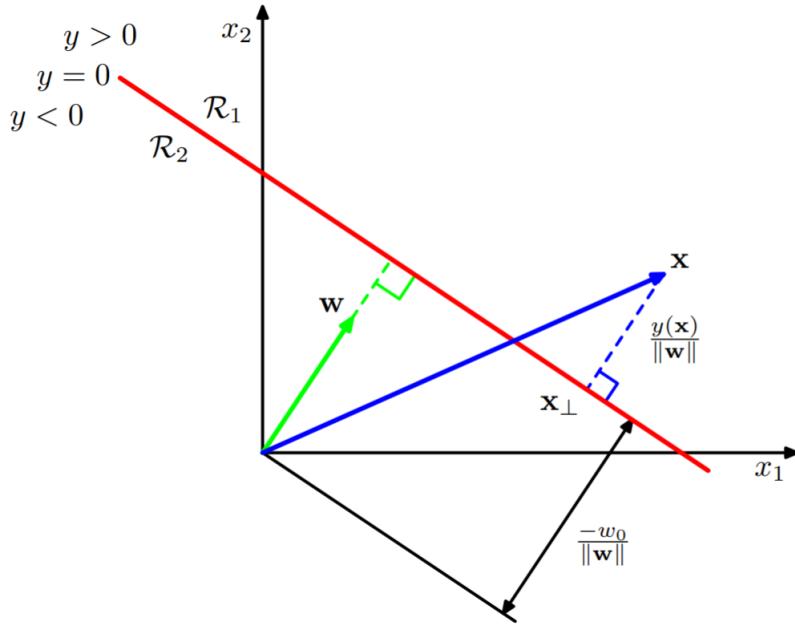
$$\frac{w^T x}{\|w\|_2} = -\frac{w_0}{\|w\|_2}$$

Hence **in a linear model** we know that the **parameters w represent the orientation of the plane**, and the **parameter w_0 determines the location of the decision surface**.

To have a better understanding of the discriminant function we can analyze it from a geometric point of view. The simplest representation of a linear discriminant function is obtained by taking a linear function of the input vector so that $y(x, w) = w_0 + x^T w$. w_0 is called bias

and its negative is called threshold. An input vector x is assigned to class C_1 if $y(x) \geq 0$ and to class C_2 otherwise. The corresponding decision boundary is therefore defined by the relation $y(x) = 0$, which corresponds to a $(D - 1)$ -dimensional hyperplane within the D -dimensional input space. Consider two points x_A and x_B both of which lie on the decision surface. Because $y(x_A) = y(x_B) = 0$, we have $w^T(x_A - x_B) = 0$ and hence the vector w is orthogonal to every vector lying within the decision surface⁷, and so w determines the orientation of the decision surface. We can also say that w_0 regulates the normal distance (d) of the boundary from the origin. To find d we can project⁸ a point x on the boundary on w

$$\begin{aligned} w^T x + w_0 &= 0 \\ w^T x &= -w_0 \\ d &= \frac{w^T x}{\|w\|_2} = -\frac{w_0}{\|w\|_2} \end{aligned} \tag{32}$$



Furthermore, we can obtain the signed distance (r) of a point x from the boundary. Let's consider the projection x_\perp of x on the boundary. Then

$$\begin{aligned} x &= x_\perp + r \frac{w}{\|w\|_2} \\ w^T x &= w^T x_\perp + w^T r \frac{w}{\|w\|_2} \end{aligned}$$

⁷Given two vectors their dot product is 0 when they are perpendicular to each other. $a \cdot b = |a||b|\cos\theta$

⁸We know that the projection of a vector a on another vector b is $\text{proj}_b(a) = \frac{ab}{\|b\|}$.

$$\begin{aligned}
w^T x + w_0 &= \underbrace{w^T x_\perp + w_0}_{=0} + w^T r \frac{w}{\|w\|_2} \\
y(x) &= w^T r \frac{w}{\|w\|_2} \\
y(x) &= r \frac{\|w\|_2^2}{\|w\|_2} \\
r &= \frac{y(x)}{\|w\|_2}
\end{aligned} \tag{33}$$

Hence, the prediction of the model $y(x)$ is proportional to the distance of the input point to the boundary.

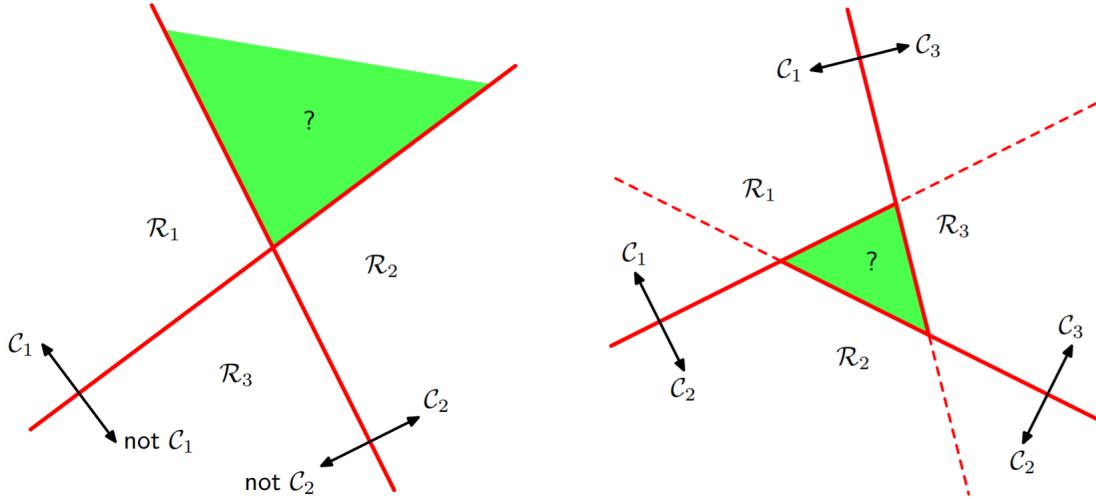
4.1.2 Multiple outputs

Now consider the extension of linear discriminants to $K > 2$ classes. We might be tempted to build a **K-class discriminant** by combining a number of two-class discriminant functions. However, this leads to some serious difficulties.

One-versus-the-rest Consider the use of $K-1$ classifiers each of which solves a two-class problem of separating points in a particular class C_k from points not in that class. The problem of this approach is that there may be region of conflicts. There are regions in input space that are ambiguously classified, as we can see in the left-hand diagram: in the green area the two classifier says both that the element belongs to their class, hence the right class to be used is undetermined.

NOTE $K-1$ classifiers (and not K) since if it does not belong to one of the $K - 1$ tested classes it would surely belong to the last class. Adding the K -th classifier would make the problem even worse since new regions of conflict would be added.

One-versus-one An alternative is to introduce $K(K - 1)/2$ binary discriminant functions, one for every possible pair of classes having a number of classifier quadratic in the number of classes. Each point is then classified according to a majority vote amongst the discriminant functions. However, this too runs into the problem of ambiguous regions with a conflict of class attribution, as illustrated in the right-hand diagram.



Hence **both approaches may create region of conflicts**, since the set of classifier used create decision areas that are not disjointed, i.e. there are **some classifiers that do not agree in the classification, so we must decide what to do in this cases.**

Linear discriminant functions We can solve these difficulties (the simplest solution) by considering **a single K-class discriminant comprising K linear discriminant functions**, each one with one of the K positive classes, of the form:

$$y_k(x) = x^T w_k + w_{k0}, \quad \text{where } k = 1, \dots, K \quad (34)$$

and then **assigning a point x to class C_k if $y_k(x) > y_j(x)$, $\forall j \neq k$, i.e. we assign the class with the highest predicted value.** The **decision boundary** between class C_k and class C_j is therefore given by $y_k(x) = y_j(x)$ and hence corresponds to a **(D - 1)-dimensional hyperplane**. The resulting decision boundaries are **singly connected**⁹ and **convex**. For any two points x_a and x_b inside that lie inside the region R_k :

$$y_k(x_a) > y_j(x_a) \quad y_k(x_b) > y_j(x_b)$$

implies that for positive α :

$$y_k(\alpha x_a + (1 - \alpha)x_b) > y_j(\alpha x_a + (1 - \alpha)x_b)$$

due to linearity of the discriminant functions. Convexity imposes that taken two points x_A and x_B that belong to the same region R_k , every point \hat{x} in between is still inside region R_k

$$\hat{x} = \lambda x_A + (1 - \lambda)x_B, \quad \hat{x} \in R_k. \quad \lambda \in [0, 1]$$

Convexity and linearity of the discriminant functions imply that

$$f_k(\lambda x_A + (1 - \lambda)x_B) > f_j(\lambda x_A + (1 - \lambda)x_B), \quad \forall j \in [1, K] \setminus k$$

⁹It means that all the linear functions are ray originating from a common point

NOTE Singly connected means that all the linear functions are ray originating from a common point, instead convexity means that given two points in a region R_k , all the points of the line that connect the points are contained in the same region R_k .

4.2 Least square for classification

Consider a general classification problem with K classes, with a one-hot (1-of-K) encoding for the target vector t . **One justification for using least squares in such a context is that it approximates the conditional expectation $E[t|x]$ of the target values given the input vector.** Each class is described by its own **linear model**

$$y_k(x) = w_k^T x + w_{k0}, \quad \text{where } k = 1, \dots, K \quad (35)$$

Using vector notation,

$$\begin{aligned} y(x) &= \tilde{W}^T \tilde{x}, \quad \text{where} \\ \tilde{W} &= \left[\begin{bmatrix} w_{10} \\ \vdots \\ w_{1D} \end{bmatrix}_{w_1^T} \cdots \begin{bmatrix} w_{K0} \\ \vdots \\ w_{KD} \end{bmatrix}_{w_K^T} \right], \quad [(D+1)\times K] \\ \tilde{x} &= (1, x^T)^T, \quad [D+1\times 1] \end{aligned}$$

where the k -th column of \tilde{W} is $\tilde{w}_k = (w_{k0}, w_k^T)^T$, and \tilde{W} has \tilde{w}_k as row, i.e. each row represent one of the K classifiers.

We classify the input x into class C_k if $y_k(x) > y_j(x), \forall j \in [1, K] \setminus k$ (i.e. $\neq k$). $y_k(x)$ corresponds to the k^{th} element of $y(x)$. **To estimate the parameter we can follow what we did for the regression problem.** To minimize least square,

$$\begin{aligned} \tilde{W} &= (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T T, \quad [(D+1)\times K] \quad (36) \\ \tilde{X} & [N \times (D+1)] \implies \tilde{X}^T \quad [(D+1) \times N] \\ \tilde{T} & [N \times K] \end{aligned}$$

With \hat{X} the input matrix (that correspond to Φ in the linear regression case) whose i -th row is \tilde{x}_i^T and \hat{T} the target matrix (that correspond to the vector t in the linear regression case, since it was a single output) whose i -th row is t_i^T . Hence the **input is assigned to the class for which $t_k = \tilde{x}^T \tilde{w}_k$ is the largest**.

But there are problems in using least squares for classification.

4.2.1 Least squares problems

The least square approach is problematic in some cases.

Outliers Assuming we have only two classes, **crosses** (negative) and **circles** (positive), we want to **separate them using a generalized linear model** able to separate the two classes. In some cases, both linear regression and logistic regression (regression with sigmoid activation function, i.e. returns probabilities) are able to classify the training set (image on the left). But **least squares** is very sensitive to outliers. Least square tries to find a line which is the most close to all points, i.e. is trying to solve a linear regression problem, not a classification one. Indeed it evaluates the square distance between the samples and the line. It means that an **outlier** will have a greater impact on the line position because it will be more distant with respect to the probable samples. In classification this could degrade a lot the performance, because the **boundary** could deviate so much that some samples, previously well classified, now lie on the other side of the boundary. (image on the right)

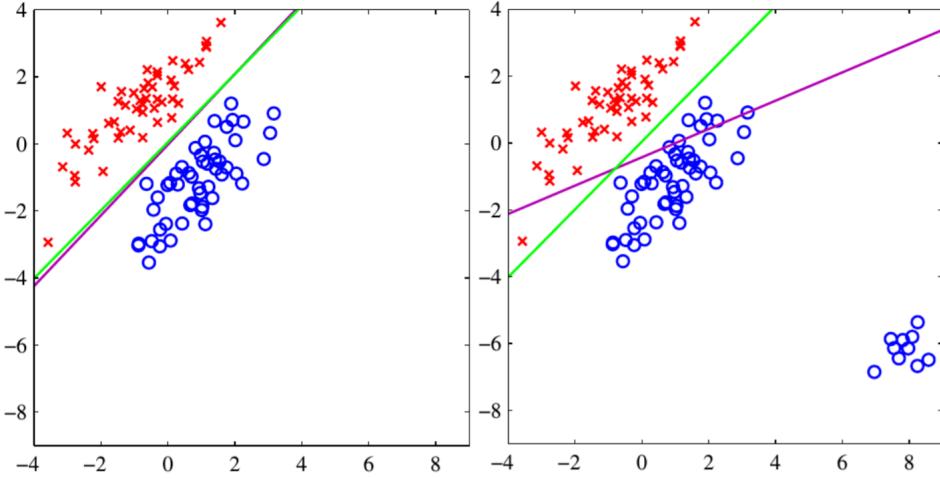


Figure 12: Input-Output space (x, t). Method: **least squares**; **logistic regression**

Instead **logistic regression is not affected by the outliers**, and this is how it should be, since **from the classification point of view the outliers were already well classified so it makes no sense to change the boundary given those points**.

Non-Gaussian distributions We recall that least squares corresponds to the maximum likelihood under the assumption of a Gaussian conditional distribution (i.e. **Gaussian noise**), whereas binary target vectors clearly have a distribution that is far from **Gaussian**. As we can see in the figure below, even though the three classes are linearly separable, **least square is not able to find good boundaries**. So the reason of failure is due to the assumption of a Gaussian conditional distribution that is not satisfied by binary target vectors.

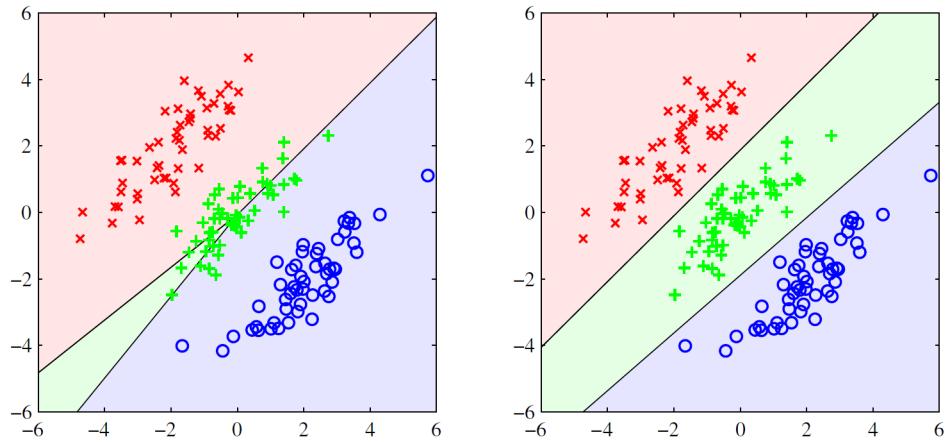


Figure 13: Left: least square. Right: logistic regression.

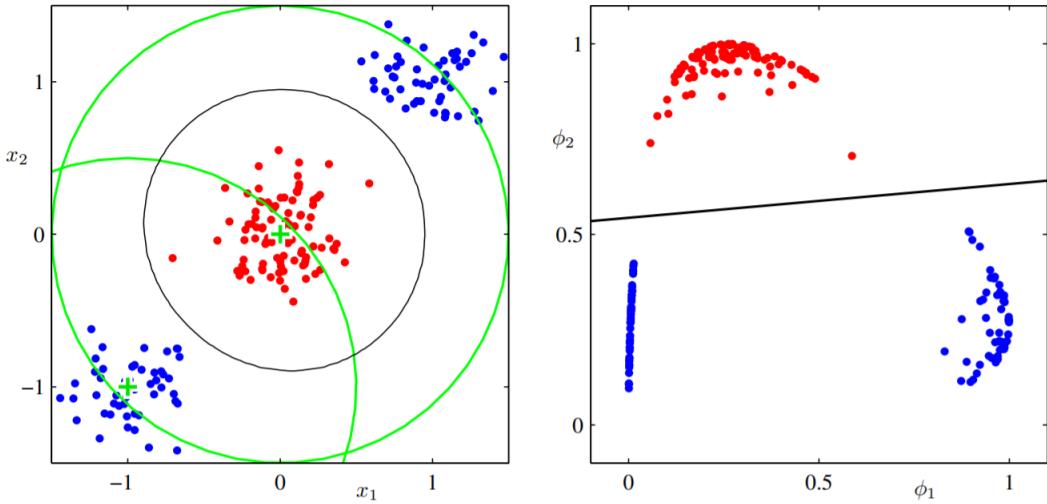
Instead **logistic regression is not affected by the fact that binary target vectors are distributed according to a non-Gaussian distribution.**

4.2.2 Fixed Basis functions

So far, we have considered classification models that work directly in the input space. All considered algorithms are equally applicable if we first make a **fixed non-linear transformation of the input space using vector of basis functions $\Phi(x)$** . Decision boundaries will be linear in the feature space, but would correspond to non-linear boundaries in the original input space. Classes that are linearly separable in the feature space do not need to be linearly separable in the original input space. This means that if we are able to project the data in a new feature space, maybe with many more dimension, the hope is to find a space of features where the points can be classified using a linear model, i.e. an hyper-plane.

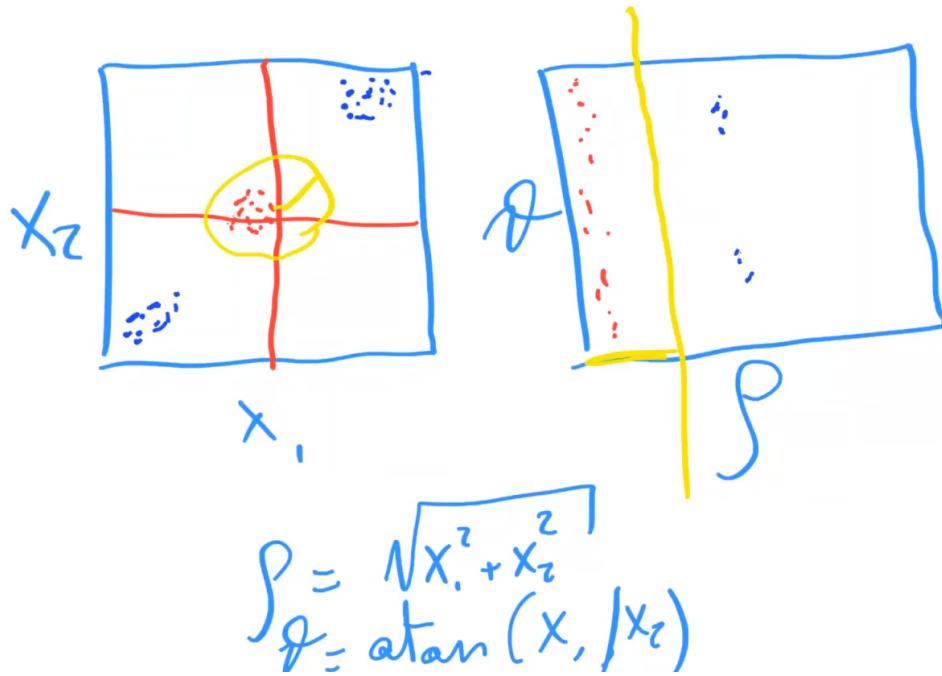
NOTE To have a **linear parametric model** we are not bounded to use linear boundaries in the input space but only **linear boundaries in the features space**.

Example - Non-linear basis function This is an illustration of the role of non-linear basis functions in linear classification models. The left plot shows the original input space (x_1, x_2) together with data points from two classes labelled red and blue, and is **easy to notice how is not possible to use a linear boundary** (i.e. an hyper-plane) to separate the two classes. Two ‘Gaussian’ basis functions $\Phi_1(x)$ and $\Phi_2(x)$ are defined in this space with centres shown by the green crosses and with contours shown by the green circles, i.e. they transform x_1 and x_2 into polar coordinates ρ and θ . The right-hand plot shows the corresponding feature space $(\Phi_1(x), \Phi_2(x))$ together with the linear decision boundary (obtained using logistic regression). This corresponds to a non-linear decision boundary in the original input space, shown by the black curve in the left-hand plot.



Hence the **a linear decision boundary in the features space $(\Phi_1(x), \Phi_2(x))$, and non-linear in the input space (x_1, x_2)** , is able to separate the two classes. Indeed the black

circle in the left image represent the black line in the right image. Another way to obtain the same result, i.e. making the problem linearly separable, is using the following features:



where $\Phi_1 = \rho = \sqrt{x_1^2 + x_2^2}$ and $\Phi_2 = \theta = \text{atan}(x_1/x_2)$, that being non-linear imply a non-linear boundary in the input-output space.

Remember From input variables we **use non-linear basis function to obtain non-linear models in the input that are still linear in the parameters**. The basis function, called **features**, apply any transformation to the input vector, even the identity one ($\Phi(x) = x$) or discard some of the input variables not using them in the basis function.

NOTE If the point belonging to different classes are not overlapping, always exist a set of features that allow a linear model to classify them, i.e. that transform the data to be linearly separable in the feature space. The **problem is to find this feature space**. And of course the more the features you use the more degree of freedom the boundaries you can represent have in the input space, but usually this means **overfitting exactly as in linear regression**.

REMEMBER There is NOT a general way to find the correct features to use in a certain problem but there are tools that can guide your choice. This is due to the fact that machine learning problem are ill-posed (miss-specified) problem: we are asked to optimize something but we are measuring something else.

NOTE Situation where the labels of the data are affected by noise, i.e. is possible to have an input that belongs to more classes, are dealt with probabilistic approaches and not with **direct approaches, because the latter simply search for the boundary that separates the classes.**

4.3 The Perceptron algorithm

The perceptron is **another example of linear discriminant models**. It is an algorithm for **online**¹⁰**supervised learning of binary classifiers**.

The algorithm **tries to find a threshold function**: a function that maps its input x (a real-valued vector) to an output value (target)

$$y(x) = f(w^T \Phi(x)), \quad \text{where}$$

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

Target values are $+1$ for C_1 and -1 for C_2 . The algorithm **finds the separating hyperplane by minimizing the loss function** i.e. the **sum distance of miss-classified points to the decision boundary**.

NOTE Using only the number of miss-classified points as loss function is not effective since it is a piece-wise constant function, i.e. not differentiable in some points and the gradient is almost 0 everywhere so it is not a good function to minimize. Instead, using the distance of the misclassified point, where the number is implicitly in the summation, we obtain a continuous function in the parameter of which we can compute the gradient.

Our **objective** is to find a parameter vector w such that $w^T \Phi(x_n) \geq 0$ when $x_n \in C_1$ and $w^T \Phi(x_n) < 0$ when $x_n \in C_2$. Now we define an error function as follow,

$$\epsilon_p(w, x_n) = \begin{cases} 0, & \text{If } x \text{ is classified correctly} \\ w^T \Phi(x_n) t_n, & \text{If } x \text{ is not classified correctly (proportional to boundary distance)} \end{cases}$$

where the multiplication for the target t_n let you have a negative (then turned to positive since the minimization of the loss require positive values) number, indeed the prediction of the model $w^T \Phi(x_n)$ is of sign opposite to the one of t_n due to the miss-classification. Now we define an error (loss) function for the parameter optimization,

$$L_P(w) = - \sum_{n \in M} w^T \Phi(x_n) t_n \tag{37}$$

¹⁰Online means that it is an iterative approach which calculate the solutions with multiple steps.

where M is the set of miss-classified points and the minus sign is needed to turn each term of the sum positive since all are negative for what we said earlier (this let us to have a positive loss that we can then minimize). To **perform minimization we use stochastic gradient descent (online)**

$$w^{(k+1)} = w^{(k)} - \alpha \nabla_w L_P(w) = w^{(k)} + \alpha \Phi(x_n)t_n \quad (38)$$

where α is the learning rate, i.e. an hyper-parameter that could be regulated and the gradient of the loss function, due to its linearity in the parameters, can be simply computed as $-\Phi(x_n)t_n$. Note that the stochastic gradient descent is an **online algorithm** so the latter is the update formula of the parameters to be used at each iteration, i.e. for each data point.

NOTE - Loss sign We have a minus sign in the loss function because $w^T\Phi(x_n)t_n$ will always be negative. This is due to the fact that if $w^T\Phi(x_n)$ is miss-classified, then it will have an opposite sign compared to t_n .

NOTE - Learning rate The learning rate α is a scaling factor for w since w is computed from a sum of terms that have in common α as a multiplicative term:

$$w^{(k+1)} = \sum_{i=0}^k w^{(i)} = \alpha \sum_{i=0}^k \Phi(x_{kn})t_{kn}$$

Hence for α we could use arbitrary values (even 1 to make it disappear) because the solution doesn't change if you scale w , indeed the value of w are orthogonal to the plane so they specify the orientation to the plain: multiplying w for a constant factor won't change the hyper-plane that defines the solution. This fact imply that there are infinite equivalent solutions for what concerns w . Being the solution insensitive of the scale of w , then α , could have any value, and the property of the decision boundary won't change. For this reason the learning rate α can be set to 1 for easiness.

Remember In linear regression the learning rate should respect the Robinson condition, so should decrease following certain condition using stochastic gradient descent. In the perceptron instead is not required due to its particular structure any value could be used.

4.3.1 Perceptron algorithm

Algorithm 1: Perceptron algorithm

Output : A parameter vector $w^{(k)}$ that correctly classifies the two classes

Input : Data set $x_n \in \mathbb{R}^D$

$t_n \in \{-1, +1\}, \forall n \in [1, N]$

Initialize: w_0 ($= 0$ is a common choice)

$k \leftarrow 0;$

while *!converged* (i.e. NO miss-classification over the dataset) **do**

$k \leftarrow k + 1;$

$n \leftarrow k \% N$ (i.e. $k \bmod N$);

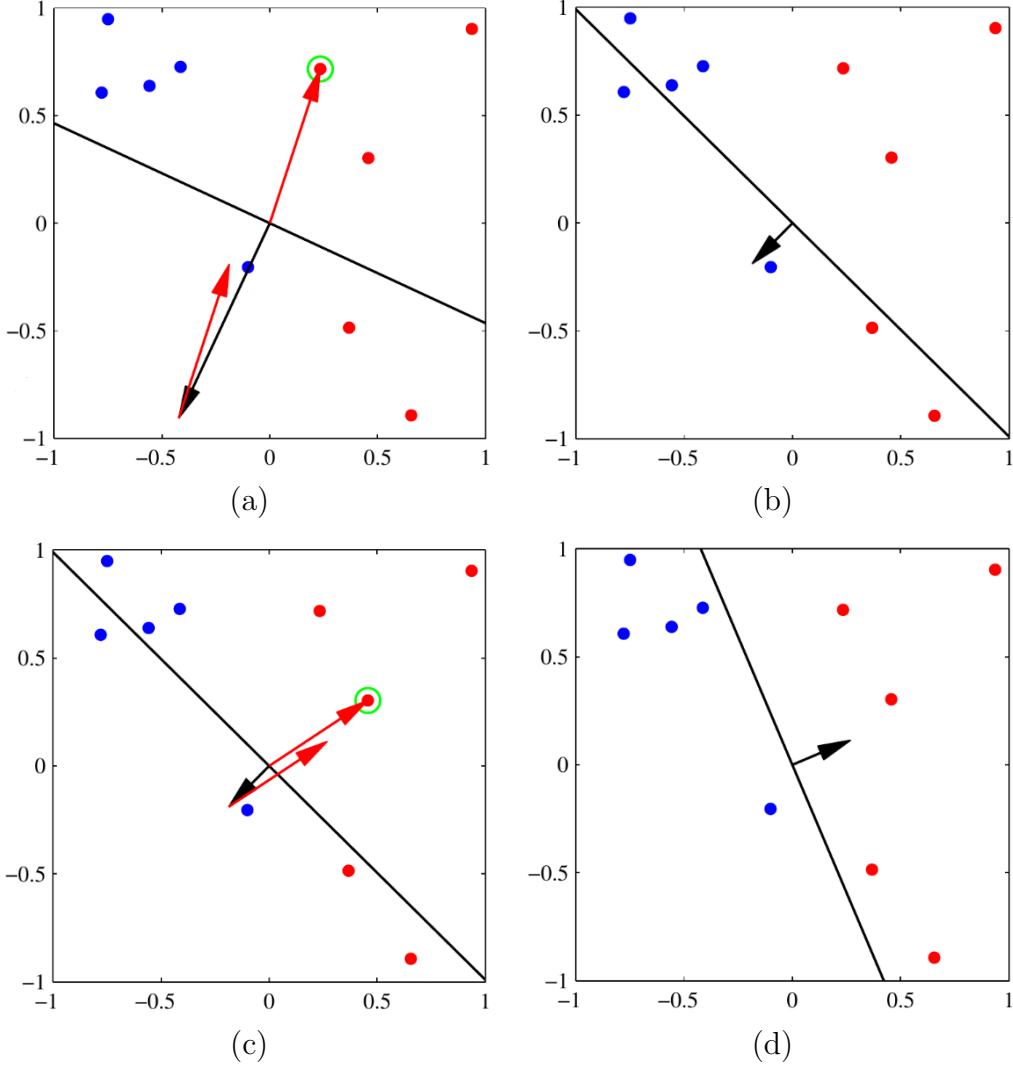
if $\hat{t}_n \neq t_n$ (i.e. miss-classification) **then**

$w^{(k+1)} \leftarrow w^{(k)} + \Phi(x_n)t_n;$

end

end

NOTE The perceptron algorithm iterate over the dataset until all the classification are correct ($n \leftarrow k \% N$ (i.e. $k \bmod N$)). For this reason the dataset can be scanned multiple times.



In the latter pictures the black arrow represent the weights $w^{(k)}$ that we have at a certain point and the red vector is the error $(\Phi(x_n)t_n)$ obtained from the classification of the green circled point. There are two red arrows since the one applied to where is pointing the black arrow ($w^{(k)}$) shows the point where the new black arrow ($w^{(k+1)}$) will point.

A fact about **algorithm steps** is that the **effect of a single update the contribution to the error of a miss-classification is reduced**,

$$L^{(k+1)} = -w^{(k+1)}\Phi(x_n)t_n = \underbrace{-w^{(k)}\Phi(x_n)t_n}_{>0} - \underbrace{(\Phi(x_n)t_n)^T\Phi(x_n)t_n}_{<0} < -w^{(k)T}\Phi(x_n)t_n$$

where $\alpha = 1$ and $\|\Phi(x_n)t_n\|^2 = (\Phi(x_n)t_n)^T\Phi(x_n)t_n > 0$. **So each time the error added to the loss is reduced.** Of course, this does not imply that the contribution to the error function from the other miss-classified patterns will have been reduced. **This means that**

we reduce the error of the miss-classified point we are considering, but we have no guarantee that the error of the other points gets better, meaning that the change in weight vector may have caused some previously correctly classified patterns to become miss-classified (indeed the boundary has been moved). Thus the perceptron learning rule is not guaranteed to reduce the total error function at each stage. What can occur are termination problems or slow the convergence.

Theorem 4.1 (Perceptron convergence theorem). *If the training data set is linearly separable in the feature space Φ , then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps*

Instead if the dataset is **not** linearly separable in the feature space chosen, as we could notice from the condition of the iterative cycle in the algorithm, it **won't stop** since not all points could be correctly classified in the feature space using a linear separating boundary, i.e. the points are not linearly separable. Unfortunately the **problem is semi-decidable** in the sense that by running, for example, 1 million of iterations and the solution is not found this does not mean that the solution does not exist, maybe has not being found. **Hence you know that the dataset is linearly separable if the perceptron finish, else if the perceptron does not stop we are not sure if the solution does not exist or the solution has still not being found.** For this reason the number of steps before convergence may be substantial. We are **not able to distinguish between non-separable problems and slowly converging ones**.

Furthermore, the theorem only grant to find a solution if it exists, so if multiple solution exists, we may find each time different solution, not always specific one. If multiple solutions exist, the one found depends by the initialization of the parameters and the order of presentation of the data points.

NOTE The solution found by the perceptron algorithm is the exact one, indeed it is a direct method.

4.4 Probabilistic Discriminative Models: Logistic regression

The **objective of probabilistic discriminative approaches** is to learn the **conditional probability of a class given the value of the features observed in the dataset** $p(C_k|\Phi)$, i.e. which is the probability of having observed the example $\Phi(x)$ to belong to class C_k .

Logistic regression is a **statistical model** that in its basic form uses a **logistic function to model a binary variable**. So it is **capable of resolve two-class classification**. Logistic regression is a **probabilistic discriminative** model so we **model directly the posterior probability** $p(C_k|\Phi)$ (differently from probabilistic generative models). In detail

it uses a **logistic sigmoid function**¹¹ as activation function.

$$p(C_1|\Phi) = \sigma(w^T\Phi) = \frac{1}{1 + e^{-w^T\Phi}} \quad (39)$$

$$p(C_2|\Phi) = 1 - p(C_1|\Phi) \quad (40)$$

and the **bias term is omitted for clarity**. Hence the idea of logistic regression is to apply the logistic function σ to the linear combination of the features to find the conditional probability that a given set of features belongs to a given class. **After having computed all the probabilities the decision boundary can be established both in the input output space and in the feature space.**

A model is linear if it is linear in the parameters w . **In the logistic regression case the model is not linear in w due to the sigmoid function.** This brings us back to what we have said for the classification task where we spoke about a general method $f(w^T x + w_0)$. **But**, as we have seen, setting a threshold (that is a number like 0.5) to separate the two classes (C_1 if $y(x) \geq th$ and C_2 otherwise), what happens is that **the separating boundaries between the two classes are linear, i.e. they are hyper-planes.**

NOTE Logistic regression is an algorithm to do linear classification not linear regression; the name is **misleading**. It is named regression since what we are **trying to learn** is not directly the value of the class $y(x)$, but the probability that a certain example (or more generally features of the example) belongs to a certain class, and **a probability is a continuous value between 0 and 1**.

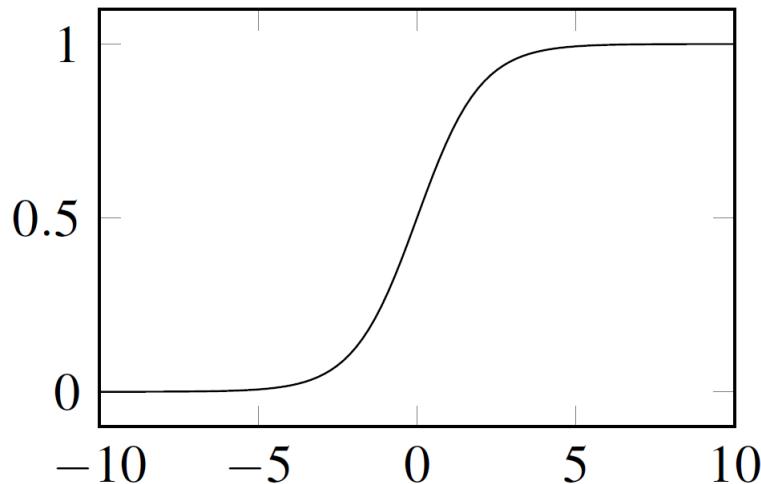


Figure 14: Sigmoid function

How can the parameters w be trained to minimize the loss function?

¹¹Sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

4.4.1 Maximum Likelihood for logistic regression

Logistic regression models use maximum likelihood to determine the parameters. So the idea is to find the weights of the logistic function that maximize the likelihood of observing the data that we have in our dataset.

Having a dataset $D = \{x_n, t_n\}, t \in \{0, 1\}$ (i.e. t can be either 0 or 1) $n = 1, \dots, N$ we call:

$$y_n = \sigma(w^T \Phi(x_n)) = \sigma(w^T \Phi_n) \equiv p(C_1 | w, \Phi(x_n))$$

so y_n is the probability of having C_1 as prediction of the model w over the sample x_n of the dataset. Turns out that the probability of the target vector given the input values X and the parameters w can be written as:

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

that is the probability of getting the right label. The formula is clear considering that $t_n = 1$ if x_n belongs to C_1 , so the term in the product would be y_n and $t_n = 0$ if x_n belongs to C_2 , so the term in the product would be $(1 - y_n)$ because we are looking for the probability of the samples to belong to the class C_1 , i.e. the positive class. **Hence $p(t|X, w)$ is the probability that our model has generated the dataset, i.e. the likelihood.** Is clear how **maximizing this expression we are looking for high terms of the product, which means: high y_n if $t_n = 1$ and low y_n (i.e. high $1 - y_n$) if $t_n = 0$.** That is expected since y_n is the probability of predicting class C_1 , so the probability of predicting C_1 when is the right class of the sample must be high, and the probability of predicting C_1 when is the wrong class of the sample must be low.

REMEMBER Data in the dataset are always i.i.d (independent and identically distributed samples) if not said differently. This assumption hold for what we have seen so far.

NOTE - Output For $t_n = 1$ we have C_1 and for $t_n = 0$ we have C_2 .

NOTE $y_n^{t_n}$ consider the probability given by the model when the answer is C_1 , and $(1 - y_n)^{1-t_n}$ consider the probability given by the model when the answer is C_2 i.e. not C_1 .

Note that from the latter definition of the likelihood is clear how the t_n follow a **Bernoulli distribution**. Indeed having only two classes the probability of having a class given the input data and the model is the one written above, that is also the expression of a Bernoulli distribution:

$$P(t_n) = y_n(\Phi(x_n)|w)^{t_n} (1 - y_n(\Phi(x_n)|w))^{1-t_n}$$

$$\begin{aligned}\implies P(t_n = 1) &= y_n^{t_n} = y_n \\ \implies P(t_n = 0) &= (1 - y_n)^{1-t_n} = 1 - y_n\end{aligned}$$

Since we want to maximize a product is always a good idea in this cases to use the **logarithm function to simplify calculus**, indeed being the logarithm a monotonic function the minimum and the maximum of the function remain the same. So, taking the log of the likelihood we define the log-likelihood and then to have a loss, i.e. a minimization problem, we take the negative. Hence we take the **negative log** defining the **cross-entropy error function** to be minimized:

Let's define a suitable loss function to use, reasoning as we have said. As seen t_n follows a Bernoulli distribution:

$$\begin{aligned}t_n &\sim Be(y_n(\Phi(x_n)|w)) \\ \implies P(t_n) &= y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}\end{aligned}\tag{41}$$

The equation (41) describes the probability of the result being t_n given the input $\Phi(x_n)$ and the parameters w . So we can take as loss function the product of all $P(t_n)$.

$$\begin{aligned}l(w) &= p(t|X, w) = \prod_{n=1}^N P(t_n) \\ &= \prod_{n=1}^N y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}\end{aligned}$$

↓ Transition to ln to simplify calculus.

Min & Max remain the same

$$\begin{aligned}l(w) &= \ln\left(\prod_{n=1}^N y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}\right) \\ &= \sum_{n=1}^N \ln(y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\ &= \sum_{n=1}^N \ln(y_n(\Phi(x_n)|w)^{t_n}) + \sum_{n=1}^N \ln(1 - y_n(\Phi(x_n)|w))^{1-t_n} \\ &= \sum_{n=1}^N t_n \ln(y_n(\Phi(x_n)|w)) + \sum_{n=1}^N (1 - t_n) \ln(1 - y_n(\Phi(x_n)|w)) \\ &= \sum_{n=1}^N t_n \ln(y_n(\Phi(x_n)|w)) + (1 - t_n) \ln(1 - y_n(\Phi(x_n)|w))\end{aligned}$$

↓ For simplicity we remove y_n parameters/arguments

$$= \sum_{n=1}^N t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n) =$$

$$= \sum_{n=1}^N L_n$$

To find the optimal parameters we would like to maximize $L(w)$. Usually loss function are minimized. To be coherent with the literature, we put a minus sign in front of our loss function $L(w)$.

$$L(w) = - \sum_{n=1}^N t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n) = \sum_{n=1}^N L_n \quad (\text{Binary cross-entropy}) \quad (42)$$

so the loss function is the summation over all the samples of the dataset of the negative log-likelihood for each sample.

NOTE Remember that y_n is a probability: is the output of the logistic model so is between 0 and 1 and can be interpreted as the conditional probability of that sample belonging to the class C_1 ($p(C_1|x_n)$).

ATTENTION Do not get confused with the perceptron, here there is not the concept of samples correctly classified. The concept used instead is the likelihood: the model outputs a probability of the sample to belong to a class, so this is different from saying that the sample belongs (100%) to a certain class. Here the boundary is not considered since we are talking about probability: a sample is correctly classified with some probability and not correctly classified with the probability 1 - the latter probability. The approach is continuous since all the samples have an amount of miss-classification, and this is the reason why we do not have the same issues we had with the perceptron.

Now we have to minimize the loss function. Unfortunately $L(w)$ is no longer linear which means that there is **no more a closed form solution**, so we have to use iterative methods. Nonetheless, we need to find the gradient of $L(w)$.

$$\begin{aligned} \nabla_w L(w) &= \sum_{n=1}^N \frac{\partial L_n(w)}{\partial y_n} \frac{\partial y_n}{\partial w} \quad \text{Chain rule} \\ &= \sum_{n=1}^N \underbrace{\frac{y_n - t_n}{y_n(1 - y_n)}}_{\frac{\partial L_n(w)}{\partial y_n}} \underbrace{y_n(1 - y_n)\Phi(x_n)}_{\frac{\partial y_n}{\partial w}} \\ &= \sum_{n=1}^N (y_n - t_n)\Phi(x_n) \end{aligned}$$

being $\frac{\partial L_n(w)}{\partial y_n} = \frac{y_n - t_n}{y_n(1 - y_n)}$ and $\frac{\partial y_n}{\partial w} = y_n(1 - y_n)\Phi(x_n)$.

NOTE The gradient we obtain is **really similar to the one we had in linear regression**: the gradient of the loss function for the sample x_n is equal to the **product of the difference between the prediction of the model y_n and the real label t_n (error), and the values of the features of x_n** . This is the **same expression of the error/gradient we have seen for both regression (i.e. of the sum of square error function)**.

NOTE The loss of the perceptron was $\Phi(x_n)t_n$ for the $n - th$ sample, indeed the error can be written as $w^T\Phi(x_n)t_n$ so the gradient would be $\Phi(x_n)t_n$. So it is a little bit different

There is **no closed form solution, due to non-linearity of the logistic sigmoid function**, but the **error function is convex¹²** and can be **optimized by standard gradient-based optimization techniques**. Indeed this approach is **easy to adapt to the online learning setting** (e.g. perceptron), where the sample are not available all together but they come in mini-batches, **for example using stochastic gradient descent** computing the gradient for each of the mini-batches continuously updating the estimate of w by following the direction of the gradient (as seen for the perceptron).

NOTE - Activation function If we replace the sigmoid with a step function we obtain the perceptron algorithm. Both algorithm use the same updating rule $w \leftarrow w - \alpha(y(x_n, w) - t_n)\Phi(x_n)$, since the gradient of the loss function is the same for both methods $((y(x_n, w) - t_n)\Phi(x_n))$. Replacing the sigmoid with the step function the prediction is not a continuous value that can be considered as a probability, indeed it is a discrete value +1 or -1 that has zero loss gradient in case of a right classification and $\Phi(x_n)$ loss gradient in case of a miss-classification (the $(y_n - t_n) = 2$ can be omitted due to the fact that w will define the same boundary no matter its magnitude, so α could be anything). The prediction, not being anymore a continuous value cannot be interpreted as a probability, so the choice of miss-classification becomes binary: a sample is correctly classified or not, differently from the logistic regression setting where each sample has a certain amount of miss-classification.

4.4.2 Multiclass logistic regression

Logistic regression can be expanded to multi-class classification. Before starting, it can be useful to talk about the role of the sigmoid function in standard logistic regression. **σ is used to remap the infinite space $w^T\Phi$ in a finite output space (0,1).** In other words it goes from the linear solution that could have every real value ($w^T\Phi$) to a solution that could have values inside the (0,1) range. Furthermore, **the output space have to be a probability distribution, so every output must be between 0 and 1, and the sum of the two classes must be 1.** In logistic regression the first property is ensured by the sigmoid function and the second by the fact that $P(C_2|\Phi) = 1 - P(C_1|\Phi)$. In the **multi-class case, we have to find a new way to ensure that the second property**

¹²Convex in this case implies that $L(w)$ has only one minimum

is still valid. To comply with both properties we can use the **softmax** operator. If we have K classes we construct K classifier as follow,

$$P(C_k|\Phi) = y_k(\Phi) = \frac{\exp(w_k^T \Phi)}{\sum_{j=1}^K \exp(w_j^T \Phi)} = \frac{e^{w_k^T \Phi}}{\sum_{j=1}^K e^{w_j^T \Phi}} \quad (43)$$

where the **denominator is the normalization term**. The term $e^{(w_k^T \Phi)}$ is an **exponential value that tries to tell how much the example x_n belongs to the class k , and can clearly be outside the interval $(0, 1)$** (indeed is in the interval $(0, \infty)$), then it is normalized using the sum of the value across all the classes, obtaining a value in the $(0, 1)$ range.

NOTE The model has a vector of parameter w_j for each of the K classifiers.

NOTE Having only two classes the softmax formula reduces to the logistic sigmoid function:

$$y_1(\Phi) = \frac{e^{w_1^T \Phi}}{e^{w_1^T \Phi} + e^{w_2^T \Phi}} = \frac{e^{w_1^T \Phi}}{e^{w_1^T \Phi} \underbrace{1 + \frac{e^{w_2^T \Phi}}{e^{w_1^T \Phi}}}_{1 + e^{w^T \Phi}}} = \frac{1}{1 + e^{w^T \Phi}}$$

with $w = w_2 - w_1$.

Differently from generative models, here we will **use maximum likelihood to determine parameters of this discriminative model directly**. Given T , a [NxK] matrix containing all output vector t_n [1xK] (one-hot encoded vectors) the likelyhood can be computed as:

$$\begin{aligned} p(T|\Phi, w_1, \dots, w_K) &= \prod_{n=1}^N \underbrace{\left(\prod_{k=1}^K p(C_k|\Phi(x_n))^{t_{nk}} \right)}_{\text{Only one term corresponding to correct class}} \\ &= \prod_{n=1}^N \left(\prod_{k=1}^K y_{nk}^{t_{nk}} \right) \end{aligned}$$

where:

$$y_{nk} = y_k(\Phi_n) = p(C_k|\Phi_n) = \frac{e^{w_k^T \Phi}}{\sum_{j=1}^K e^{w_j^T \Phi}}$$

NOTE The expression is very similar to the binary classification case. What changes is that in the the binary case \prod_n^N contained only the product of the probabilities of the two classes, instead here having K classes they must be considered all. So $y_n^{t_n}(1 - y_n^{t_n})^{1-t_n} = p(C_1|\Phi(x_n))^{t_n} p(C_2|\Phi(x_n))^{1-t_n}$ extends to $\prod_{k=1}^K p(C_k|\Phi(x_n))^{t_{nk}}$. In both the latter expression **for each n only one y_{nk} out of the \prod_k is different from 1, since t_{nk} is one hot encoded if $K > 2$ and for $K = 2$ is $\in \{0, 1\}$** .

NOTE t_{nk} is the $k - th$ target value of the example x_n : if x_n belongs to the $i - th$ class then only the $i - th$ element of t_n would be equal to 1, and all the other equal to 0.

As before the loss function would be equal to the likelihood:

$$L(w_1, \dots, w_K) = p(T|\Phi, w_1, \dots, w_K)$$

Taking the negative logarithm of the loss function we obtain the **cross-entropy function** for the **multi-class classification problem**:

$$L(w_1, \dots, w_K) = -\ln(p(T|\Phi)) = -\sum_{n=1}^N \left(\sum_{k=1}^K t_{nk} \ln(y_{nk}) \right) \quad (44)$$

and then the gradient for the $j - th$ class will be:

$$\nabla L_{w_j}(w_1, \dots, w_K) = \sum_{n=1}^N (y_{nj} - t_{nj}) \Phi(x_n) \quad (45)$$

that can be used to update the weights w_j associated to the classifier of the class j . You need to update the weights for all the classes using each example, but for computing the update for the weights of each class the latter formula is enough.

NOTE The gradient formula is the same seen in the past for both logistic regression and regression.

NOTE As before the reason of the negative sign is to transform the maximization problem into a more traditional minimization problem since we are speaking of loss/error. Maximizing would change the sign of the gradient since the direction would be opposite.

ATTENTION To use cross entropy there isn't any condition (other than i.i.d. samples).

NOTE The loss function is equal to the likelihood function, indeed in maximum likelihood we are optimizing the likelihood, so the optimization function, i.e. the loss function, must be the likelihood.

4.4.3 Connection between Logistic regression and Perceptron Algorithm

Perceptron could be seen as a particular case of logistic regression: if we replace the logistic function with a step function.

$$y(x, w) = \frac{1}{1 + e^{-w^T \Phi}} \rightarrow y(x, w) = \begin{cases} 1 & \text{if } w\Phi(x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Actually this happens if the scaling factor in the logistic function is changed making the latter degenerate in the step function. So the logistic regression can be seen as a smooth version of the perceptron. This is useful since with the perceptron we have a clear response that is $x_n \in C_2$ or $x_n \in C_1$, instead the **logistic regression output a degree of correctness of the classification**, and this allows to find compromise in the error. Furthermore, the logistic function is **differentiable everywhere**. Hence the advantage is that we **solve the termination problem of the perceptron**, since with logistic regression we can **always find a solution even if the samples are not linearly separable in the feature space**. Of course, may happen that after we have set the threshold between 0 and 1, defining the separating boundary not all the samples are correctly classified if they are not linearly separable, but it will find a solution that is a **compromise** (i.e. there may be some miss-classification if the samples are not linearly separable in the feature space) and that maximize the likelihood of the data give your model.

NOTE In logistic regression case, since the output is a continuous value between (0,1) that can be considered as a probability, the separating boundary is not explicitly defined since each sample has a probability of being of one of each class.

The **updating rule** is the **same** for both:

$$w \leftarrow w - \alpha \nabla L = w - \alpha(y(x_n, w) - t_n)\Phi_n$$

the only difference is that **while in the logistic regression I have always a kind of miss-classification**, so I use all the examples to update the parameter since all the examples contribute to the error and so to the gradient, instead in the **perceptron only miss-classification produce an error** and so the gradient comes only from miss-classified samples; i.e. the **contribution** that construct the **error** and so the gradient are **in one case all the samples and in the other only the miss-classified ones**.

So logistic regression is one of the common used approach for linear classification.

NOTE The update rule is the same but in logistic regression you have always a value between 0 and 1 so all the examples contribute to the gradient (more or less), instead in the perceptron the answers are binary (0 or 1), so if the sample is correctly classified the gradient does not contain its contribution.

NOTE Another advantage of the logistic regression is that there are no local optima but only global optima, so with the gradient approach you always find the same solution, if it exists, i.e. if the input are linearly separable in the feature space.

NOTE Logistic regression is still linear classification, so you are still looking for linear separating boundaries. Changing features you can represent arbitrarily complex

separating boundaries in the input space. The usual problem is how I could find the features.

NOTE We have seen perceptron only for binary classification. It can be generalized for multiple classes but then you go in the neural networks field.

Theorem[section]

5 Bias-Variance and Model Selection

5.1 “No Free Lunch” Theorems

In this section we will talk about a generic learner performance on different problems. We want to know if a given algorithm is better than the others in every case. The short answer is no, **any two optimization algorithms are equivalent when their performance is averaged across all possible problems**. In a more formal manner we can define:

- $ACC_G(L)$ as the **accuracy of L on unseen (non-training) data (i.e. the generalization accuracy of learner L)**.
- \mathcal{F} as the **set of all possible concept $y = f(x)$ (all possible problems, i.e. functions)**.

where $ACC_G(L)$ is our performance metric, and L is any learner (e.g. machine, human).

Theorem 5.1 (No Free Lunch theorem). $\forall L, \frac{1}{|\mathcal{F}|} \sum_{\mathcal{F}} ACC_G(L) = \frac{1}{2}$, given any distribution \mathcal{P} over x and training set size N .

This means that **independently on the method used for learning and also independently of the distribution and size of the training set, the average accuracy of the learner over all the two class classification problem is 0.5, that means that performs as random guessing** (like tossing a coin). In other words, whatever the best algorithm you can imagine, it cannot perform better than random guessing. This can seem really strange. Looking a little bit more into particular the theorem is **stated considering the set of all possible functions**, so it needs a better understanding. The problem, indeed, is that **it consider the set of all possible functions \mathcal{F} , so each function is assumed to have the same probability**.

The theorem is talking about the problem of evaluation of a learner L . What is saying this theorem is that if the possible function is the set of all possible function, each one with the same probability, I have no possibility of learning because what I see in the training set can be completely unrelated on what happens outside the set. Hence, observing data is not useful for predicting data, because anything can happen and is impossible to train against any possible model.

The fact ML works depends on the assumption that not all the function are possible, i.e. we are assuming that in the problem we are trying to resolve there is some kind of regularities/structure that can be exploited for learning. If any function is possible it means that there is no regularity in our problem, i.e. there is uncorrelation between points, this would mean the impossibility to generalize. So machine learning could work only on problems where generalization is possible: if generalization is possible, not all the function can generate our data, but our data can be generated by a subset of function with some structure. Furthermore, using other words we can say that the model could not learn everything but only specific tasks, the model accuracy over the set of all models is the same of a random guesser. If you cannot propagate/correlate what you see in some point with other point machine learning is useless. The goal of ML is to learn a model that observing a point predict the value of another point, but the uncorrelation of the data due to the fact that any function is possible remove the possibility of learning anything.

NOTE In few words, considering the set of all possible function the problem is not learnable, because what you observe is unrelated to what you would need to predict. I.e. if all the function have the same probability of being the correct one the result is the one stated by the theorem; instead, if some function are more likely and other less (prior knowledge) things change.

Example If data can have only two target 0 or 1, seeing input of class 0 or 1, since could be any concept (i.e. function) we cannot say anything about other inputs.

NOTE Uncorrelation between data points means that we cannot extract information about data, since observing a data point does not tell anything of what could happen.

NOTE This concept is important since when we use machine learning we are assuming that there is a structure in the problem we are solving. Without a structure machine learning (learning from data) is hopeless since data points are uncorrelated from each other, so even with a huge amount of data learning is not possible. So structured data is good and not structured data is really bad :(.

NOTE The structure of the problem is not in the dataset but in the process that generate the dataset, but off course we can assume this but cannot evaluate this. The fact that the problem has a structure is a basis assumption we do for each machine learning theorem.

NOTE Observing data you could always find a structure, since we are observing only a partial amount of data. But if any function is equally likely, the structure is not real, is just an artifact of the sample seen, so learning is not possible.

NOTE There are many "No Free Lunch theorems", this is the one of classification. The other tell us more or less the same things.

ATTENTION We speak about stationary processes, i.e. processes that do not change through time.

Corollary 5.1.1. *For any two learner L_1 and L_2 , if \exists a learning problem s.t. $ACC_G(L_1) > ACC_G(L_2)$ then \exists a learning problem s.t. $ACC_G(L_1) < ACC_G(L_2)$*

This means that **there is no method that is superior to any other method**, again considering the assumption of the "No free lunch" theorem, that are: **considering the set of all possible functions \mathcal{F} and given any distribution \mathcal{P} over x and training set size N .**

The takeaway from these theorems is that we **must not expect that there is a learner always superior to the other on any task**: for each problem may exist a learner that is **superior since it exploits better the structure of the problem**. What we need is to have a toolbox with many different techniques: each of them may be useful in a certain problem. Furthermore, the **best way to proceed is to try different approaches and compare, leaded by experience and without focusing too much on the same algorithms**.

NOTE Hence, in practice we are saying that it doesn't exist a perfect learning algorithm that performs well in every scenario. So **every algorithm is "specialized" on a given learning task. No algorithm is universally better than another one**.

NOTE But on a specific problem you can have 100% accuracy.

A **structured dataset**, from a mathematical point of view, is a **dataset that has a probability distribution over all the possible function that is not uniform**. The **more the distribution has a peak over a small set of function, the more structure you have; the more is uniform the less structure you have**.

The question that comes to mind is: how could a deep neural network be less accurate than a single-layer one? Yes. Remember what we have said about the hypotheses space, overfitting, ...

5.2 Bias-Variance trade-off

This concepts will help us to better understand underfitting and overfitting. Indeed, the **bias-variance trade-off relates the dimension of the hypotheses space with the performance of the model**.

Since the analysis is similar for the different problems, we focus on the regression problem.

To efficiently select the model complexity we want to analyze its error on unseen data.

5.2.1 Bias-Variance decomposition

The bias-variance decomposition is a way of analyzing a learning algorithm's expected generalization error with respect to a particular problem as a sum of three terms, the bias, variance, and a quantity called the irreducible error, resulting from noise in the problem itself.

Assume to have a dataset \mathcal{D} with N samples generated as

$$t_i = f(x_i) + \epsilon$$

where ϵ is the noise of the data characterized by $E[\epsilon] = 0$ and $Var[\epsilon] = \sigma^2$ ($\epsilon \sim WN(0, \sigma^2)$). We observe x_i and t_i (i.e. we have them in the dataset). Our objective is to find a model $y(x)$ that approximate the function f as well as possible on unseen data.

Let's consider the expected square error (i.e. the error variance) of our model, trained over D , on an unseen sample x fixed having the true target value t :

$$\begin{aligned} E[(t - y(x))^2] &= E[t^2 + y(x)^2 - 2ty(x)] \\ &= E[t^2] + E[y(x)^2] - E[2ty(x)] \end{aligned}$$

The variance is equal to the subtraction of the second moment and the first moment squared: $Var[x] = E[x^2] - E[x]^2$.

So we sum and subtract the first moment square for t and $y(x)$

$$\begin{aligned} &= E[t^2] \pm E[t]^2 + E[y(x)^2] \pm E[y(x)]^2 - E[2f(x)y(x) + 2\epsilon y(x)] \\ &= E[t^2] \pm E[t]^2 + E[y(x)^2] \pm E[y(x)]^2 - E[2f(x)y(x)] - E[2\epsilon y(x)] \end{aligned}$$

But ϵ and $y(x)$ are uncorrelated so $E[2\epsilon y(x)] = 0$

$$\begin{aligned} &= E[t^2] \pm E[t]^2 + E[y(x)^2] \pm E[y(x)]^2 - E[2f(x)y(x)] \\ &= Var[t] + E[t]^2 + Var[y(x)] + E[y(x)]^2 - 2E[f(x)]E[y(x)] \end{aligned}$$

But $E[f(x)] = f(x)$ since $f(x)$ is a value

$$= Var[t] + Var[y(x)] + E[t]^2 + E[y(x)]^2 - 2f(x)E[y(x)]$$

But $E[t] = E[f(x)] + E[\epsilon] = f(x)$

$$= Var[t] + Var[y(x)] + (f(x) - E[y(x)])^2$$

$$= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(x)]}_{\text{Variance}} + \underbrace{E[f(x) - y(x)]^2}_{\text{Bias}^2} \tag{46}$$

NOTE $f(x)$ is the real value of the function in the considered point x .

NOTE The expectation value is performed over different realization of the training set D . Hence, the infinite models obtained by the training over the infinite realizations (each with N examples) are evaluated over a new unseen example x and then the

value is used to find the expected square error.

As we can see the **variance of the error** comes from **three term**, so we can say that we have **three sources of error**:

- σ^2 : This is the **irreducible error**. It generates directly from the problem. It represent the **intrinsic noise**. We **cannot do anything about it**, indeed the only thing we can do is change the model y , and this does not depends upon y . It **cannot be eliminated even with infinite sample**.
- **Variance**: This is an **error from sensitivity to small fluctuations in the training set**. In particular this is the variance of the model training it over different realization of dataset D with the same number of samples N . Hence this **measure how much the model is sensible to the noise in the dataset**. Because if by changing the dataset (coming from the same function) the prediction change a lot, it means that the model is unstable. High variance can cause an algorithm **to model the random noise in the training data, rather than the intended outputs (overfitting)**, indeed high variance would mean that changing the realization of the dataset the prediction change. Hence, this term **measure how much the model is overfitting, how much the model is sensible to the training set** (indeed when the model overfits the data it is interpolating them).
- **Bias**: tell **how much the predicted value is far from the true one**, so if it is **high it means that the model is not able to represent the correct model of the function**. This is an **error from erroneous assumptions** in the learning algorithm. High bias can cause an algorithm **to miss the relevant relations between features and target outputs (underfitting)**. Hence, this term **measure how much the model is underfitting, how much the model is not able to represent the correct value of the function**.

Hence analyzing the term of the error the **variance σ^2 is not interesting for the selection of the best algorithm because it is present with whatever is the algorithm**, instead the other two terms are really interesting since they **represent the two unwanted behaviour of the model: overfitting and underfitting**. Of course our objective is looking for a **model that minimize the error, i.e. that minimizes the sum of the variance and the bias**. As we will see there must be a trade-off, indeed in most of the cases decreasing the bias would mean to increase the variance and vice versa. We are not interested in a model with low or zero variance but high bias, and the same thing for the bias. **A good estimator/ML model has a low error, and low sum of variance and bias, not low variance and high bias or vice versa**. Indeed:

- **A model with low bias is a very complex model** able to represent everything, i.e. with a **very large hypotheses space**, that indeed has a **very large variance if you do not have enough samples**. Being the hypotheses space very large the degree of freedom of the model are a lot, so the selected model will overfit data, which means

having an high variance over non-observed data, since what the model is representing is not the output function but is describing the noise of the training set.

- A model with **low variance** is a model with **small hypotheses space**, the **bias will be increased**. being the hypotheses space small the possible model are restricted, which means that some kind of wrong assumptions are being exploited, obtaining a model that cannot really capture the function we are trying to approximate.

Let's see a graphical representation of the bias-variance trade-off:

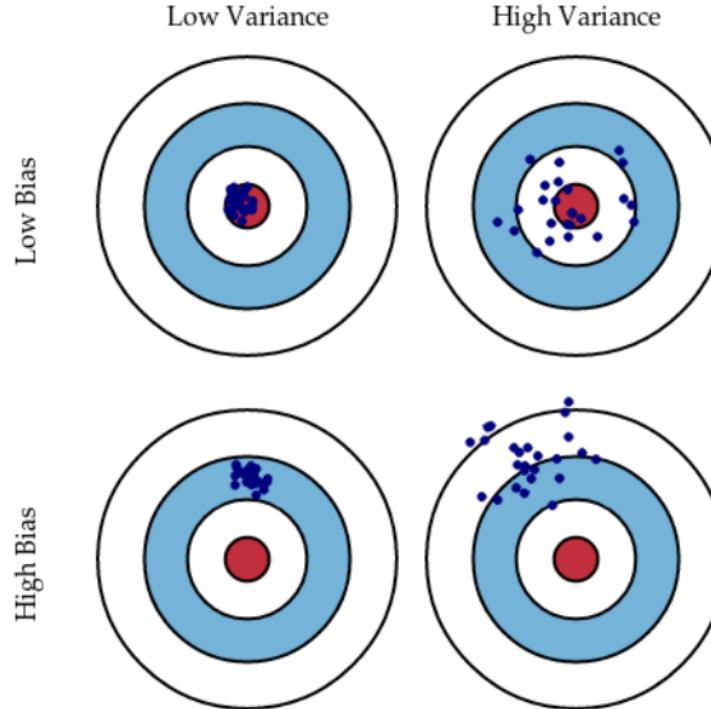


Figure 15: Points represent the prediction of an unseen example x done by models trained with different realization of the dataset. The circular red target represent the closest area to the true value.

In the upper picture:

- **Low Variance:** different training produce very close predictions.
- **Low Bias:** different training produce predictions very close (on average) to the real target value (red circle)
- **High Variance:** different training produce very far predictions.
- **High Bias:** different training produce predictions very far (on average) to the real target value (red circle)

The **ideal situation** is the top left one where we have **both low variance and low bias**, but unfortunately is **almost never reached**. Also the case with **high variance and high bias never happens since it would mean almost guessing at random**. Usually we are in one of the two remaining cases, but better in between these two cases since they are related to underfitting (low variance, high bias) and overfitting (high variance, low bias). Start from the underfitted model and trying to get the points closer to the **red target** but as we do it the variance increase and the points becomes more distant. Vice versa starting from the overfitted model and trying to get the points closer they would move away from the **red target**. Hence, as said, we **must find a trade-off**.

NOTE Remember that the expectation is performed over the infinite different realization of the training set D (which means sample in different point in time for a time series), each with N samples. So the analysis of the error above is done considering infinite realization of the dataset with a fixed number of samples N .

NOTE Changing N would change the variance errors as we have seen overfitting depends on the amount of samples.

NOTE The model is trained over the dataset D but then the error is evaluated over a single unseen example x .

Summarizing, sampling a dataset D multiple times you expect to learn a different $y(x)$. The expected hypothesis is $E[y(x)]$.

- **BIAS:** The bias is the difference between the truth and what you expect to learn i.e. the squared (for bias²) distance of the average model prediction around the true value (in the above picture: the distance between average point in the blue point group and the center of the red circle). Above we have computed it only on a single unseen sample x , but to **compute it over the whole input domain**:

$$\text{bias}^2 = \int (f(x) - E[y(x)])^2 p(x) dx$$

where $p(x)$ is the probability of observing the value x . The bias is reduced considering more complex models (e.g. in linear models by adding more features, i.e. enlarging hypotheses space). Is not reduced considering an higher number of samples, since it is independent to it (***)�.

- **VARIANCE:** The variance is the difference between what you learn from a particular dataset ($y(x)$) and what you expect to learn ($\hat{y}(x)$) i.e. the variance of the model prediction around the average model prediction value (in the above picture: the variance i.e. the concentration/distance of the blue points

around the average position of the group of points). Above we have computed it only on a single unseen sample x , but to compute it over the whole input domain:

$$\text{variance} = \int E[(y(x) - \hat{y}(x))^2]p(x)dx$$

where $\hat{y}(x) = E[y(x)]$. The **variance decreases with simpler models** (e.g. less features i.e. smaller hypotheses space, regularization) and also with more samples, indeed the **variance disappear with infinite samples**.

Having many many samples it means that the variance is not a problem and we would like to have an unbiased estimator, so an estimator with a very large hypothesis space (i.e. features): **the samples allow you to make the variance disappear so we could reduce the bias considering larger hypothesis spaces, without worrying about the increase of the variance since it is limited by the amount of data we have.** Having few samples, it would mean we are more concerned about the variance, so we do not want to use unbiased models, because the variance would increase as the bias decrease. So in order to control the variance of the estimator we would need to introduce the bias in our model.

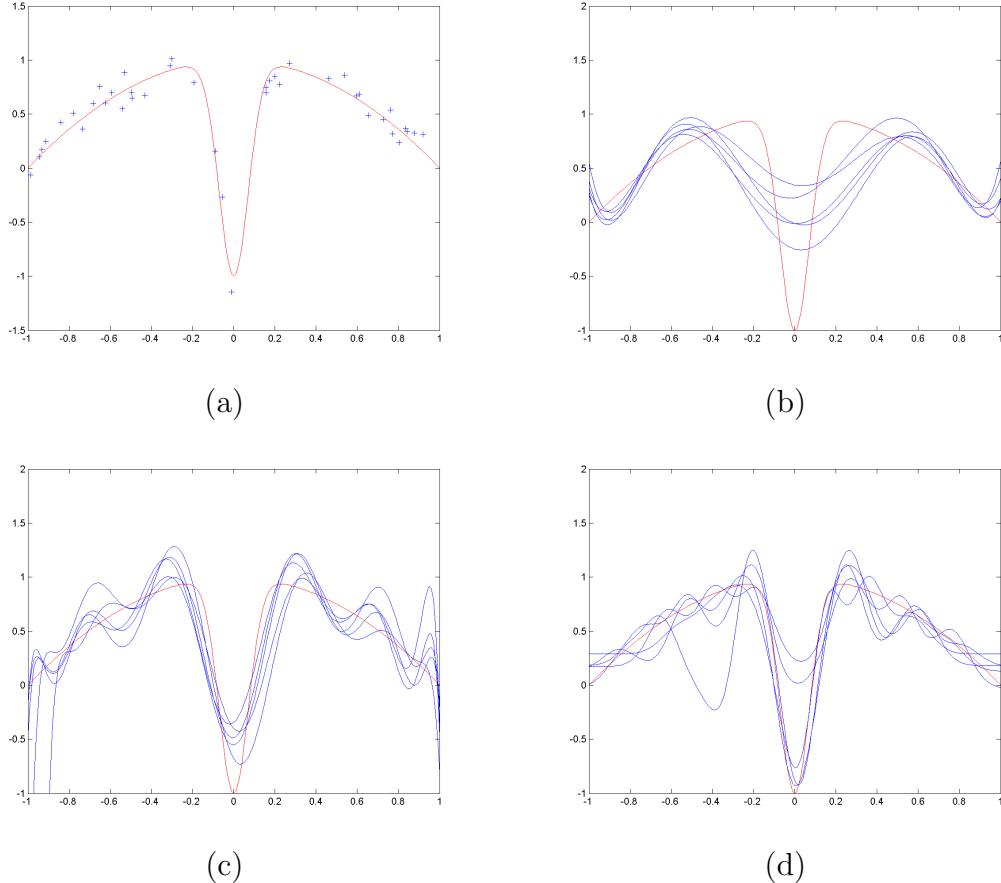
NOTE $E[y(x)]$ is the expected value predicted by the model: taking infinite realization of the dataset with N samples we train the models over this datasets and then we get the expected value over all this infinite models evaluating over x .

NOTE The bias definition is not the integral of the expected value of $f(x) - E[y(x)]$ but the integral of $f(x) - E[y(x)]$.

NOTE The latter and former formulas of bias and variance are computed over the whole input domain, but obviously the concept is the same of the one we have found from the error over the unseen sample x .

NOTE The bias represent the distance of the mean value (expected value, i.e. across infinite training sets) of the prediction $E[y(x)]$ (that is the mean value of the blue points) and the true value of the function $f(x)$ (the middle of the red zone). For this reason the bias over the whole input space is the integral of this distance (squared). Instead the variance of a single model is the distance of the expected model prediction in x and a certain the model prediction in x . For this reason the variance over the whole input space is the integral of the expected value, indeed the variance must be computed over all the possible models $y(x)$, i.e. all the possible training set of N samples, that is like averaging the squared distance of the center of the blue group of point and each point.

Let's see an example

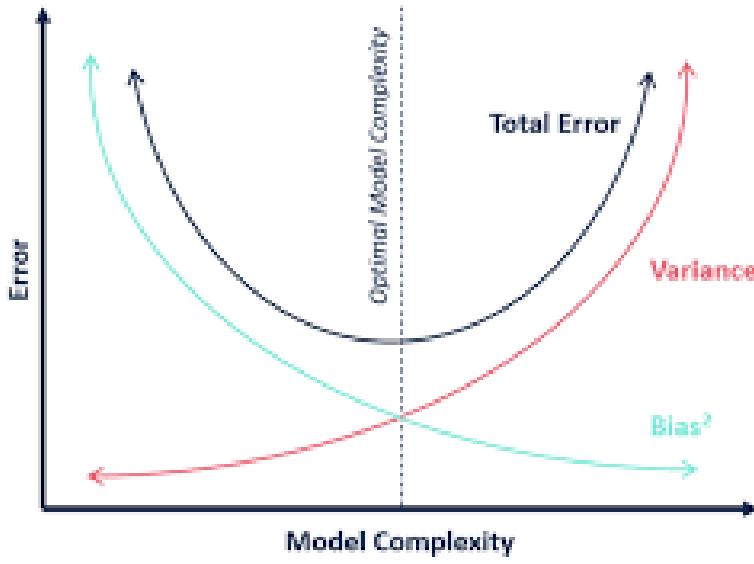


So let's take the dataset in figure (a) where the true function is the red line. For each case we take **five different realization** of the dataset and we estimate a model (blue line). In figure (b) we have an underfitting situation because our model is too simple. We can observe that we have a big bias because the models cannot predict data very well (even the dataset). But the variance between the models is very low. In figure (c) we have the right trade-off between bias and variance. In figure (d) we have a very low bias because all the models estimate the dataset well, but the variance between the trials is very big. In this case we are **overfitting the data**.

NOTE High variance means that the model is not generalizing well i.e. overfitting.

From what we have said, we can see how model complexity can affect the estimation error. Generally speaking **the bias decreases with model complexity and variance increase with model complexity** (considering the same amount of data during the training).

Remember Model complexity is proportional to the dimension of the hypotheses space (it can be seen as other things but this is a simple way to see it).



Simpler models imply less variance since when the hypothesis space is small the bias is high, because we are exploiting a lot of knowledge on the problem, in other words the bias we are putting in the model is a lot. By increasing the model complexity we are able to reduce the bias: adding more features the bias reduces. On the other hand complex models, i.e. having a lot of features, the variance becomes really high since the model will fit the noise of the data instead of the data itself, and this cause prediction far from the real value for points distant to the observed ones. We are not interested in models with either low variance or low bias, since we could see that their behaviour is opposite, but we are **interested in models with low error**. Machine learning problem objective is to find not an underfitted or overfitted model, but a model with the **optimal fit**, which means **having the best trade-off between variance and bias able to minimize the error** (ideally are both low), i.e. reducing bias and variance together not independently. The model able to minimize the total error must have the right complexity to balance between variance and bias, indeed as we have seen both depend on complexity.

NOTE Is important to notice that the **No Free Lunch theorem** says that there is **not a superior model if we consider unstructured data, for which each function is possible**, i.e. does not exist THE machine learning model superior for all the problems. Considering structured data exist a model superior to the other, i.e. that better exploits the structure in the dataset; this model and its complexity changes even depending on the amount of observed data, indeed saying problem we refer both to the function to learn and the size of the dataset.

NOTE In the figure the **irreducible error (σ^2)** coming from the noise of the data, is **not represented since is present in the same way no matter the complexity of our**

model.

NOTE When we consider training error and validation error to assess the performances of our model we should consider the two phases as two different groups of points: one related to observed data and the other to unseen data. The error showed in a graph is the average across the points of each group: **if the model has high training error it means that the bias is high, since the model is not able to predict a value close to the real one**, indeed the bias decrease by increasing the model complexity; **instead the variance could be seen as a ratio between the distance of the validation and training error and the magnitude of the training error**, if the training error is high also the validation one will be high but the variance will be low (high bias low variance), instead for low training error and high validation error the variance will be high (low bias high variance).

Example - K-NN In the case of **K-Nearest Neighbor** the input consists of the k closest training examples in data set. The output depends on whether k-NN is used for classification or regression:

- In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
- In k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

We can derive an explicit analytical expression of the expected squared prediction error

$$E[(t - y(x))^2] = \sigma^2 + \frac{\sigma^2}{K} + \left(f(x) - \frac{1}{K} \sum_{i=1}^K f(x_i) \right)^2 \quad (47)$$

- σ^2 is the irreducible error
- $\frac{\sigma^2}{K}$ is the variance term. It depends on the irreducible error and decreases as K increases
- $\left(\frac{1}{K} \sum_{i=1}^K f(x_i) \right)^2$ is the bias term. It depends on how rough the model space is. The rougher the space, the faster the bias term will increase as further away neighbors are brought into the estimates.

Now we are interested in **finding out how to measure the error (the black line) to find out which is the optimal complexity**. That is not easy at all since the error is **computed over unseen examples**, and since those are unseen data **i.e. data that is not in our possession**, we have to estimate it.

5.2.2 Training-test error

Given a dataset \mathcal{D} with N samples, we can split it, randomly, in a training set and a test set.

So far we have focused on the training error, indeed when we defined the empirical (i.e. computed over observed data) loss function (that we substituted to the theoretical loss function), we were computing the training error. **To calculate the training error we have to choose a loss function (e.g. RSS).** We can distinguish, for example, between:

- Regression:

$$L_{train} = \frac{1}{N} \sum_{n=1}^N (t_n - y(x_n))^2$$

- Classification:

$$L_{train} = \frac{1}{N} \sum_{n=1}^N I(t_n \neq y(x_n)) \quad I(t_n \neq y_n) = \begin{cases} 1 & t_n \neq y_n \\ 0 & t_n = y_n \end{cases}$$

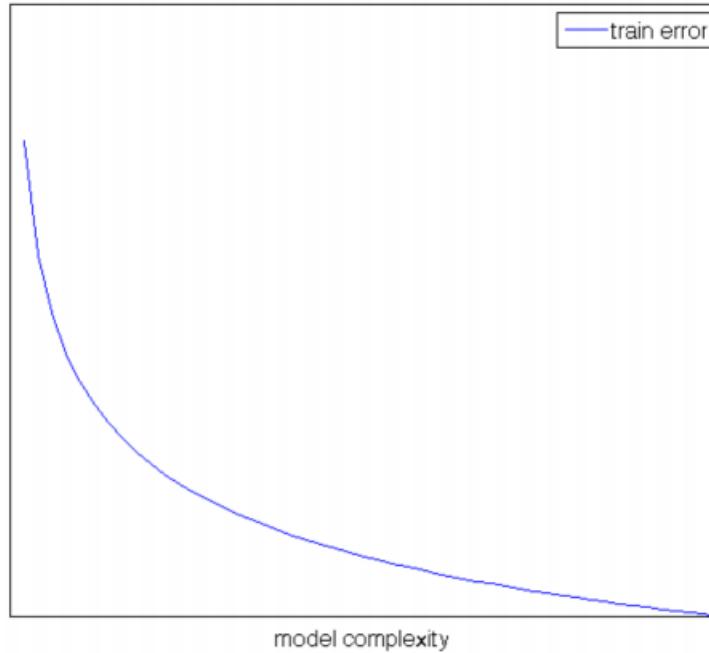


Figure 16: How the train error changes in relation with the model complexity (fixed the number of training set samples).

The training error measures how close our model is to training data. As we can imagine, increasing model complexity decreases the training error. So can we

say that the train error is a good measure for the generalization capabilities of the model, i.e. a good estimation of the prediction error on unseen data? No, indeed we also know that passed a certain complexity the generalization (i.e. the error over unseen data) capability of our model will be decrease, due to the variance increase. So training error is not a good estimator of our model performance because it is monotonically decreasing with model complexity, so it is an optimistically biased estimate of prediction error, that completely disregards the variance of the model (***)). So, if we want the training error can be a noisy but good estimate of the bias², since it does not consider the variance component of the error. This error will be always lower than the prediction error, indeed it is an optimistically biased estimate of the prediction error, so is it like the training error comes from the sum of only two of the three terms of the prediction error: the irremovable error and the bias².

NOTE Optimistically biased estimation since our estimation of the error is biased by the fact that we are evaluating the performances with the same data we used for the training, so it like a subjective estimation and not an objective one.

Our objective is to estimate the true prediction error, over all the input points, not only over the training examples

- Regression:

$$L_{true} = \int (f(x) - y(x))^2 p(x) dx$$

- Classification:

$$L_{true} = \int I(f(x) \neq y(x)) p(x) dx$$

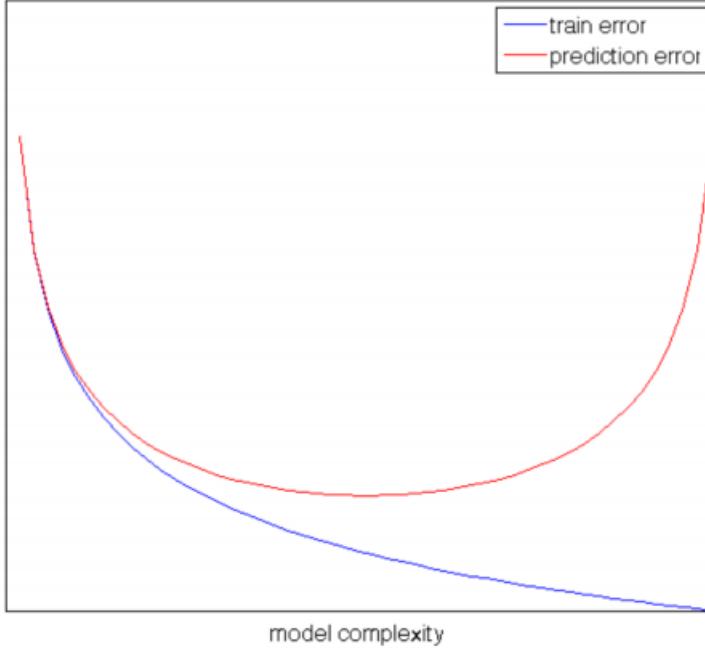


Figure 17: **Training error** (what we have) vs **prediction error** (what we want, but unknown), depending on the model complexity (fixed the number of training set samples).

So choosing the model by looking only at the training data you would obtain a model with bias² close to zero and high variance, i.e. we will always choose the more complex model that overfits data and has bias and training error to 0. Hence the **training error** is a good estimation of the **prediction error** (total error in the figure above) only for lower complexities where the variance is small, indeed by looking at the figure above when the variance is low then the error is almost equal to the bias² of the model, then the **estimation becomes worse and worse as the variance** (that is not considered) **increases**. The problem is that **when the model is simple (low complexity), it is underfitting the data**, so we are not really interested in those models and their performances.

The **true prediction error** over all input data is impossible to estimate directly because we would need the true model $f(x)$. A good way to estimate the prediction error is through the test error. The dataset is **randomly divided** into **training set** and **test set**. Then we use the training set to estimate the model's parameters (i.e. **optimize the parameters**) and the test set to estimate the prediction error (i.e. **evaluate the performance**).

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(x_n))^2 \quad (48)$$

The test error is calculated with the test set. It is **very important** to **not to use data of**

the test set for learning (i.e. not in the training set), in order to get an unbiased estimation of the prediction error. Furthermore, we must have a **trade-off** between training set and test set, indeed we need both **enough data to train the model but also enough data to evaluate its performance**: a great amount of data for training is good to obtain a better model, but a low amount of test data is not enough to select the best model; vice-versa having a lot of data for testing is good for the model selection but then a lower amount of training data does not let you to find better models.

NOTE To select the good division percentage between testing and training data the usual rules (e.g. 20% split) do not really work well, so you have to **look at the specific case considering how much noise is in the data and how much accuracy is needed in the evaluation**.

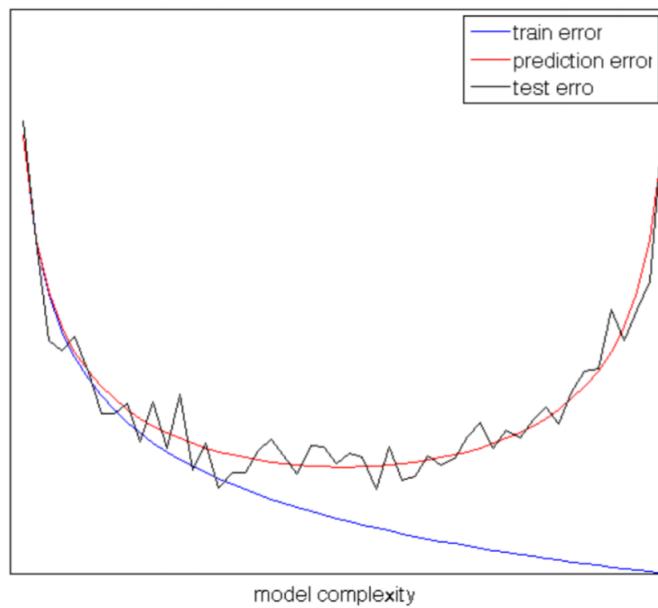


Figure 18: Test error as estimation of the **prediction error** (unknown), depending on the model complexity (fixed the number of training set samples).

The **noise in the test error in the latter picture would be larger as the samples for the computation of the test error decrease**. So to obtain a good approximation the noise should be reduced enough, otherwise **if the noise is large is a problem since the minimum could be in a position far from the true one**, meaning that the complexity we choose is not the best one.

NOTE The test error is unbiased only if none of the data in the test set is used for learning the parameters of the model, but even in parameter selection, i.e. complexity selection.

The **test set** (as we will see better in the future), **cannot be used even to do model**

selection (i.e. parameter selection). Indeed, **using the test error for choosing the model complexity is just like using the test error for learning** but the test error must not be used for learning but only for evaluation. Taking the model with the minimum test error the error becomes biased since I am taking a minimum, that is an optimization and is a form of learning. So by using the test error to choose the model, the test error is no more an unbiased estimate of the performance of the model, since it was used in a minimization process. In other words, my estimation of the error is biased over the test set, indeed in this way we **risk to overfit the test set and then the test error becomes optimistically biased.** To solve the problem we will need a **third set of data, the validation set:** a set for training, a set for the model complexity selection and a set for the final evaluation.

NOTE The error used for the evaluation of performances must not be used in any of the learning phases, indeed the risk is that the estimation of the performances becomes biased by the data, indeed the choice we do in the learning phase would be guided by the same data we are using for the evaluation, meaning that the evaluation is not objective and the performance will be overestimated.

NOTE If we take the test error as the performance measure, this evaluation would be biased if we use the test error to select the best model, since it is a kind of learning. **The performance estimation would be biased over the test set.** All the data has been used to train the model, i.e. observed and used in optimization, so all the estimate over those data are optimistically biased with the **risk of overfitting** them. In other words, our model is constructed upon that data, using that data for the choice of the final model, so if we use the same data to evaluate the performance of the model that evaluation may not be objective and cannot be valid in general since we only know that its performance are the best over that training set, so **the generalization may be bad since our error estimation is biased, influenced by the test set.** Using the test error for both model selection and performance evaluation then the latter is biased since **we are taking the model complexity in which we know that the test data have a low error but the model may be complex enough to overfit the data (not that the error is close to 0 but very low).**

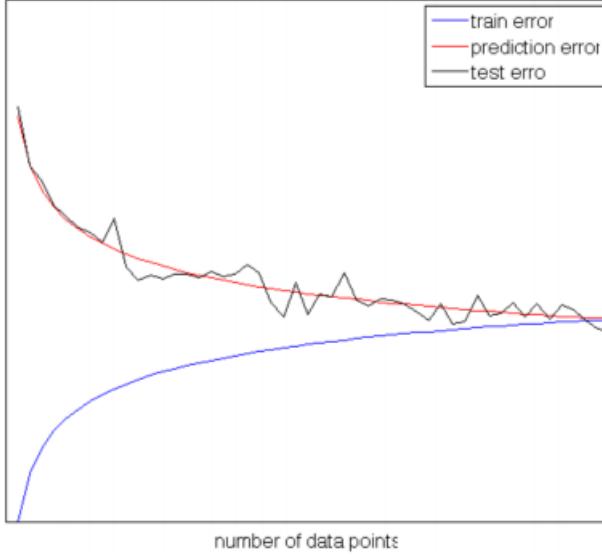


Figure 19: Test error as estimation of the **prediction error** (unknown), **depending on the number of data samples in the training set (fixed the complexity of the model)**.

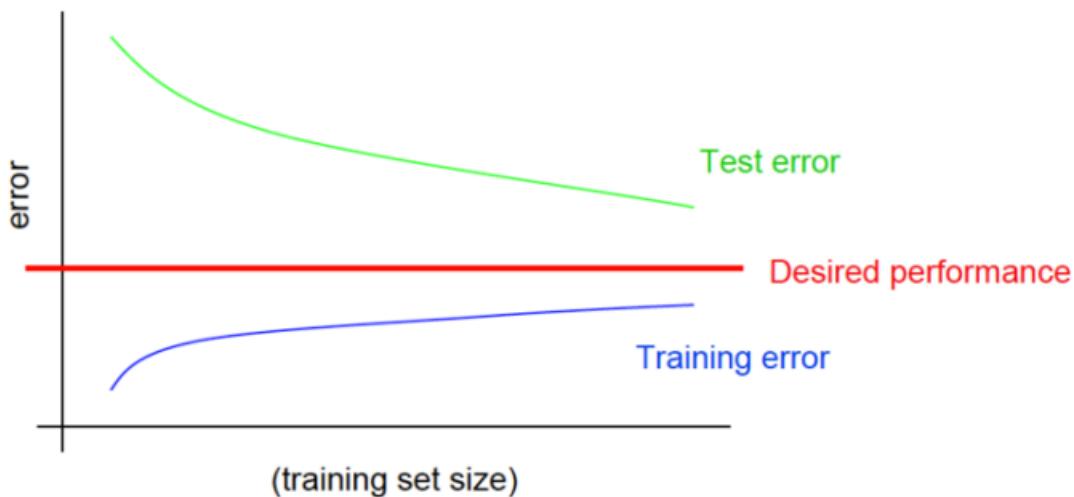
As expected with a small number of data the model may overfit them (the training error is 0 and the test error is very high, symptoms of bad generalization, i.e. high variance), but the more the number of the training set the larger is the training error, since the model complexity is fixed (hypotheses space fixed). On the other hand more data for the prediction error is always beneficial, because the **more the data the better the performance** of the model, **so the prediction error decreases**. Of course, when the number of samples is very high the **variance disappear and the gap between the training error and the prediction error vanishes**. This happens since the **training error**, as seen, does **not consider the variance term of the error**, but only the irreducible error and the bias², so if the **variance term becomes close to null then it becomes a good estimation of the prediction error** (error = irreducible + bias + very small variance). For the same reason the **variance could be seen as the gap between the training and the prediction error**.

Increasing the model complexity the train error goes down and the prediction error goes up, hence more data is needed to reach the situation where the variance is very low and the training error is a good estimate of the prediction error. This behaviour is expected since the the increase in complexity means that the hypotheses space is larger, so the variance is larger and is more difficult (i.e. more data is needed) to decrease.

NOTE The prediction error (in red in the figure) is NOT KNOWN, is what we are trying to estimate.

NOTE The training error is increasing since the complexity of the model is fixed, so increasing the training data the overfitting behaviour decreases which means that the variance is decreasing. Furthermore, being the training error a noisy estimate of the bias for lower complexities, as the trade-off between data and complexity becomes better it increases, becoming a better estimate of the bias. Since the variance decreases with the increase of samples what remain in the prediction error is the bias, so the training error becomes a good estimation of the former.

Let's consider a **situation** where we have a **fixed model complexity and a variable training set size**. Having the following situation, the model we are training is too simple or to complex?



The problem is that the **gap between the test error and the train error doesn't reduce so fast with the size of the training set, being not able to reach the point of zero variance** (training = test = desired performance). Since the **two error are far**, the model is affected by **high variance**, and when there is a lot of variance it means that the **model is too complex** because you are overfitting. The variance is really high so increasing the training set size the **variance is decreasing slowly because the model is too complex**. Hence when the test error and the training error have a **large gap**, it is a **sign of overfitting**.

Instead, having the following situation, the model we are training is too simple or to complex?



Here the **test error** and the **training error** converge even with smaller sizes (i.e. converge much quicker), so the **problem is high bias**, indeed **high bias, with few data, means low variance**. Indeed the test error comes close to the train error really soon, but the **performances are very bad because the model is underfitting, its too simple**. Hence when the training error and the test error are **too close and high you have the possibility of enlarge the hypotheses space, to improve the performances**.

Remember High variance is connected to overfitting, instead high bias is connected to underfitting.

Now we will see how the **bias-variance trade-off** can be managed using different **techniques**:

- Model selection
 - **Feature selection:** more features mean higher complexity which increase the risk of overfitting, so we **select the features from the original set**.
 - **Regularization:** usage of a hyper-parameters to regulate the trade-off.
 - **Dimensionality reduction:** somehow **related to feature selection**. In feature selection you select features from a set, instead **in this case starting from a set of features you can generate new features in different spaces with the idea of removing a redundant information or making the representation more efficient (so is related to the unsupervised problem of dimensionality reduction)**.
- Model ensemble: is able to **reduce bias or variance without increasing too much the other** (as we have seen happens usually and is the reason of the search of a trade-off)

- **Bagging**: is able to **reduce the variance** without increasing the bias too much.
- **Boosting**: is able to **decrease the bias** without increase the variance too much.

Each model in machine learning has a hyper-parameter that regulates the bias-variance trade-off, and as soon as we start a problem analysis we should look for it.

NOTE Dimensionality reduction may be better than feature selection. Imagine to have a model with two features, and that removing either one of them the model does not work well. Maybe with dimensionality reduction you may found a new feature that is the combination of those two features, that alone contains the same information of the original two features. Hence, **the hypotheses space may be reduced only resorting to dimensionality reduction and not feature selection**.

NOTE Three classes of model selection methods:

- Feature selection: identifies a **subset of input features** that are most related to the output.
- Regularization: all the input features are used, but the estimated **coefficients** are **shrunken towards zero**, thus reducing the variance.
- Dimension reduction: the **input** variables are **projected into a lower-dimensional subspace**.

5.3 Model selection

Model selection cover a key role in the performance of our model and can be performed in several ways. **Increasing the complexity of a model** means to add dimensions to the input space. This **could have really bad consequences**.

5.3.1 Curse of dimensionality

Why dimesionality is an important topic in machine learning? Why dimensionality reduction is important to use the data in a better way?

In ML we have the so called **curse of dimensionality**. It is related to the **exponential increase in volume associated with adding extra dimensions to the input space**, i.e. adding parameter to our model. I may want to use model with **many parameters because I want to reduce the bias and represent complex things**, but it has a huge cost. **Working with high-dimensional data is difficult** because:

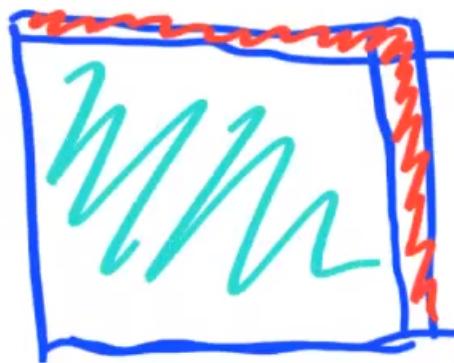
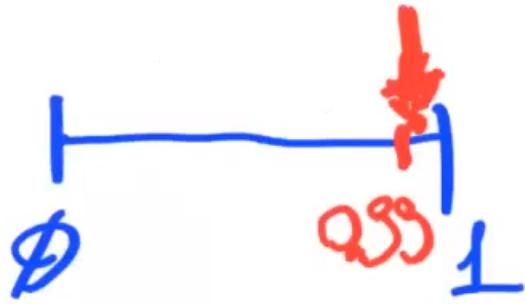
- the variance of the model becomes larger \Rightarrow overfitting
- need of many samples N^d that grow exponentially with the complexity of the model (exponential increase in volume).
- high computational cost i.e. high computational power to counteract this phenomenon.

A common pitfall when we can't solve a learning problem with few features is to add more features to the input space (it seems like a good idea). However the number of samples remain the same and so the importance of the old features decreases leading to worse performance. So the method with more features is likely to perform worse instead of better (as we expected after what we thought was a good idea).

NOTE The exponential increase in volume is related to the fact that adding a feature will increase the volume of the data needed to cover the feature input space, so that the parameter of the model learned are meaningful.

NOTE As we will see, if we are not able to properly assess the performance of the model, we will think that by increasing the complexity the performance are improving but this is not true.

To have a better grasp of what can lead to an increasing search volume we can think of the following. Imagine the parameter space of a learning problem. Adding a feature will add a dimension in the parameter space. Let's imagine a uni-dimensional parameters space. Suppose we have two points on a line, 0 and 1. These two points are unit distance away from one another. What is the probability of sampling a point in the interval between 0.99



and 1 by taking a random sample in the interval $(0, 1)$? The probability is 1% so 0.01. Suppose we introduce a second axis, again distributed a unit distance away. Now we have two dimensional space, $(0, 1)^2$. What is the probability of sampling a point in the region between $(0.99, 1)$ for the first axis and $(0, 0.1)$ for the second one by taking a random sample in the region $(0, 1)^2$? The probability is $1 - (1 - 0.01)^2 = 1 - 0.99^2$ (failing after failing two times). If we iterate this process adding new dimension the probability of getting a sample in the region will be $1 - 0.99^n$. Furthermore, if n goes to infinite the probability goes to 1 since $0.99^n \rightarrow 0$. So this means that going with very high dimension with probability 1, you will sample in that very small region, so taking random samples in high dimension with probability of almost 1 they will be close to the boundary of the region (red region), instead the internal part (blue region) will have probability close to 0 to be sampled. **Hence to fill well all the possible combination, you will need an exponential number of samples to compensate this phenomenon.**

Other explanation of curse of dimensionality To have a better grasp of what can lead to an increasing search volume we can think of the following. Imagine the parameter space of a learning problem. Adding a feature will add a dimension in the parameter space. Let's imagine a uni-dimensional parameters space. Suppose we have two points on a line, 0 and 1. These two points are unit distance away from one another. Suppose we introduce a second axis, again distributed a unit distance away. Now we have two points, $(0,0)$ and

(1,1). But the distance between the points has grown to $\sqrt{2}$. If we iterate this process adding new dimension the two point will go further away. More formally, consider a p -dimensional hyper-cube with unit volume. Suppose that we have n points uniformly distributed inside the hyper-cube. Let r be the ratio of points inside the cube which are within some neighborhood. To capture an r -full of points in the data, we need to grow a cube which takes up r of the unit cube's volume. Since the length of an edge on the cube is simply 1, we have to find the cube edge e so that $r = e^p$ is equal to the desired volume. So expressed in terms of e_p , the edge length necessary to fill a p -hypercube is $e_p = r^{\frac{1}{p}}$. For example, to take 10% of the point in a 2-dimensional space we have $e_2(0.1) = 0.1^{\frac{1}{2}} = 0.31$. Similarly, for a 10-dimension space we would have $e_{10}(0.1) = 0.1^{\frac{1}{10}} = 0.8$. To include 10% of the data in a 10-dimensional space we need to take up to 80% of the possible parameters values, in contrast in a 2-dimensional space we only need 31%. The searching space grows exponentially.

The ability of selecting the features or reduce the dimensionality of the model is useful, since if I am able to remove the input information that are useless for the problem, this helps the model to focus only on the parameter relevant for the problem and do not waste samples to learn useless parameters.

REMEMBER The more is the dimension of the input feature space, the larger is the number of samples that you need to cover this large input space.

5.3.2 Feature selection

Identifies a subset of input features that are most related to the output, i.e. the best subset of features to learn our problem. Ideally, we can use the **following approach to select the best features**.

1. Let \mathcal{M}_0 be the null model which contains no input feature (it simply predicts the sample mean for each observation)
2. For $k = 1, \dots, M$ (M different features)
 - (a) Fit all $\binom{M}{k}$ models containing k features
 - (b) Pick the best model and call it \mathcal{M}_k . To define the *best* model we need to define a metric.
3. Select a single best model among $\mathcal{M}_0, \dots, \mathcal{M}_M$ using some criterion (cross validation, AIC, BIC,...)

The best subset selection **has problems when M is large**:

- **Computational cost:** The problem is that the **problem is combinatorial so the number of subset** with a certain number of features m is **exponential in the number of features 2^m where m is the number of features**. So it is not practical

to try all the possible subset to select the best one, because you would need to **try too many models**. Hence feature selection **try to use an heuristic approach to select a good subset of features**.

- **Overfitting:** if you compare all the possible subsets, i.e. you are looking at many possible alternatives of the hypotheses space, the risk is that you are overfitting that information since **every time that you compare a large number of alternatives the risk is overfitting**, i.e. the risk that the estimator suffers of a large variance due to the fact that the hypotheses space is too large.

To solve this problems there are 3 commonly used **metaheuristics**:

- **Filter approaches:** use **statistical tools to find how much a feature is useful for the problem**, i.e. the correlation between the feature and the output variable and then **rank the features and select the best ones**. e.g. by looking at the correlation between the feature and the target variable.
 - **Forward step-wise selection:** starts from an empty model and adds features **one-at-a-time** to see which are the performance at each step, taking the model that performs better on a validation set. At each step you **try all the features from the set that were not selected before**. Usually, you go on until adding more features do not give an improvement.
 - **Backward step-wise elimination:** starts with all the features and removes the least **useful feature, one-at-a-time**, i.e. each time consider the model without one of its features then **choosing the features whose removal less impact on the performance of the model**. Usually, you go on until removing more features imply a significant drop in performances.
- **Embedded (built-in) approach:** algorithm that **try to solve the supervised learning problem directly but they implicitly apply feature selection while they solve the problem**. e.g. Using lasso and changing the regularization parameter we can select features, indeed the weight of some features goes to 0, meaning that it is useless. Hence lasso can be **used to understand which features are useful**, and then we may use them with another technique to find the model (e.g. neural networks). Another example is using decision trees or auto-encoding to select the features.
- **Wrapper approach:** tries to **evaluate some subsets of the feature set**

NOTE **Wrapper approaches** are **greedy**, they are **not solving optimally the problem**, i.e. they are not finding the best subset selection because **they do not try all the possible combination but they perform in a greedy way**.

NOTE Filters are often used due to their efficiency but often they are not so accurate, there may be correlation not captured.

NOTE Using different embedding techniques will lead to different subset of selected features.

5.3.3 Choosing the optimal model

We can start from understanding how we can evaluate and select a single best model, i.e. **how we can choose which model is the best one**. Obviously we **can't use training error**, because the most complex model, with all the features, will perform "better", even though is overfitting, being training error is **optimistically biased**. We would like to **choose the best model among a collection of models with different number of features that has a low test error**, not a low training error, i.e. **a low error over sample that we have not used for training**. Therefore, RSS and R^2 are not suitable for selecting the best model among a collection of models with different numbers of features. There are **two approaches to estimate the test error**:

- **Direct estimation** using a **validation approach** (i.e. uses the validation set)
- Making an **adjustment to the training error to account for model complexity**.

Direct estimation So far we have considered to randomly split the dataset into two subsets: training set and test set. We do so to **decouple the test data** from the training phase in order to have an **unbiased estimation of model performance**. This decoupling must be preserved so **we can't use both training and test data to evaluate the various models and their features**. The choice of the model through the test error would make the test set part of the training set: **test data must never be used for learning and choosing the model is part of learning**. In other words to have an unbiased and objective performance evaluation the set used for the evaluation must be independent from the set used for learning (i.e. model choice). **To have a fair evaluation** we can introduce a new subset of data, **randomly splitting the dataset into three parts**:

- **Training dataset:** The sample of data **used to fit the models**
- **Validation dataset:** The sample of data **used to measure the performance of different models learned with different kind of hypotheses space** i.e. different features. Hence, are used for tuning the learning algorithm (e.g. model selection), **tuning model hyper-parameters**¹³
- **Test dataset:** The samples of data **used to provide an unbiased evaluation of the final model fit**, the best we have found.

In practice, the training set is used to learn the parameters, the validation set to tune the hyper-parameters and the test set to evaluate the performance of our fit. Of course we **want to have each of this set as big as possible**:

¹³The hyper-parameters are those **parameters describing a model representation that cannot be learned by common optimization methods, but nonetheless affect the loss function**. An example is the regularization parameter λ in lasso.

- Few data for training means that the model will be weak.
- Few data for validation means that we risk of choosing a model that is not the best one.
- Few data for testing means that the performance of the best model found, cannot be accurately measure, so we cannot give much guarantees to the client about the performance of the model.

but since they are independent split of the same dataset there must be a trade-off, highly dependent on the considered problem.

NOTE The rule of thumb with different percentages for the split do not really work well. The split depends upon the dataset, how much data you have and how much is the noise of samples. Indeed for example a larger noise would require more data for testing otherwise the performance cannot be assessed in a good way.

Validation set can't be used directly to estimate the error, because is used indirectly in the training phase when we compare a lot of models with different hypotheses space to find the best one. The risk is indeed to overfit the validation set: larger the number of models we compare using the validation error the larger is the probability that you are overfitting the validation error. In this case the model we are comparing may overfit the validation error resulting in a wrong selection of the best model that does not really generalize well but overfits the validation set, more model we test more probable is to find that kind of (bad) model. Selecting the model over a certain set of data creates an optimistically biased evaluation of the model.

The **solution usually** is to use **cross validation** procedure. In practice we want to train the model on training set and evaluate it on the validation set. The novelty in the cross validation approach is that **training and validation set are not fixed**: every sample is used as training data and validation data in different phases of the learning process. The **goal** of cross validation is to **use as much data as possible for training**. As we have seen using fixed sized training and validation set if we want a robust evaluation from the training set, we need a lot of samples for validation i.e. we have less samples for training, so the trained models will be weak. **But once we have chosen the best model through the validation error, the chosen model is retrained using both validation and training set, and then evaluating the model on the test set.** The problem is that the choice was done considering model trained on a training set that was much smaller to the training set used after the choice (i.e. validation + training) so we are evaluating with the validation set something that can be far from what we finally deliver. Not only the model will be better but probably we could have chosen a model with higher complexity, because as we have seen the more the sample used for training the larger the complexity I can afford (and we used few

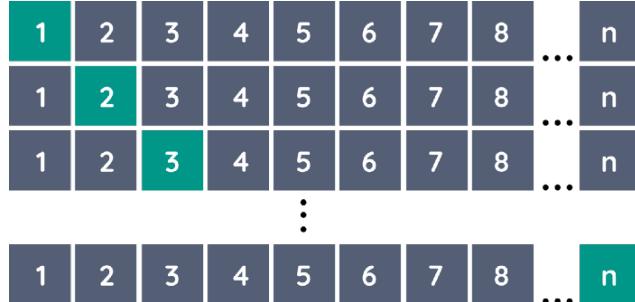
training set in the choice). Hence, a fixed validation set enforce us to use fewer data for training and usually to select simpler models than the one we could achieve. This is the problem cross validation tries to solve, i.e. tries to avoid to compare models trained with few training samples because we need more samples for validation. Of course nothing is for free and cross validation achieves this goal (evaluation of better model) but at a cost of much higher computational complexity.

We can randomly divide the training data into k subsets of equal size: $\mathcal{D}_1, \dots, \mathcal{D}_k$. We learn k times the parameters producing k different models, each time excluding a different \mathcal{D}_i . Then we evaluate the performance of every model on the relative excluded set. To estimate the validation error we can calculate the average of every subset \mathcal{D}_i error. Based on the parameter k we can have different type of cross validation error:

LOO (Leave One Out) In this case we consider at each iteration a validation set with only one sample ($k = N$). So considering N the number of data in the dataset, we could build N different couples of training ($\mathcal{D} \setminus x_i$) and validation (x_i) sets. On each couple we evaluate the validation error, and the final validation error is obtained by averaging all the validation errors. Once the LOO cross validation is applied to each hypotheses space considered we will choose the best model/hypothesis space by looking for the minimum LOO cross validation error L_{LOO} .

Algorithm 2: LOO cross validation

Output: Validation error L_{LOO}
Input : Data set \mathcal{D} with N samples
for $i \leftarrow 1$ to N **do**
 | Train model $y_{\mathcal{D} \setminus \{n\}}$ on $\mathcal{D} \setminus \{n\}$
 | $L_{\{n\}} \leftarrow (t_n - y_{\mathcal{D} \setminus \{n\}}(x_n))^2$
end
 $L_{LOO} \leftarrow \frac{1}{N} \sum_{n=1}^N L_{\{n\}} = \frac{1}{N} \sum_{n=1}^N (t_n - y_{\mathcal{D} \setminus \{n\}}(x_n))^2$



NOTE At this point we are not trying to find the best model over the N models but we want to know if the hypotheses space that each of these model

share, is good or not for the problem. So the goal of cross validation procedure is the model selection, the choice of the best hypotheses space, the choice of the best complexity of the model, i.e. the minimization of the validation error. We are looking for an unbiased validation error such that the model selection is itself unbiased. Indeed, cross validation solve the problem of model selection through fixed validation set, which is the risk of overfitting the set obtaining a biased model selection over the validation set considered.

Remember Talking about model complexity we are talking about the class of complexity in which we want to search our model, so it is the hypotheses space.

To evaluate different input features set, we apply the cross validation for every model having those input features. LOO is almost unbiased , only slightly pessimistic (as opposed to k-fold, but from a computational point of view k-fold is far better). So this is a very strong estimation of the validation error, since at the end when we average the N validation errors is like we are computing the validation error over N samples. Hence it is the best way to use the samples we have for the estimation of the validation error. The drawback is clear, the computational cost is not low, indeed the training not only must be repeated for a number of times equal to the size of the dataset (N) but also for each hypotheses space we want to compare (if the latter are M then the LOO algorithm must be repeated M times for a total of $N * M$ training steps). For example, if we have 100.000 samples and each iteration of the algorithm takes 1 seconds, computing L_{LOO} will take more than a full day. If you have to do it for every permutation of the input features it will take a very long time, making it not usable.

Example Linear regression supposing we want to use either quadratic or cubic features of the input. So we have two hypothesis space: the one with quadratic features and the one with cubic features. The dataset has 1200 samples, of which 200 are removed for testing. With LOO we build 1000 models using each time 999 samples for both quadratic and cubic features, having at the end two validation error estimation obtained averaging the validation error of the models over the same hypotheses space. Then the best hypotheses space will be the one with the lowest validation error. Finally the best model is trained using all the data (not considering the test set obviously).

NOTE The estimation is totally unbiased if we would be able to evaluate the validation error using all the samples for training, but obviously this is impossible. So it is almost unbiased because at the end we are not really using the same data for training and validation, but at the same time the validation is done using all the data. The fact that is pessimistic is due to the fact that at the end is like we are using all the sample to compute the validation error but we are averaging the errors obtained over a very small validation set.

K-fold cross validation k-fold cross validation is the **most used**, because is a **good trade-off between performance and accuracy**. Usually we have $k \sim 10$, using in an iterative way as for LOO each one of the k dataset split as validation set, **containing N/k samples**.

Algorithm 3: k-fold cross validation

Output: Validation error L_{k-fold}
Input : Data set \mathcal{D} splitted in $\mathcal{D}_1, \dots, \mathcal{D}_k$
for $i \leftarrow 1$ **to** k **do**
 | Train model $y_{\mathcal{D} \setminus \mathcal{D}_i}$ on $\mathcal{D} \setminus \mathcal{D}_i$
 | $L_{\mathcal{D}_i} \leftarrow \frac{k}{N} \sum_{(x_n, t_n) \in \mathcal{D}_i} (t_n - y_{\mathcal{D} \setminus \mathcal{D}_i}(x_n))^2$
end
 $L_{k-fold} \leftarrow \frac{1}{k} \sum_{i=1}^k L_{\mathcal{D}_i} = \frac{1}{N} \sum_{i=1}^k \sum_{(x_n, t_n) \in \mathcal{D}_i} (t_n - y_{\mathcal{D} \setminus \mathcal{D}_i}(x_n))^2$



NOTE Is the same algorithm as LOO where we have $k = N$.

The **properties** of this technique are the following:

- Much faster than LOO, indeed for each hypotheses space considered the number of models to train is not N but k , that is chosen by you.
- Smaller k faster is the computation for each hypotheses space, but also more pessimistically biased is the estimation. This means that for smaller k we are going to choose simpler model than we can achieve, indeed in that way we are going back to a situation similar to one of fixed validation and training set, where the training set are small, so with a smaller number of training data less complex model perform better due to risk of overfitting. This is why as a rule of thumb usually $k = 10$.

Again after we have selected the best hypotheses space we will train again the model using the whole dataset (of course leaving out the test set).

NOTE As k becomes higher the computational effort increase and the estimation is more unbiased and less pessimistic. Instead as k becomes lower the computational effort decreases but the estimation is more biased and more pessimistic tending to the behaviour of the fixed training and validation set, where the estimation is biased, the computational effort is reduced and the estimated model is simpler than the one we could achieve (pessimistic estimation).

NOTE It's worth mentioning that sometimes the validation partition is called test. Be aware that is not the same test set used ultimately to evaluate model performance.

NOTE Each part D_i is called fold.

NOTE The loss used, is of course, the one chosen for the problem.

Cross validation procedure We have a dataset D . One part of the dataset will be put away in the test set, not used in the procedure of model selection. The remaining dataset is used for the **cross validation** procedure that will tell us **which is the best hypotheses space** (e.g. set of features) to use **and hyper-parameters selection** (e.g. λ in regularization, prior of bayesian regression). The idea is to make different hypotheses of the latter things and by the mean of cross validation, find which is the best one. Once the best set of hypotheses is found then you use all the **examples of the dataset except the test set, for the training of the final model**. Now the final model performances can be evaluated on the test set. Once we compute the performance on the test set, what we get is not a number, we **should never give** to our client (that **ask for accuracy, the error**) a number, but we **should respond with an interval since we are never sure, of the accuracy or the error of the model, since those values have been computed using a finite set of samples, so they are affected by uncertainty** (uncertain estimation). Hence we have to produce an interval, the accuracy (e.g. accuracy between 95-98%).

ATTENTION If the client ask for a model with at least 90% accuracy and we have found a model with an accuracy interval of 85–95% then we do not satisfy the requirements, indeed even if on average the accuracy is of 90% you have not a high confidence of that accuracy because maybe there is a high probability that the performance are between 85–90%. What we should do now? You cannot do anything using the same data because anything you will do using the same data, i.e. you cannot try a new model, try other hypothesis, since you already done it and it could be done only once. **Indeed using the test set to know when the model meet the requirements is like using the test set for model selection, totally eliminating the purpose of the test set, since there is again the risk of overfitting.** Hence the **test set is a one shot evaluation**. Statistical tests are very tricky with numbers. Going back and forth repeating the model selection, sooner or later we will find the model that meets our requirements, even if this is not true. As we have said the

test set is no more unbiased since we are doing an optimization over the test set (we have many error over the test set and we will select the one that meets the requirements, the one with maximum/minimum accuracy/error), you are trying to find a solution that works well over the test set. So when we have to repeat the model selection procedure, we must **use a different test set, since you cannot work on the same test set. Not doing this we would obtain an optimistically biased model, i.e. not a good one.** At this point either the new test data is composed by new data available or, if no new data is available, you can **re-use the data but at the cost of reducing the confidence level: each time the data is reused it is losing power.** Each time the process is repeated the confidence interval must be increased, not for having a larger confidence but for having the same confidence you would like to have in the beginning, since every time you perform the confidence test you are pulling out a value from a random variable and so this means that . **Using over and over the same data will make the problem more difficult making the bar to jump higher and higher** (e.g. if the confidence level of being in the interval 85 – 95% at the first test is 95% then after each reuse of data the confidence level must be higher and higher e.g. 98%, 99% ...) [correzione di Bonferrone]

You have to guarantee that the confidence of all the test is bounded and not only the one of the single test, so it must be adjusted. Beware that **using another random split** for the whole process we are not changing the dataset, we are re-using the same data and the **problem of the test set overfitting remains.**

Hence, before the testing over the test set we have to be sure of what we have, and that we won't need to test a new model again. You can **re-use the validation set is possible, with a little bit of overfitting to take into account, but the real important thing is the test set:** maybe the model due to the overfitting over the validation set will not be the best model but the important thing is that the evaluation of the performances is done correctly. Usually what is done for the 1-confidence interval is using **Bonferrone correction**, that halves the confidence each time for instance, if we want a 5% of confidence of error (i.e. 5% of times we are outside the specified interval) each time that the process is repeated you halves the 5% (i.e. 1-confidence of 95%) which means increasing the 1-confidence. **Changing the confidence the bounds becomes bigger and bigger:** e.g. starting from 85-95% and repeating the process over the same test set, maybe the average improves but the bounds become larger e.g. 75-98% due to the increase in 1-confidence. **The increase in confidence is due to the fact that we have an overfitting of the test set, so maintaining the same interval and confidence would be an error, we must adjust the confidence and so the interval bounds to consider the fact that performances may seem better due to the overfitting but probably are worse, indeed this is considered by enlarging the confidence interval in both upper and lower bounds, and the confidence must be increased due to this bound enlarging.** The more you repeat the process the more the bound will increase not being able to catch up with it. The **main takeover** from this analysis is that **we must be sure to try all the alternatives that are worth considering, before the usage of the test set.** The other message is that **not all the problem can be solved**

through machine learning.

ATTENTION The interval has an associated confidence level that gives the probability with which the estimated interval will contain the true value of the parameter. A 95% confidence level does not mean that for a given realized interval there is a 95% probability that the population parameter lies within the interval (i.e. a 95% probability that the interval covers the population parameter). According to the strict frequentist interpretation, once an interval is calculated, this interval either covers the parameter value or it does not; it is no longer a matter of probability. The 95% probability relates to the reliability of the estimation procedure, not to a specific calculated interval. Furthermore, for a given estimation in a given sample, using a higher confidence level generates a wider (i.e., less precise) confidence interval.

Adjustment techniques This is the **alternative to direct validation**, and it is used when the latter is not practical, i.e. when a lot of samples are available and building many models is not possible due to the curse of dimensionality: the hypotheses space can be large and we can have more complex model but then the combination are much more. So this approach comes to help, tries to account for model complexity when evaluating the training error. The error expression, that you use to choose the model, will be consisting of two terms: a term that accounts and penalizes complexity of the hypotheses space and the term that accounts the accuracy (i.e. error term). There are several way to do that:

- C_p :

$$C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma}^2)$$

where d is the number of parameters, $\tilde{\sigma}^2$ is an estimate of the variance of the noise ϵ of the data. The term that account the complexity of the model is $2d\tilde{\sigma}^2$ and penalizes too complex models, i.e. with too many parameters d .

- BIC:

$$BIC = \frac{1}{N}(RSS + \log(N)d\tilde{\sigma}^2)$$

we replaces $2d\tilde{\sigma}^2$ of C_p with $\log(N)d\tilde{\sigma}^2$. Since $\log(N) > 2$ when $N > 7$, BIC selects smaller (simpler) models since the penalization for the complexity is more pessimistic.

- AIC:

$$AIC = -2\log(L) + 2d$$

where L is the maximized value of the likelihood function for the estimated model.

- $AdjustedR^2$:

$$AdjustedR^2 = 1 - \frac{RSS/(N - d - 1)}{TSS/(N - 1)}$$

where **TSS** is the total sum of squares. Differently from the other criteria, here a large value indicates a model with a small test error.

The error term for complex model that tend to overfit data will be very small, so by adding a term that penalizes complex model, we are enforcing that the resulting value is a trade-off between complexity and accuracy, hence overly simple or overly complex model will be penalized: the former from the accuracy term and the latter from the complexity term. So the big advantage of this techniques is that you can have an idea of what is the best model by using simply the training error as above, indeed the term to add (complexity term) can be simply known because is related to the characteristic of the hypotheses space. The important part is that the validation set is not considered, so the data split is done only to define the test set, and the remaining data is all part of the training set (as we would want to learn more complex models)(*). It's important to notice that this techniques are different to regularization: in regularization the penalty for the complexity is used during training, instead here is used after training (of course the training can even use regularization). We could say that in regularization the complexity penalization is actively used during all the training, instead in adjusted training error techniques the complexity penalization is (passively) used only in the last part of the training, the model selection.

Hence, the idea behind this technique is that we take into consideration that the training error is optimistically biased, and it try to obtain an unbiased estimation trying to approximate the variance, since we want to optimize an error that correctly accounts for both the variance and the bias² (and of course the irremovable error). The training error accounts for the bias² (and the irremovable error), instead the term added by this techniques accounts the variance considering the complexity of the space, indeed as we have seen the variance increase with model complexity. Since the training error is adjusted to take into consideration the variance (that was lost due to the fact that the error is computed on the same data used to train the model), the validation set is no more needed since its existence reason is to obtain an unbiased error estimation that consider both variance and bias² but we achieve the same result in other way. The advantage of this technique is clearly the savings of computational time, indeed there is not the need of training the same model across different folds (as in cross validation) but simply train on all the training set once and use the simple expression to evaluate it. Hence is less expensive than cross validation, and this really comes to help for problems with a lot of data and really big hypothesis spaces. This at a cost (disadvantage) of simpler problems than the ones we can afford, it is a trade-off between model complexity and computation effort.

NOTE The formula of these techniques are obtained considering some assumptions.

NOTE The bounds specified are quite loose so **usually you will obtain model a little bit simpler than what you can really afford/achieve.**

Example Assuming we have a linear model with K parameters and a dataset with $N + M$ samples. We split the dataset in training and test set respectively with N and M samples (**no validation set is needed**). Then we train the model using the training set and the method you prefer. **The value of the training error obtained is then used to evaluate the model using one of the adjustment techniques indices** (e.g. AIC). Once this is done for each hypotheses space we can select the best one by taking the one with the minimum value for the considered index.

NOTE As for cross validation the objective of adjustment techniques is the **model selection**, so the expression of the index (e.g. AIC) must be **evaluated for each hypotheses space considered** to choose the best one among them by taking the one with the minimum index value.

5.3.4 Regularization

To manage the complexity of the hypotheses space regularization comes to help. We have already seen regularization approaches applied to linear models (**ridge regression** and **lasso**). Such methods **shrink the parameters towards zero**. It may not be immediately obvious why such a constraint should improve the fit, but it turns out that **shrinking coefficient estimates can significantly reduce the variance** (*). A possible motivation could be that **smaller parameters are less prone to produce fast changing output function, so they are less prone to overfitting** because they can't interpolate directly all the samples. Regularization methods are very **useful when we have limited dataset and a large number of features compared to the dataset size**.

To visualize the **importance of choosing the correct regularization coefficient**, we can consider the following example. We pick $N = 50$ samples from a given function and we try to fit it with a model with 45 features. As a regression method we use ridge regression.

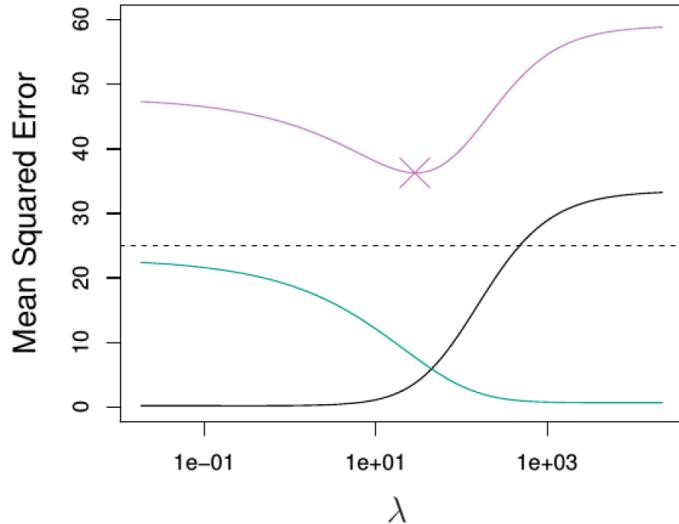


Figure 20: Ridge regression vs OLS. Squared bias, variance, MSE, Dashed: minimum possible MSE, cross: ridge regression model with minimum MSE.

From the figure above we can see in action the bias-variance trade-off. When λ gets bigger we obtain simpler models (high bias, low variance), so the variance is reduced and the bias gets bigger. We need to **find the optimal λ which minimizes the MSE**.

How we can find the best value for the λ parameter used for regularization? How much I have to penalize large weights? **The proper value for λ parameter is usually computed using cross validation, trying different values of λ and evaluating the performance of this different values using the cross validation error** (i.e. performing hyper-parameter tuning). Cross-validation provides a simple way to tackle this problem. We choose a grid of λ values, and compute the cross-validation error rate for each value of λ . We then select the tuning parameter value for which the cross-validation error is smallest. Finally,

the model is re-fit using all of the available observations and the selected value of the tuning parameter.

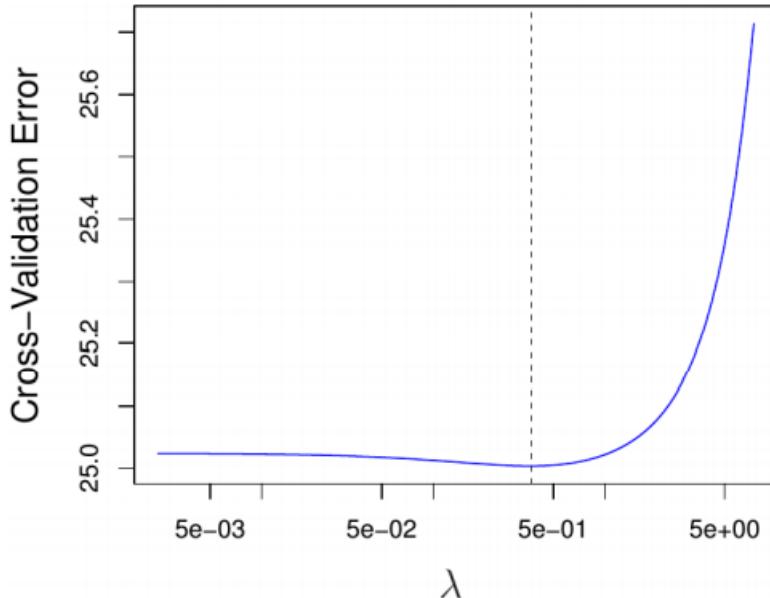


Figure 21: Example of cross validation applied to a model using ridge regression.

The problem of which lambda is better to consider is usually solved through trial and error, trying to find a U shape, in the sense that **when we use low value of λ we usually have bad performance due to overfitting (complexity not penalized enough)**, instead for **high values of λ we have bad performance due to underfitting (complexity too penalized)**, and in the middle the performance are better. Since we know the shape we want to reach we can decide to increase or decrease the lambda we picked, the former if the error value is decreasing with an increase of λ and the latter if the error value is increasing with an increase of λ . In the above figure the U shape is not so evident.

NOTE May also worth considering the following images:

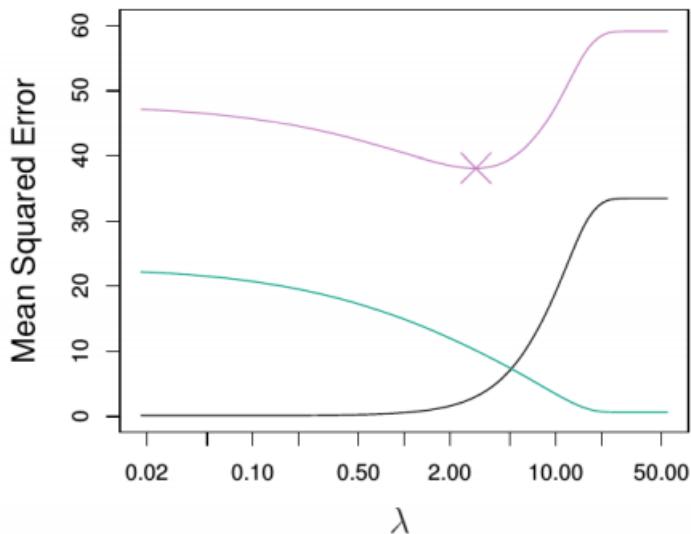


Figure 22: Lasso vs OLS. Squared bias, variance, MSE, Dashed: minimum possible MSE, cross: lasso regression model with minimum MSE.

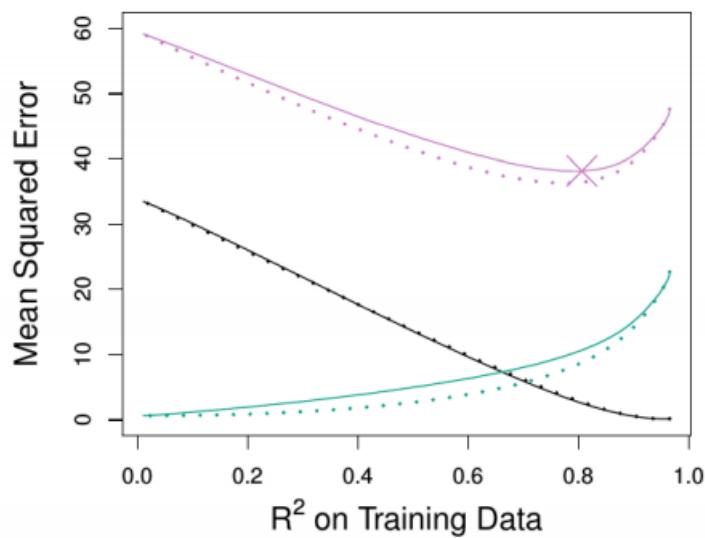


Figure 23: Lasso vs Ridge. Squared bias, variance, MSE, Solid: Lasso, Dashed: Ridge, cross: lasso regression model with minimum MSE.

NOTE R^2 in statistics is the **coefficient of determination**

$$R^2 = 1 - \frac{RSS}{TSS}$$

where TSS is the total sums of squares and RSS is the residual sum of squares:

$$TSS = \sum_{n=1}^N (y_i - E[y_i])^2$$

$$RSS = \sum_{n=1}^N \epsilon_i = \sum_{n=1}^N (y_i - \hat{y}_i)^2$$

where $E[y_i]$ is the mean of the observed samples and \hat{y}_i is the prediction over the i-th sample of the considered model \hat{y}

5.3.5 Dimension reduction

The approach we are going to present **differs from the previous approaches** because it **doesn't operate directly on the original features** (like the method seen so far), **but** has the **same goal of reducing the features number**. Dimension reduction methods instead of selecting the features from the original set like in feature selection, **just reduces the dimension of the features set transforming the original features** and then the model is learned on the transformed variables. Dimension reduction is an **unsupervised learning approach, since the hypotheses space is reduced without looking at the target variable**, i.e. the variable that encodes the supervision, **but simply looking at the input data/features**. The **goal is to find a smaller set of new features that retain the most information contained in the original features**. There are many techniques to perform dimension reduction,

- **PCA** (Principal Component Analysis): **produces new features that are linear combination of the original features.**
- **ICA** (Independent Component Analysis):
- Self-organizing Maps
- **Autoencoders**
- ...

NOTE Autoencoder, Self-Organizing Maps consider also non-linear combinations of the features.

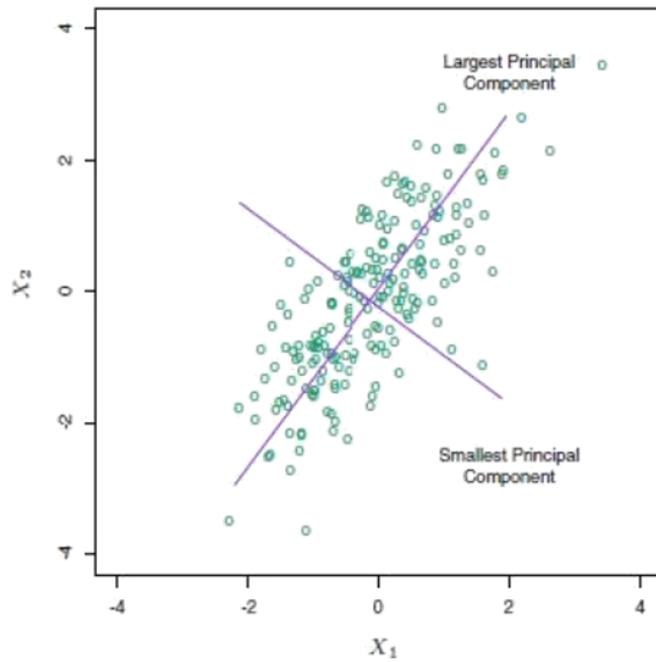
NOTE The approaches seen until now instead used the target value in the training, so they are not unsupervised.

For example features with constant value it means that that feature is irrelevant since it does not contain any information and can be removed. Even the

presence of linear combination of other features imply a redundant feature space and so the dimension of the space can be reduced by projecting it on another space (indeed the model itself will linearly combine the features). How I can measure the utility of a features or a combination of features? Starting from the observation that constant features are useless, one can say that a feature with low variance (i.e. the value of the feature is almost constant) is less informative than a feature with a lot of variance (i.e. the value of the feature changes a lot). Since this is an unsupervised approach even a feature with low variance may be important, and maybe more important of a feature with high variance, for the prediction of the target value, so the approach may be not precise. But since is an unsupervised approach the inductive assumption is that feature with low variance are less important than the one with higher variance, hence the dimension is reduced in order to retain as much variance as possible.

PCA The most used methodology is PCA. The idea is to find a new orthogonal base in the input space, which accounts for most of the input variance.

Let's see some intuitive concepts:



We can see that the data has **2 dimension**: one where the spread of the sample is larger i.e. where the **variance of the sample is higher** (**Largest Principal Component**, or **PCA 1st dimension**), and one where the spread of the sample is lower, i.e. **the variance of the samples is lower**, and orthogonal to the other dimension (**Smallest Principal Component**, or **PCA 2nd dimension**). By projecting the data over the original features, i.e. on the two axis, and computing the variance over those two axis the variance will be

more or less the same. Instead by rotating the data one feature will have a lot of variance and the other feature low variance, so the latter can be removed retaining most of the variance, i.e. the variability of the dataset.

NOTE The fact that the new feature set we find retain most of the variance, is an index of the capacity of the features of describing and most of the information of the variability of the dataset.

The idea of the algorithm is to find a line such that when the data is projected onto that line, it has the maximum variance. Hence I look for the main direction of the data, i.e. the one in which data is distributed the most. Then we find a new line, orthogonal to the first one, that has maximum projected variance (in higher dimensional space there are many direction orthogonal to the one we found, instead in a two dimensional space this direction is unique). We repeat this process until we have reached a number of principal components (lines) equal to the number of the original features. Once we have built this new representation of the points, that is simply a rotation of the original representation, then I can choose to remove the components that are less relevant, i.e. that have a smaller variance. Remember that all the component are ordered according to how much variance they have, so we will keep the first k components. Once the components are selected the value of each data point with respect to that component can be simply obtained projecting them over the component (line) itself that is like a axis (like when we have a point in a R^2 and we want to know its x value, so we project over the x-axis). Since the value is obtained through a simple projection, then the new feature space is simply obtained by a rotation of the original feature space, without some components (e.g. imagine a dataset with 3 features i.e. R^3 space, then we remove one feature, so the resulting feature space will be a R^2 space with the 2 principal component selected oriented as x and y axis, so they must be rotated to coincide with them).

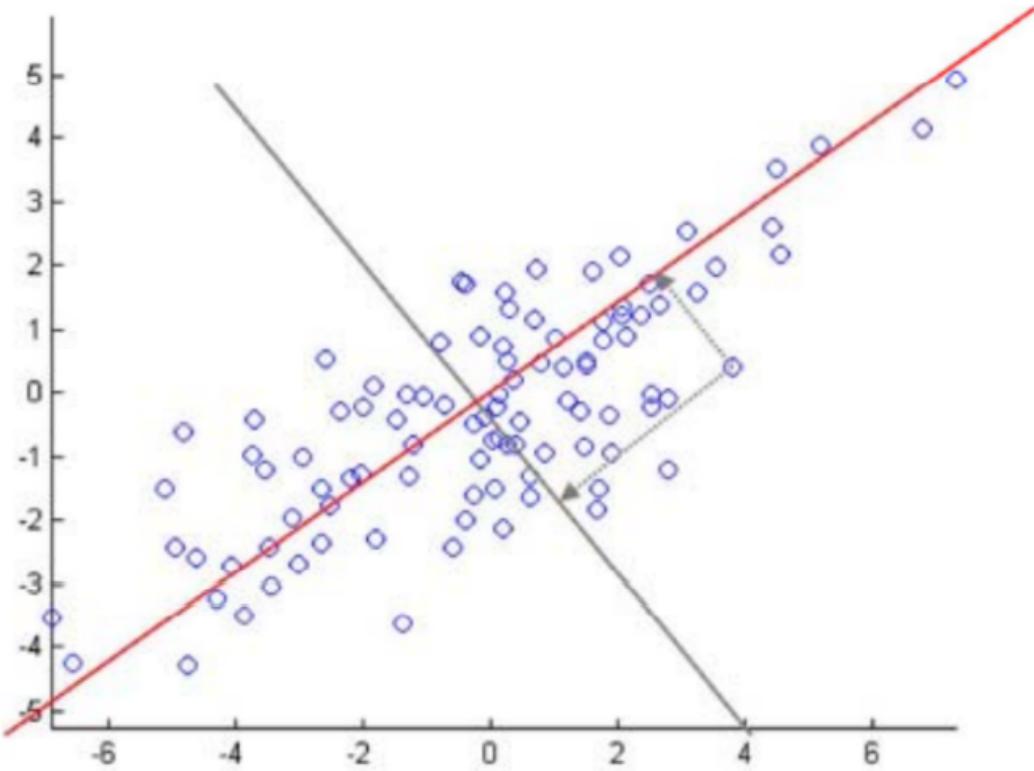


Figure 24: Data projection over the new feature space, i.e. components

In a more formal way, the complete process consists of,

1. Compute the mean (average) of all the samples

$$\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n$$

needed later to center the data on the origin for the computation of the variance matrix. Notice that \bar{x} is a **vector whose length is the number of features**, like x_n

2. Compute the covariance matrix

$$S = \frac{1}{N-1} \sum_{n=1}^N (x_n - \bar{x})(x_n - \bar{x})^T$$

That is the sum of the square difference between the value of the point and the average (i.e. centered data squared). Its size is [number of features x number of features], furthermore the diagonal contain the variances of the original features, and in the other places the co-variances. **This matrix describe how data are spread in the space.**

3. Get the eigenvalues λ_i and eigenvector e_i of S . The eigenvectors of S are the principal components of the data and the corresponding eigenvalue is the variance of data along the eigenvector, indeed in general eigenvectors are direction and eigenvalues is a measure of how much data are spread in that direction. Remember that the eigenvectors of a matrix are mutually orthogonal. Hence PCA provides a ranking over the new features, where the ranking is done considering the variance of the samples projected along that direction i.e. the eigenvalues.
4. Select the first k largest eigenvalues. The corresponding eigenvectors are the PCA components. Moreover:

$$\frac{\lambda_j}{\sum_i \lambda_i}$$

is the percentage of variance captured by the j^{th} principal component (PC), that can be extended to the first k component we decide to get as:

$$\frac{\sum_{j=1}^k \lambda_j}{\sum_i \lambda_i}$$

Using this expression we can find out how many features we have to consider if we want to have a new representation of the dataset that explains at least a certain percentage of the variance of the dataset ($k | \frac{\sum_{j=1}^k \lambda_j}{\sum_i \lambda_i} \geq \dots \%$)

Hence the full set of PCs comprise a new orthogonal basis for feature space, whose axes are aligned with the maximum variances of original data. Once we have found the new PCs, i.e. the k eigenvectors, we can project all the original data onto the new representation, i.e. the new orthogonal base:

$$E_k = [e_1 \ \dots \ e_k] [M \times k]$$

obtaining a reduced dimensionality representation of the data:

$$X' = X E_k, \quad [N \times k] \tag{49}$$

where X' is $[N \times k]$, X is $[N \times M]$ and E_k is $[M \times k]$, with M the number of original features and k the number of new features. Hence we go from the original dataset X (M columns) to the new representation of data with reduced dimensionality X' (k columns ; M columns, where the columns are the features of the sample). Notice that the new features of the new dataset X' are obtained through a simple linear combination of the original features of X using the value contained in the eigenvector. So each eigenvector contains the weights that you use to combine the original features to obtain a new feature, that is one of the principal component.

Furthermore, transforming reduced dimensionality projection back into original space gives a reduced dimensionality reconstruction of the original data. Of course the reconstruction will have some error, but it can be small and often is acceptable given the other benefits of dimensionality reduction.

NOTE The matrix of formed by the k eigenvectors E_k has M rows since each eigenvector is represented in the original feature space, i.e. in the original orthogonal basis with M components.

NOTE A single input value of the original dataset x_i (i-th row of X) is $[1 \times M]$, hence contain M values each one corresponding to a feature. So in the **multiplication for E_k** those M components are combined using as weights the values of the eigenvectors e_j (j-th column of E_k), each of which is $[M \times 1]$ and contains a direction of a principal component. In the end each row x'_i of X' will be the input representation over a new set of k features, more precisely the input values (i.e. projections) over the k principal components.

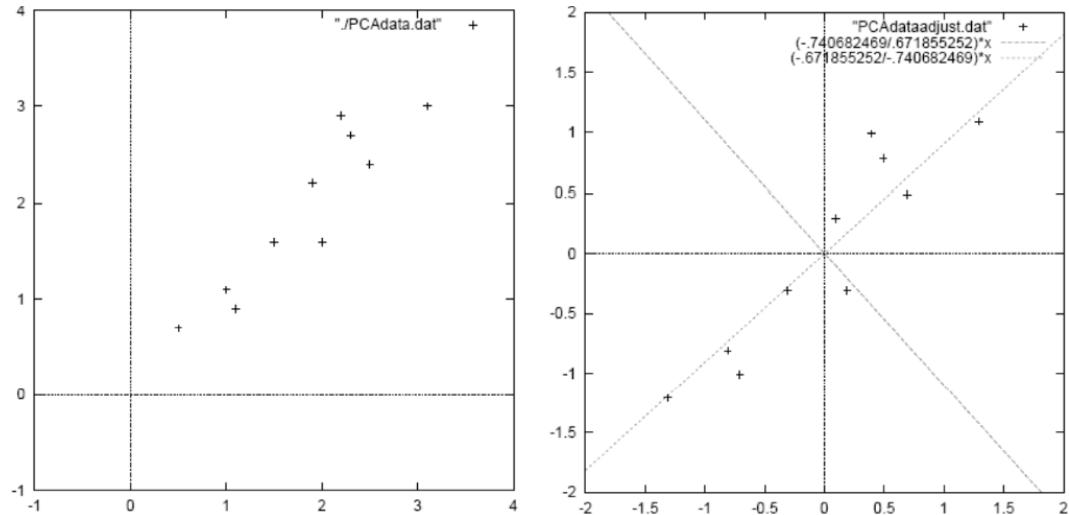
NOTE Since E_k is linear combination of the features it can be interpreted as a **rotation matrix**. With linear transformation the only thing we can do is **rotate the feature space, that is equivalent to make a combination of the original features** (think about a R^3 space that rotate, we do not have no more (x,y,z) but each one of the three components is a linear combination of all three).

NOTE PCA is a linear transformation of the feature space since indeed is generated through a matrix multiplication (think about a R^3 space that rotate, we do not have no more (x,y,z) but each one of the three components is a linear combination of all three).

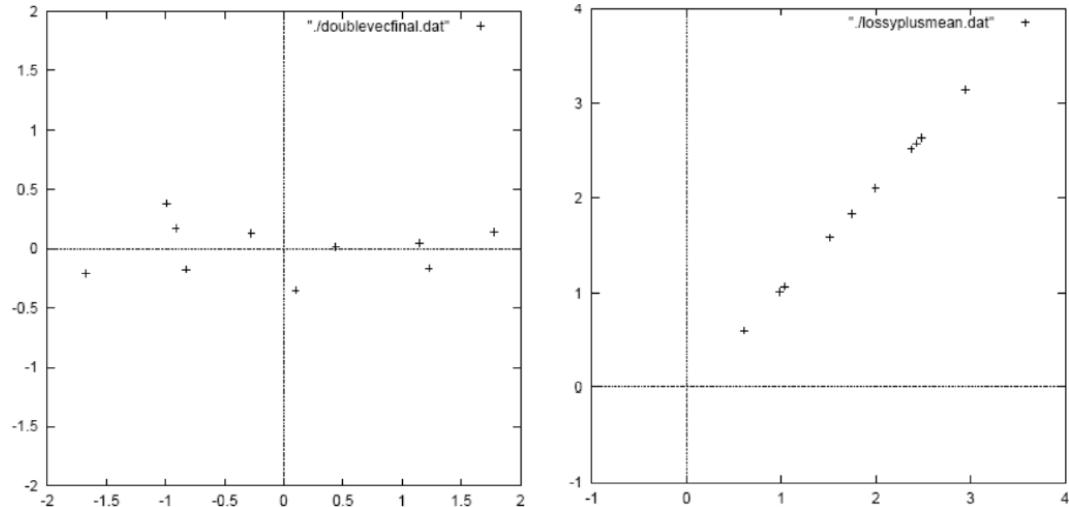
The PCA is a **lossy compression** of the feature space: we are compressing the information since we are **using less features** but we have a certain **percentage of information that is lost, proportional to the lost variance percentage** ($100 - \frac{\sum_{j=1}^k \lambda_j}{\sum_i \lambda_i}$). The consequence depends on how much the information lost is relevant for our task, but unfortunately we have no clue on how much is the impact: if we are lucky and the information lost are not important we lose as much we have estimated in performance (e.g. 10% of lost variance 10% of performance loss), otherwise we may have a more significant lost of performance than predicted (e.g. 10% of lost variance more than 10% of performance loss). The fact that we have no clue about the importance of the lost information is **due to the fact** that PCA is an **unsupervised approach so it is not related to the task that we are solving (encoded by the target value) but is a more general approach**. PCA is usually used since the assumption that retaining most of the variance the performance will be similar, is usually true but we have no guarantees.

NOTE Again, k is another hyper-parameter we may want to properly choose using cross validation or an adjustment technique.

Example - PCA application In the following pictures an example of PCA application:



Left: Original data, Right: Mean centered data with PCs (i.e. eigenvectors) overlayed



Left: Original data projected into full PC space, Right: Original data reconstructed with first PC

In the bottom right figure we can notice how data has been reconstructed from the new feature space with less dimension we have found using PCA: the data points are the same number of the original dataset and they have been projected over the largest PC that is indeed the feature we had selected, but comparing the reconstructed dataset with the original one we can notice that some information was lost, in particular we lost the variance over the PC we discarded.

Example - Face Recognition A typical image of size 256×128 is described by $n = 256 \times 128 = 32768$ dimensions. Each face lies somewhere in this high dimensional space, it is a point in this space. The space of images is so large that the space of faces covers a small sub-manifold of all the possible images. So, picking an image at random the probability of getting a face is almost 0, so we should be able to find a **better representation space for faces, a much low dimensional space, for example using PCA**. The idea is to take the dataset of 128 carefully aligned faces. We want to use the 15 dimension found through PCA, i.e. 15 eigenvector each one representing how the value of the pixel must be combined to obtain a feature, so each eigenvector can be shown as an image:

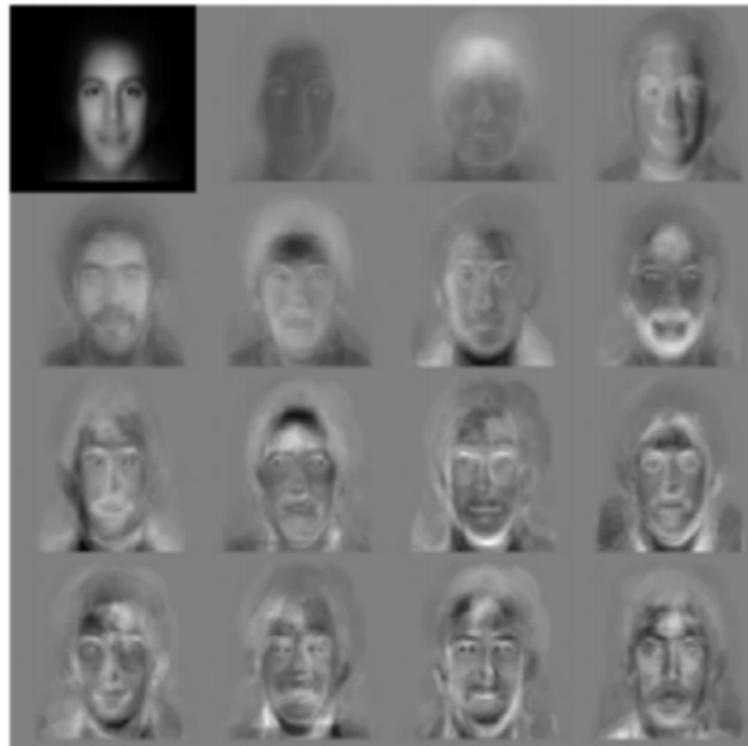


Figure 25: Eigenfaces: the top left face is the combination (sum) of the 15 eigenfaces.

due to the fact that the images are face-like, eigenvectors are called **eigenfaces**. This eigenfaces if combined in a linear way are good to reconstruct the 128 faces in the dataset. The **space obtained using PCA is much smaller than the original one and is good for the representation of faces**.

PCA is very useful pre-processing step indeed it:

- reduce computational complexity, due to the selection of a smaller set of features in which I can work better. And k is the hyper-parameter that tuned help us to find the best model complexity.

- can help in supervised learning, since reducing the number of dimension leads to smaller hypotheses space resulting in models that are less prone to overfitting.
- can be seen as a noise reduction method since it selects only the features components which have more variance.

This technique have some **defects**,

- Fails when data distribution is far from Gaussian, for example when data consists of multiple cluster in different region, so the co-variance matrix doesn't capture well the distribution of data in the space (you need more than second statistical order to describe data).
- Directions of greatest variance may not be most informative.
- Computational problems with many dimensions (SVD i.e. singular value decomposition approach can help).
- The main limitation is that PCA computes linear combination of features, but data often lies on a non-linear manifold, hence we would want to use non-linear combination features to obtain a better representation of data.

5.4 Model Ensembles

The methods seen so far can reduce bias by increasing variance or vice versa. But there's a class of technique that can reduce variance or bias, without increasing the other too much.

- To decrease the variance without increasing the bias we can use bagging.
- To decrease the bias without increasing the variance we can use boosting.

Bagging and Boosting are **meta-algorithms**, in the sense that they are applied to other algorithms (e.g. bagging/boosting + linear models/decision trees/neural networks). Their basic idea is: instead of learning one model, learn several and combine them. Typically they greatly improve accuracy.

5.4.1 Bagging

As we have said before bagging reduces the variance. The best way to reduce variance of a model with a lot of variance is averaging multiple models together. We know that, having a random variable X with variance $\text{Var}(X)$, and taking many realization of the random variable X and we average them obtaining \bar{X} , the variance of \bar{X} is simply obtained by taking:

$$\text{Var}[\bar{X}] = \frac{\text{Var}[X]}{N} \quad (50)$$

hence I have reduced the variance by taking the average over multiple realization of the random variable. So this can be **used for models with a lot of variance**, i.e. **model with a very large hypotheses space** (many features in a linear model, very deep decision trees) so when you are prone to overfitting. Hence, **it works with models that are overfitting and the overfit is reduced by building many models to reduce the variance**.

There is a problem: the above formula is true if the different model that I am averaging in \bar{X} are independent, but we have only one training set. In order to be able to apply this method we need to find a way to generate multiple models from one dataset. Where do multiple models come from, having one training set? To solve this problem we use the **bootstrap aggregation technique**. This statistical technique is able starting from a dataset of N samples, to produce B dataset each one with N samples. It generates this dataset using **random sampling with replacement**, so data is randomly sampled with the **possibility of sampling the same example multiple times (even in the same dataset)**, since each time there is a sample the whole original dataset is considered (drawn examples are replaced). These datasets are **not completely independent since they come from the same original dataset**, but they are somehow different since this technique will produce for each new dataset different distribution from the original distribution. In particular, it consists generating B bootstrap samples (i.e. new datasets) of size N from an initial dataset of size N by randomly drawing with replacement N observations (random sampling with replacement).



After bootstrapping we **train a model for each bootstrap sample**. These **models will not be completely independent but are somehow correlated**, so the **formula would give a variance that is a little bit lower than the real one according to how much the bootstrap samples are correlated**. But at the end we have the **desired effect of reducing the variance**.

In the prediction phase, in case of **classification** the result will be the **majority vote** on the classification results of every model, and for **regression** will be the **average** of the predicted values estimated by every model.

The advantages of bagging are that:

- **reduces variation.** In practice we want to **average multiple overfitting model**, we do it on purpose since from what we have studied before, an **overfitting model has low bias and high variance**. By combining different model we **reduce the variance maintaining the low bias**. Hence, **since bagging do not reduce bias it is important that each model has the bias as low as possible, i.e. we need overfitting models**, and perfectly works since bagging reduce instead the variance.
- **improves performance for unstable (i.e. overfitting) learners which vary significantly with small changes in the data set.** We have very different models even if trained on bagging samples, that comes from the same dataset, but **then the averaged prediction stabilize (i.e. reduce the variance) the final prediction**.
- **works particularly well with decision trees** and they are called **random forest**. May be a good idea to use as first method bagging on decision trees to have an idea if the problem is solvable or not, because they require a small number of hyper-parameters and so few tuning. You can look at the performance in validation of this approaches. If the performances are not far from the target probably you can hope that using other techniques and trying different hypotheses space you can achieve your target. So they are a good tool to understand the difficulty of the problem.

In practice bagging almost always helps. Is important to notice that **Is able not to increasing the bias since we are averaging a models and averaging does reduce variance but not bias**. Of course you are also **combining the bias of the models but if is low for each model the ensemble will have really low bias**. This improvement comes at the **cost of the necessity of training multiple models**, with the **limitation of having only one dataset that implies the correlation between the models and the larger the number of models the more is the correlation between them**. So the **reduction of variance is not linear in the number of models, the more the models you consider the less is the reduction of variance due to the fact that the linear reduction would be possible if the models were independent, but the increasing correlation imply a less effective reduction of the variance than the one expected from the formula**.

NOTE Using underfitting models with bagging does not works since, they have low variance and high bias, but bagging reduce only variance, so we must use on purpose overfitting models.

NOTE The different model can either have the same hypotheses space (usually done) but also different hypothesis spaces.

NOTE Bagging can be applied to **any** model that can overfit the data.

NOTE Higher the number of models we use lower is the variance but also the bias increases a bit since we have to consider the bias of all the models.

Summarizing, the MSE is the sum of noise, squared bias and variance, and bagging decreases variance due to averaging. This method typically helps when applied to an overfitted base model highly dependent on the training data. It does **not help much when there is high bias**, i.e. the model is underfitting and so is robust to change in the training data, **since usually high bias is associated with low variance, so it does not make sense to use bagging since it lowers the variance and not the bias**.

5.4.2 Boosting

Boosting is another (**very different**) technique of generating an ensemble of models. Its **aim** is to **reduce the bias**. It achieves so by **sequentially train weak learners**¹⁴, i.e. model with a lot of bias, hence an **underfitting** model, since this type of models have a **low variance**, indeed similarly to bagging, boosting is able to **reduce the bias but the variance of the models remains**, so for this reason we need models with the lower variance possible.

A weak learner has a performance that on any train set is slightly better than chance prediction, i.e. better **than random** (if the performance are worse than random it does not work). Hence it must be consistently better than random, but **not too much better than random**, since we want to avoid the risk of having too much variance.

NOTE Boosting is the answer of a theoretical question: **is it possible to combine many weak learner to produce a strong learner?** i.e. combining models with low accuracy to obtain one with high accuracy? Yes, if the weak learners are better than random choice for any distribution: **with boosting many weak classifier/regressor turn into a strong classifier/regressor**.

As said models are **trained sequentially**, which **means** that we **train a model based on the prediction of the previous**. The steps to perform boosting are the following,

1. Give an **equal weight to all training samples**
2. Train a **weak model on the training set**
3. Compute the **error of the model on the training set**
4. For each **training sample increase its weight if the model predicted wrong that sample**. Doing so we **obtain a new training set**.
5. **Iterate**: the training on the new training set, error computation and samples re-weighting **until we are satisfied by the result**.

¹⁴A weak learner is a learner that has a slightly better performance than chance prediction on any training set.

So at each iteration we are trying to train a model that performs well where the previous model was performing poorly. The final prediction is the weighted prediction of every weak learner. In practice we are combining a set of sequential underfitting model. Doing so, we have low variance and the bias is improved by combining the weak learner to form a strong learner. On average, boosting helps more than bagging, but it is also more common for boosting to hurt performance.

Let's compare boosting with bagging:

- Bagging is an ensemble of overfitting models, trained on different replicates of the dataset, hence the models can be trained in parallel due to the fact that the dataset are computed at the beginning.
- Boosting is an ensemble of underfitting models, trained on different dataset, again, but the model are trained sequentially due to the fact that the dataset for each model is computed using the prediction of the previous trained model, to know where it was weak. The training cannot happen in parallel but needs to be sequential.

Example - Boosting for Classification AdaBoost So at the beginning you have your training set, the learner, i.e. the loss function defined over the training set and the weights over the examples that are initialized all equal (uniform $w_i = 1/N$). Then the algorithm is applied for a certain number of rounds, that is the number of models we want to build ($r = 1 \dots T$). Note that p_r are the normalized weights of the r -th model, that represent how much each example is important in the training. From the picture we could see how we are using weak learners: the linear model with linear features is not able to classify correctly the dataset i.e. the dataset is not linearly separable. After the error is computed there is a check that could terminate the algorithm if the error is over (ζ) than the one of a random guesser (0.5), i.e. the algorithm is worse than the random selection. This situation is not desirable and should never happen, indeed if the algorithm stops there you have no guarantees of what the model could obtain. Hence the learner error should always be, at each iteration, less than 50% i.e. better than a random guesser. If the algorithm proceeds without stopping at the latter check (i.e. if the learner is better than random) then the coefficient β_r is computed. Since ϵ_r goes from 0 to 0.5 (for the latter check) then β_r have values between 0 and 1, and this identify the situation where the model is close to overfitting ($\beta_r \sim 0$) and where the model is close to random guessing ($\beta_r \sim 1$). Then the weights are updated, following the idea that if a sample was miss-classified the exponent of β_r is zero , hence the value of the weight does not change, instead if the sample is correctly classified then the old weight is multiplied for β_r , that being between 0 and 1 imply a reduction of the weights. Hence since the weights are reduced (changed) only for correctly classified samples, then at the normalization step, the importance given to the miss-classified samples is grater than the one given to the correctly classified once. This is repeated T times.

The reduction of the error is clearly not monotone, i.e. the loss is not reduced at each iteration, but it is guaranteed that having a weak learner in the long run (if the model are not overfitting, i.e. if the variance is not increasing) we will have a loss function that will decrease always more. The algorithm could be stopped when the improvement is no more significant. Then the final prediction is taken as the maximum taken vote weighting the answers: **the prediction is done considering the target whose prediction give the maximum value of the summation of the $\log(1 - \frac{1}{\beta_r})$ over the models that predict that value.** The weight of the prediction is function of β_r (that is function of ϵ_r) whose value is close to 0 if the error was close to 0 and is close to 1 when the error is close to the one of random guesser (i.e. very large). Using this kind of weighting factor we obtain the prediction. Hence **the votes (for a target) are weighted by the corresponding model performances during training, encoded in β_r associated to the error ϵ_r** , in particular: if $\epsilon_r = 0$ $\beta_r = 0$ so the weights $\log(1/\beta_r)$ become infinite (indeed this should never happen since it means that the dataset is being overfitted, but we are combining simple models) on the other end $\epsilon_r \sim 0.5\beta_r = 1$ so the $\log(1/\beta_r) = 0$ hence **the closer is the performance to random guessing the closer to 0 is the weight of the model in the boosting procedure.**

Finally, using many weak learners we were able to produce a stronger predictor, able to isolate well red points from blue ones.

Hence the boosting technique is able to reduce the bias of the linear regression in this feature space, without increasing too much the variance. **If the models are not weak models but are a little bit more powerful the risk is that increasing the number of models we combine the error do not decrease but at a certain point it start increase because you are accumulating also the variance**, so is no more true that adding models is beneficial for the problem.

NOTE The training for bagging is parallelizable but the training of each model is complex since they are overfitting models. On the other hand, boosting requires the training of simple models, but this must be done sequentially.

NOTE When we are considering the ensemble techniques we have to consider not only the computational effort in terms of complexity but also in terms of memory, since it must contain all the models that are used in bagging and boosting.

NOTE Bagging vs Boosting:

- Bagging reduces variance; boosting reduces bias.
- Bagging doesn't work so well with **stable models** (i.e. underfitting). **Boosting might still help.**
- **Boosting might hurt performance on noisy datasets.** Bagging doesn't have this problem.

- In practice **bagging almost always helps**.
- On average, boosting helps more than bagging, but is also more common for boosting to hurt performance.
- The weights grow exponentially
- Bagging is easier to parallelize.

Theorem[section] Corollary[theorem] Definition[section]

6 PAC-Learning and VC-Dimension

In this section we will have a look to some **statistical learning method**.

Previously we focused our attention on estimating the generalization error of a given model in order to measure the true performance. The training error usually is not a good indicator of how a model is behaving, but is easier way to estimate than other error terms. So we would like to **extract as many information as possible from training error**. Also there are **cases where the training error is somewhat a good estimation of the performance**. For example when we have a good number of samples relative to our hypotheses space we are less prone to overfitting (indeed the training error does not consider the variance that is low due to the size of the dataset). A question arises naturally. **Can we estimate how many samples are necessary given an hypotheses space?** We can answer this question in a theoretical way. Due to the fact that the **answer is theoretical, usually the number that comes out from the formula is very large and unpractical**. But at least we have an answer that tells us how many samples we would need if we want a certain accuracy with a certain number of complexity.

Remember Overfitting happens when the learner doesn't see "enough" examples, balanced on its complexity.

The **main problem with supervised learning** is to **choose the proper model** selection, since we want to avoid overfitting and underfitting we want to find out the proper **balance between bias and variance**. As we have seen the training error is a sort of measure of the bias of the model, indeed higher the model complexity lower the bias and the training error. On the other hand the training error does not give any information about the variance of the model, so to estimate the variance we have seen the two main techniques: cross validation (i.e. using different samples to evaluate the test error) that implies lower samples for training and higher computational complexity due to the fact that a lot of models must be trained; computing adjustment values that add to the train error a quantity that accounts for the number of parameter (i.e. the complexity) of the model. Here we will

discuss another approach that can be seen as an adjustment technique that tries to theoretically upper bound the variance of the model. This is useful since allow us to have some insights on which are the elements that have an impact on the variance of our model, simply computing the training error.

Hence the two question we are asking are:

- Overfitting happens because training error is bad estimate of the generalization error. **Can we infer something about generalization error from the training error?**
- Overfitting happens when the learner does not see "enough" examples. **Can we estimate how many examples are enough? How many samples are needed to reduce the variance?**

6.1 PAC-Learning

To introduce the ingredients of the theoretical setting we use a character recognition task, i.e. a **classification setting**.

Given an array of n bits encoding a binary-valued image we have the usual setting

- **X Instances set.** In the character recognition problem, the instance space is $X = \{0, 1\}^n$. The set of all possible input binary images.
- **H Hypotheses space.** The space where lies all possible combination of parameters.
- **C Set of target concept.** A concept is a subset $c \subset X$. One concept is the set of all patterns of bits in $X = \{0, 1\}^n$ that encode a picture of the letter "P". Hence the **target concept are the possible true functions** (e.g. the one that identify a "P" character) and for this reason the concept are **unknown**.
- **\mathcal{P} Probability distribution over X (unknown).** Training instances are generated by a fixed, unknown probability distribution over the input space X .

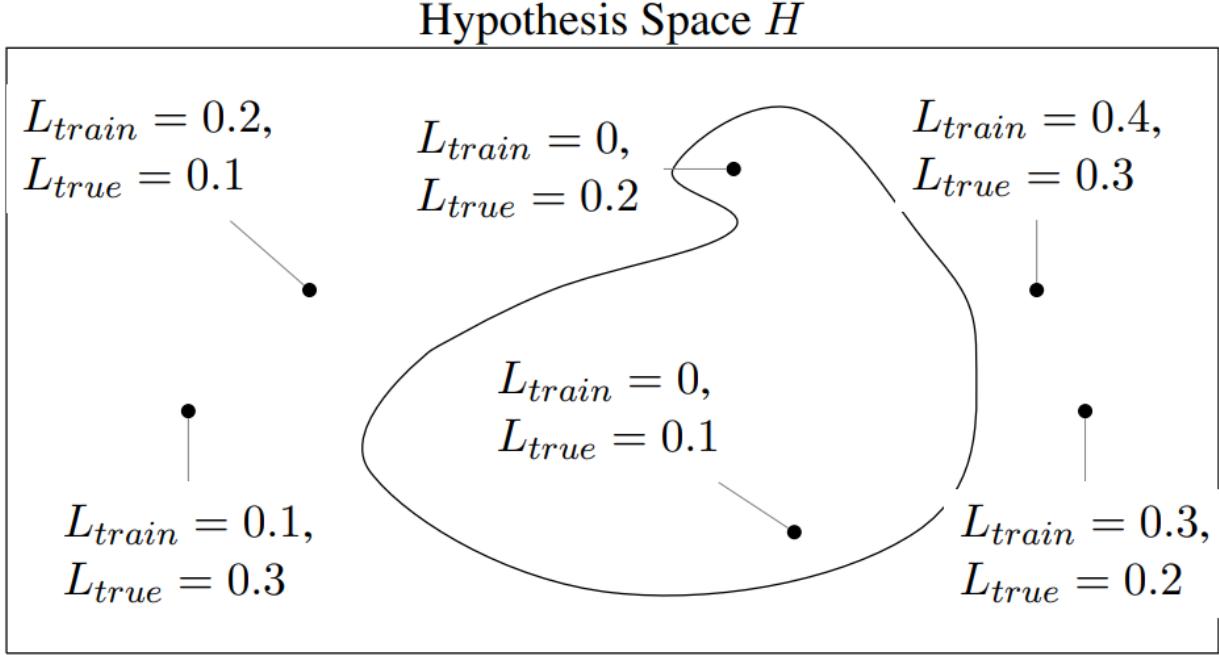
The learner observes a sequence \mathcal{D} of training examples $\langle x, c(x) \rangle$ for some target concept $c \in C$ and x is drawn from \mathcal{P} , where the teacher provides **deterministic** target value $c(x)$ for each instance. The learner must output a hypothesis h estimating c . h is evaluated by its performance on subsequent instances drawn according to \mathcal{P}

$$L_{true} = P_{x \in \mathcal{P}}[c(x) \neq h(x)]$$

that is minimizing the true loss function L_{true} , the one we could obtain knowing the concept c , that is the probability of having a miss-classification where the input x are drawn from the distribution P . In other words is the **probability of our hypotheses space h is different from the true concept c** . As we know this loss function cannot be computed since it is computed over the whole input space knowing the distribution P that is unknown so, our

objective is to bound L_{true} given L_{train} i.e. bound the generalization performances using the information we get from training.

Now we introduce the so called **version space** $VS_{H,D}$. It is a subset of H where the training error L_{train} is zero (0). So the hypothesis h are always correct on training instances.



For now, we **assume that VS is non-empty** (with some hypotheses space it could be empty, in particular if the true function is not contained in the hypotheses space). So the **setting we consider now is that given an hypothesis that is perfect on the training set, how bad can be when is tested on future examples** (i.e. generalization capabilities). Making some consideration we can **bound L_{train} to L_{true} inside VS**.

Theorem 6.1 (L_{train} bound in VS). *If the hypotheses space H is finite and \mathcal{D} is a sequence of $N \geq 1$ independent random examples of some target concept c , then for any $0 \leq \epsilon \leq 1$, the probability that $VS_{H,\mathcal{D}}$ contains a hypothesis error greater than ϵ is less than $|H|e^{-\epsilon N}$:*

$$P(\exists h \in H : L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \leq |H|e^{-\epsilon N}$$

Where ϵ is the error threshold that we want to use to bound the probability of the error (miss-classification). The fact that $L_{train}(h) = 0$ tells that it belongs to VS, instead $L_{true} \geq \epsilon$ is quantifying the probability of a bad event, since is the situation where the error is greater or equal to the threshold considered. Hence the probability of a bad event is the VS is upper bounded (cannot exceed) by $|H|e^{-\epsilon N}$. If the latter value increases is bad since then the value of having a bad event becomes higher, and in particular:

- It increases if the cardinality $|H|$ of the hypotheses space is high. This is a not desirable behaviour.
- (as expected) If the number of examples you observe in the training set is large and you have zero error in the training error it means that the probability of making large error in prediction would be low. This is a desired behaviour. Hence it decreases for bigger N .
- Larger the ϵ the smaller the probability of a event to be bad. Furthermore larger the ϵ worse is the bad event.

This result so shows the influence of the three main components: the size of the hypotheses space $|H|$, the number of examples N and the threshold ϵ on the prediction error that I want to consider. **The badness (how bad is the error) of the behaviour is encoded in ϵ .**

NOTE The theorem tells us **how likely is the learner to pick a bad hypothesis (***)**.

NOTE The theorem is evaluating the generalization capabilities of the learned model, indeed it consider the probability of existence of a hypothesis point in the VS that on the true function (L_{true}) performs poorly, hence the term $L_{true} \geq \epsilon$ is telling us that is a new sample.

Proof.

$$\begin{aligned}
& P \left\{ (L_{train}(h_1) = 0 \wedge L_{true}(h_1) \geq \epsilon) \vee \dots \vee (L_{train}(h_{|H|}) = 0 \wedge L_{true}(h_{|H|}) \geq \epsilon) \right\} \\
& \leq \sum_{h \in H} P(L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \quad (1) \text{ Union bound} \\
& \leq \sum_{h \in H} P(L_{train}(h) = 0 | L_{true}(h) \geq \epsilon) \quad (2) \text{ Bound using Bayes' rule} \\
& \leq \sum_{h \in H_{bad}} (1 - \epsilon)^N \quad (3) \text{ Bound on individual } h_i \text{s} \\
& \leq |H|(1 - \epsilon)^N \quad (4) |H|_{bad} \leq |H| \\
& \leq |H|e^{-\epsilon N} \quad (5) (1 - \epsilon \leq e^{-\epsilon}, \text{ for } 0 \leq \epsilon \leq 1)
\end{aligned}$$

□

NOTE At the beginning the existence quantifier can be restated in a disjunction of conjunction (... \wedge ...) \vee (... \wedge ...).

NOTE (1) For the union bound $P(A \vee B \vee C) \leq P(A) + P(B) + P(C)$, since we are summing twice the intersections the \leq is introduced.

NOTE (2) Applying the Bayes rule: $P(A|B) = \frac{P(A \wedge B)}{P(B)} \implies P(A|B)P(B) = P(A \wedge B)$ $\implies P(A|B) > P(A|B)P(B) = P(A \wedge B) \implies P(A|B) \geq P(A \wedge B)$ since $P(B) < 1$, in particular $B = L_{true} \geq \epsilon$, so in the proof we are already upperbounding (\leq) and we can discard $P(B)$.

NOTE (3) After the application of the Bayes rule we know that at least the error is ϵ (since is a given information, the event under which we are conditioning the probability $\geq \epsilon$). Since the conditioning event is that $L_{true} \geq \epsilon$, the error of having an example without mistakes with a model that has an error rate that is at least ϵ , is at least $(1 - \epsilon)^N$. Indeed we are restricting the summation only to the bad hypothesis, so the probability of having a train error equal to 0 is equal to the probability of having N examples without mistakes with a model that at least have an error rate of ϵ is $(1 - \epsilon)^N$, indeed: $(1 - \epsilon)$ is the upperbound of a correct classification by a model that have an error lowerbound of ϵ (e.g. $\epsilon = 0.1, L_{true} \geq \epsilon \implies p(correct) \leq 0.9$), and the probability of being correct for N samples will be $(1 - \epsilon)$ multiplied N times since the samples are independent. Notice that the summation is done over the bad part of the hypotheses space, since the probability was conditioned by a bad event ($L_{true} \geq \epsilon$).

NOTE (4) Since we do not know how many bad hypothesis we have we need to upperbound that with the total number of hypothesis.

NOTE (5) This is only another way to express the formula at (4), used to have a comparison with other theorems. This step is obtained comparing the line $1 - \epsilon$ with the function $\exp^{-\epsilon}$ between 0 and 1.

We can notice that the **dimension of the hypotheses space influences negatively the bound**, in fact a **larger** searching space will give us **less guarantees on the value of L_{true}** since the probability has an higher upper bound. On the other hand, **having more samples is always better**, in fact $e^{-\epsilon N}$ is monotonically decreasing with N . **Larger ϵ will lead to smaller bound because we are less demanding on the similarity between L_{true} and L_{train}** , i.e. we are considering a larger error threshold, which means that we allow more bad event to happen, so obviously is easier to find an hypothesis that satisfy this bound.

Now we can **bound the probability** of $P(\exists h \in H : L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \leq |H|e^{-\epsilon N}$ using a **Probably Approximately Correct (PAC) bound**. We can set a parameter δ

$$|H|e^{-\epsilon N} \leq \delta$$

which means that the confidence we want a bad event to happen is upper bounded by delta i.e. we are saying that we want the probability of a bad event to happen for an hypothesis inside the VS of N samples bounded by δ . After choosing δ the inequality can be solved to find N or ϵ .

Given ϵ (the accuracy you want to have for the model) and δ (the probability

you want the result to hold) you can **know** the **minimum number of examples** that gives you this guarantees

$$N \geq \frac{1}{\epsilon} \left(\ln(|H|) + \ln\left(\frac{1}{\delta}\right) \right) \quad (51)$$

that is a **lower bound of the number of samples** N to have a low probability to have a bad hypothesis. Of course, **smaller** the ϵ , i.e. model less prone to error, larger the size of the dataset needed. Furthermore it depends on the **logarithm of H** and the one of the inverse of δ . Since δ is the **confidence** that I want to have, if I want to be pretty sure of satisfying the ϵ error I should have a very small value of δ but this imply **higher number of samples**. So this is the **theoretical answer** to the question how many samples do I need. Instead given N (the number of samples) and δ the accuracy ϵ could be computed

$$\epsilon \geq \frac{1}{N} \left(\ln(|H|) + \ln\left(\frac{1}{\delta}\right) \right) \quad (52)$$

which tell us that the **error reduces with more samples** N but **increases with for larger hypotheses spaces and for lower delta the error may be greater since we are restricting the term $|H| \exp^{-\epsilon N}$ to lower values**, which means that the power of the exponential should be greater allowing the greater confidence level given by δ . In other words smaller delta implies that the probability of having a bad event is more strict (lower), but this obviously forces the badness of the error to increase, indeed the confidence comes at a cost of worse problem.

NOTE The term **Probably** in PAC is represented by δ , so it represent the **confidence** we want to have on the result, the upper bound of the failure probability; instead the term **Approximately** is represented by ϵ i.e. **the approximation that I want to the correctness of our method**, i.e. the **accuracy**.

NOTE δ represent a confidence, since we are in a PAC framework so we **give guarantees in probability** (e.g. I give you a model that has 99% probability of having the error lower 1% with a certain size of hypotheses space).

NOTE Larger the size of the hypotheses space larger is the number of samples required to obtain the same accuracy (ϵ), but this is predictable since increasing the size of the hypotheses space we increase the variance and to reduce the variance we need more samples.

NOTE $|H|$ can be very large and grows very fast (even faster than exponential). If we take as an example a binary decision problem with M binary inputs (features), the size of H i.e. the number of functions you can build will be 2^{2^M} . So N lower bound will have a exponential relationship with M ($\ln 2^{2^M} \propto \exp^M$). This is **related to the curse of dimensionality**.

NOTE The formulas are valid for binary classification, but they can be easily adapted.

NOTE The formula derives from an upper bound $|H| \exp^{-\epsilon N}$ and it can be very loose as a formula. By using this formula we are trying to estimate the variance whose bound can be very loose, so usually using this we end up choosing overly simple models, since we are overestimate the variance a lot. Usually more practical method such as cross validation are used, but theoretical formulas are useful to put in formula what we are talking about from the beginning: analytically measure the effect of having a large hypotheses space.

Example Suppose H contains conjunctions of constraints on up to M boolean attributes (i.e., M literals). For example with conjunction of three $|H| = 3^M$. How many examples are sufficient to ensure with probability at least $(1 - \delta)$ that every h in $VS_{H,D}$ satisfies $L_{true}(h) \leq \epsilon$?

$$N \geq \frac{1}{\epsilon} \left(M \ln(3) + \ln\left(\frac{1}{\delta}\right) \right)$$

so the number of samples is linear in the number of features and not exponential, and this is due to the fact that the size of the hypotheses space was reduced a lot.

Now we are ready to define formally what PAC¹⁵learning is. Consider a class C of possible target concepts defined over a set of instances X of length n , and a learner L using hypotheses space H .

Definition 6.1. C is **PAC-learnable** if there exists an algorithm L such that for every $f \in C$, for any distribution \mathcal{P} , for any ϵ such that $0 \leq \epsilon < \frac{1}{2}$, and δ such that $0 \leq \delta < \frac{1}{2}$, then algorithm L , with probability at least $(1 - \delta)$, outputs a concept h such that $L_{true}(h) \leq \epsilon$ using a number of samples that is polynomial of $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$

Definition 6.2. C is **efficiently PAC-learnable** by L using $H \iff \forall c \in C$, distributions \mathcal{P} over X , ϵ such that $0 \leq \epsilon < \frac{1}{2}$, and δ such that $0 \leq \delta < \frac{1}{2}$, algorithm L , with probability at least $(1 - \delta)$, outputs a concept h such that $L_{true}(h) \leq \epsilon$, in **time** that is polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, M and $\text{size}(c)$ ¹⁶

NOTE Hence the PAC-learnability is based on the fact of the main terms of the inequality from the theorem being polynomial.

Now we need to **generalize to problems where the train error is not equal to**

¹⁵Probably Approximately Correct learning. Probably refers to δ and it is the confidence. Approximately refers to ϵ and it is the accuracy

¹⁶ $\text{size}(c)$: Number of bit necessary to encode the concept c . It comes from information theory.

zero (agnostic learning) **over the VS because usually this lead to empty VS** (unless hypotheses space is very large). Hence we have to deal with inconsistent hypothesis, i.e. hypotheses that have not a zero error over the training set. We can **simply bound the difference (gap)** between L_{train} and L_{true} in H .

$$L_{true}(h) - L_{train}(h) \leq \epsilon \implies L_{true}(h) \leq L_{train}(h) + \epsilon$$

NOTE We will see that the train error is related to the bias, instead the **gap error is related to the variance** since the true error obviously account for both the bias and the variance.

NOTE This **inequality was already used but before we considered VS with a 0 training error**.

From now on we will consider only binary classification problems for simplicity. As we did before, we need to find an upper bound for the probability of having a "bad event", which in this case consists in having a gap between L_{train} and L_{true} bigger than ϵ . To achieve this we use the **Hoeffding bound**, which states

Definition 6.3. For N i.i.d. coin flips X_1, \dots, X_N , where $X_i \in \{0, 1\}$ (i.e. N realization of random variables) and $0 < \epsilon < 1$, we define the empirical mean $\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$, obtaining the following bound

$$P(E[\bar{X}] - \bar{X} > \epsilon) \leq e^{-2N\epsilon^2}$$

Which means that the empirical average \bar{X} of the random values concentrates around the expected value of these random variables with a certain speed, indeed the probability that the difference is greater than ϵ is upperbounded by a term, i.e. the empirical average can be far from the expected value more than ϵ with a probability smaller than that quantity. When N goes to infinite the probability upperbound goes to 0, so the empirical average converge to the expected value (as we know, since the definition of expected value is the mean over infinite samples). Hence larger the number of samples, smaller the probability of the empirical average of being far from the expected value. Of course larger the ϵ smaller the probability, due to the dependency of the exponential to ϵ^2 .

Theorem 6.2. Given an hypotheses space H , a dataset \mathcal{D} with N i.i.d. samples, $0 < \epsilon < 1$: for any learned hypothesis h :

$$P(\exists h \in H | L_{true}(h) - L_{train}(h) > \epsilon) \leq |H|e^{-2N\epsilon^2}$$

This encodes the bound of probability of having a bad event and is very similar to what we have found in the non-empty case. Furthermore we consider $\delta = |H|e^{-2N\epsilon^2}$.

IMPORTANT What we are saying with this theorem is that chosen a δ we are bounding the probability of having an hypothesis inside the hypotheses space whose error is far from the training one (i.e. the one we use for the model performance analysis) more than ϵ . Since we are bounding the probability of a bad event that is equivalent to say that the probability of having a good hypothesis is $1 - \delta$.

Like we did before, we can calculate the number of examples needed given ϵ and δ

$$N \geq \frac{1}{2\epsilon^2} \left(\ln(|H|) + \ln\left(\frac{1}{\delta}\right) \right)$$

still very similar to the one we found before but the dependency over ϵ is inverse squared so lower the ϵ higher the minimum number of samples needed to reach the confidence δ . Or ϵ given N and δ .

$$\epsilon \geq \sqrt{\frac{\ln(|H|) + \ln\left(\frac{1}{\delta}\right)}{2N}}$$

Now we can rewrite the gap between L_{train} and L_{true} as

$$L_{true}(h) \leq L_{train}(h)_{Bias} + \epsilon \implies L_{true}(h) \leq \underbrace{L_{train}(h)}_{Bias} + \underbrace{\sqrt{\frac{\ln(|H|) + \ln\left(\frac{1}{\delta}\right)}{2N}}}_{Variance(\epsilon)} \quad (53)$$

so the true (generalization) error is bounded by the training error that encodes the bias plus a penalization term that encodes the variance and depends on the size of the hypotheses space, on the confidence you choose to achieve and of course reduces for a greater number of samples. Once more, we can see how $|H|$ influences the loss function, as we intuitively said before:

- For large $|H|$ we assume a low bias because it's more probable to find a good h and a high variance. Hence we are no more sure of what the train error is saying because the variance term increases. Of course this behaviour can be compensated with a large number of samples.
- For small $|H|$ we have high bias because we have a low probability of including a good h and low variance.

In practice what we are saying is that we have to justify a large H with a lot of data. If we do so the training error will be a good estimation of the overall performance (test/true error).

6.2 VC Dimension

So far we have considered only finite hypotheses space which means a finite set of hypothesis i.e. a finite number of weights values combinations. If we use the bound that we have just

found in an infinite¹⁷ H , we would mean infinite variance requiring infinite examples for each ϵ, δ combination. This is not the case, for infinite H the previous bound is too pessimistic. So we need to find a new one. What is really important when we want to measure the size of a hypotheses space? What is important to measure is the flexibility of the hypotheses space, indeed the number of hypothesis is not really significant, indeed having many hypothesis that are all similar (i.e. the model that they encode are similar) the effective size of the hypotheses space is small. What **I want to measure is the capacity of my hypothesis space to overfit data**, i.e. the number of points that can be classified exactly. Is possible to get a bound error as a function of the number of points that can be completely labeled, i.e. the capability of the H to overfit the data.

In the finite H case we encoded the complexity of H in the number of possible hypothesis. In the infinite case we can't do this, so we need to find a new metric to measure the complexity of H . We will use the **VC dimension**, that is a way to measure the complexity of the hypotheses space in terms of capability to overfit data. To lay the ground for our theoretical discussion, we need to introduce two definitions,

Definition 6.4 (Dichotomy). *A dichotomy of a set S is a partition of S into two disjoint subsets*

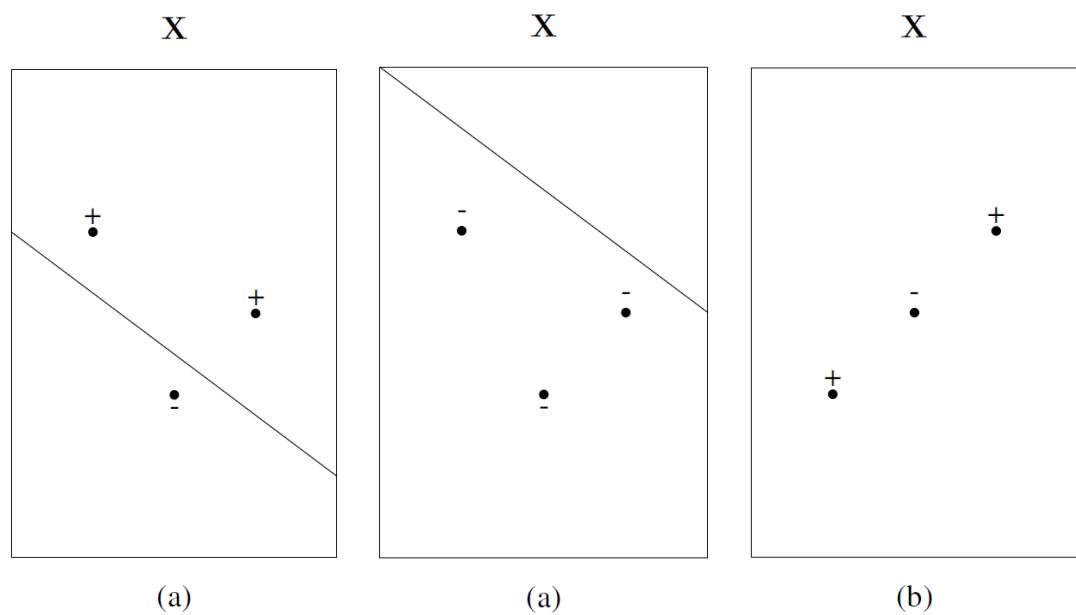
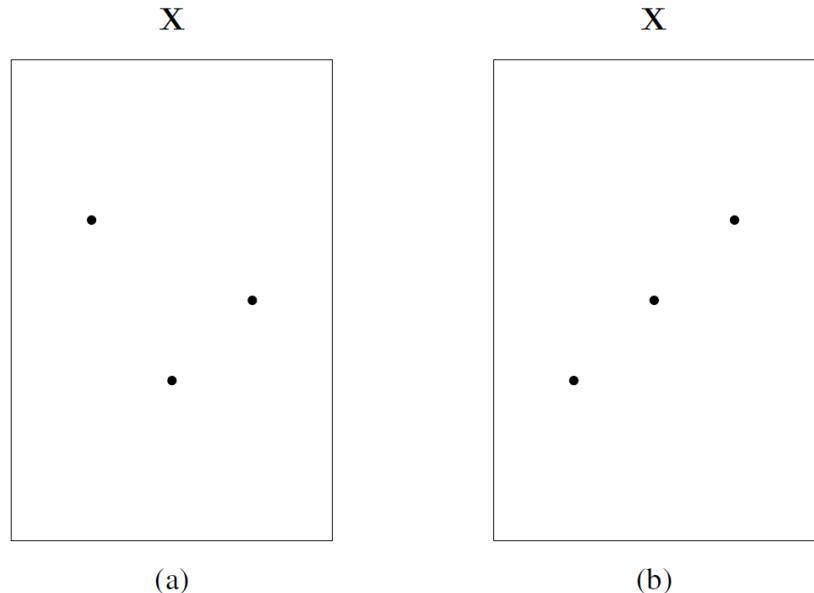
Definition 6.5 (Shattering). *A set of instances S is shattered by hypotheses space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy*

In practice shattering means that if we split our instances set in two not overlapping subset, H shatters S if and only if for every dichotomy, we can find an hypothesis which classify correctly every instances (i.e. $L = 0$). As a reminder we are still considering only binary classification problems, so the H shatters S if and only if for every dichotomy of S , i.e. independently of how you label them in positive or negative (i.e. assigning a label to the two disjoint subsets), we can find an hypothesis which classify correctly every sample to its subset of the dichotomy.

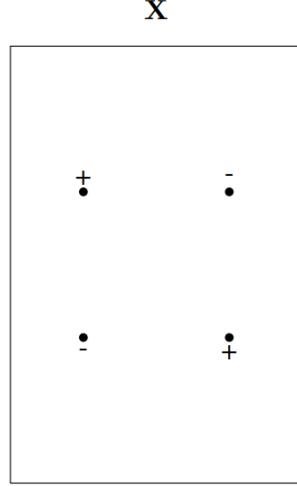
Example We consider an instances set with three example¹⁸ (i.e. three points in the input space) and an hypotheses space representing a linear classifier. In this case a dichotomy is a specific assignment of class (+) or class (-) to every point. We can have two cases, one where the examples are not aligned and one where they are. H shatters any of the two instances set? Yes, only (a) because we can always find a line that divides every possible dichotomy. In the specific dichotomy shown for (b), we can't find a linear classifier (line), that classifies correctly the three points, so H containing linear classifiers doesn't shatter (b).

¹⁷Infinite hypotheses spaces are very common. For example linear regression or classification have infinite hypotheses spaces

¹⁸This is still infinite because every instance can have a different "position" in the instances set



Note that for four instances, is not possible to find a way such that a linear model H can shutter S^{19} .



Now we can define what is the VC dimension

Definition 6.6 (VC dimension). *The Vapnik-Chervonenkis dimension, $VC(H)$, of hypotheses space H defined over instance space X , is the size of the largest finite subset of X shattered by H . If arbitrarily large finite sets of X can be shattered by H , then $VC(H) \equiv \infty$*

In our previous example, the VC dimension of linear classifiers in two dimension is three. Indeed, we can have a configuration of three instances whereby every dichotomy is perfectly separable. This doesn't hold for four instances, indeed does not exist a positioning of 4 points that can be shattered by a linear classifier H .

Example Few examples of VC dimensions,

- Linear classifier: $VC(H) = M + 1$, for M features plus the constant term. For the linear model the number of parameter of the model is equal to the size of the space.
- Neural networks: $VC(H) = \#\text{parameters}$ (for some kind of neural networks).
- 1-Nearest Neighbor: $VC(H) = \infty$, i.e. any dichotomy with any number of points can be shattered since it always predict the class of the closest training example to the queried point, so by definition on the training set it makes zero error since it predicts the values of the training set. Remember that in this method there is no training, it simply pick the class of the closest point in the training set to the query input.
- SVM with Gaussian Kernel: $VC(H) = \text{finite}$. In particular support vector machines have potentially infinite number of parameter but its VC dimension can be finite. We will see SVM in future chapters.

¹⁹This is the XOR problem. It's not linearly separable

For many algorithms the rule of thumb is that the number of parameter in the model often matches the maximum number of points that can be scattered, i.e. the VC dimension. But in general as we saw with last examples it can be completely different, indeed, in some cases the parameters can be infinite and the VC dimension is finite or can also happen that a problem has an hypotheses space with 1 parameter and infinite VC dimension (e.g. imagine the point aligned over the only dimension and binary classified, then exist a sinusoidal with the only parameter that changes the frequency that can classify any labeling with any number of points). Hence there exist some class of functions that can overfit very complex dataset in some input spaces.

NOTE If the VC dimension is ∞ you do not have many guarantees, indeed the method overfits a lot so the bound to the variance cannot be performed.

Now, considering that we were able to find the VC dimension of our model, we can find a new bound for the error between L_{train} and L_{true} , and so we can find how many randomly drawn examples suffice to guarantee an error of at most ϵ with probability at least $(1 - \delta)$

$$N \geq \frac{1}{\epsilon} \left(4 \log_2 \left(\frac{2}{\delta} \right) + 8 \text{VC}(H) \log_2 \left(\frac{13}{\epsilon} \right) \right) \quad (54)$$

that has the term $1/\epsilon$ as on the case of VS, the usual term that depends on the confidence δ and finally a term that instead of considering the complexity of the hypotheses space H with its cardinality ($\ln(|H|)$), it considers its VC dimension that does not go to infinity if the hypothesis are infinite but the value depends on the **flexibility of our hypotheses space to shatter instances in a given input space**.

Equally we can express this as an upper bound for L_{true}

$$L_{true}(h) \leq \underbrace{L_{train}(h)}_{Bias} + \sqrt{\underbrace{\frac{\text{VC}(H)(\ln(\frac{2N}{\text{VC}(H)}) + 1) + \ln(\frac{4}{\delta})}{N}}_{Variance}} \quad (55)$$

where again there is a variance reduction with respect to N and a dependency on the VC dimension of the hypotheses space H . So the same bias variance trade-off.

Is called **structural risk minimization** the choice of the hypotheses space H minimizing the above bound. This is like an **adjustment technique**, indeed we use the train error as the measure of the bias and the penalization term that measure the variance considering the VC dimension of the hypotheses space and the number of examples. Hence without doing cross validation you can use this function choosing the model complexity that minimizes the upperbound to the true error.

NOTE The algorithm works in the feature space, not in the input space

Properties of VC dimension The first one concerns the finite hypotheses space VC dimension

Theorem 6.3. *The VC dimension of a hypotheses space $|H| < \infty$ is bounded from above*

$$VC(H) \leq \log_2(|H|)$$

so in the last bound the VC dimension can be substituted with $\log_2(|H|)$.

Proof. If $VC(H) = d$ then there exists at least 2^d functions(combination) in H, since there are at least 2^d possible labelings

$$\begin{aligned} |H| &\geq 2^d \\ |H| &\geq 2^{VC(H)} \\ VC(H) &\leq \log_2(|H|) \end{aligned}$$

□

Furthermore, concerning the PAC learnability of a concept class

Theorem 6.4. *Concept class C with $VC(C) = 1$ is not PAC-learnable.*

indeed it means that if the VC dimension of the concept class C is ∞ you cannot learn the solution with a polynomial number of samples (i.e. PAC learnable) since the model is able to overfit any number of examples, so the number of examples that you need to have the theoretical guarantees about confidence is no more polynomial in the quantities we saw.

Remember The goal of the adjustment technique is to use only the training error to evaluate the performances of a model, without saving samples for validation or using cross validation. Hence they not only save time for the complexity choice but also samples that can be used all for training. The drawback of this techniques is that the **upperbound can be loose** due to the fact that the term that accounts for the variance can grow faster than the real variance, leading to the selection of simpler model (than the one we can choose) since other are more penalized than they should be.

NOTE The other adjustment techniques we saw comes from other assumption, instead this is more general since it works for a larger class of algorithm and hypotheses space.

NOTE The decision of the VC dimension of an hypotheses space can be imagined as a two player game: the first player starts deciding the number of examples and their position in the hypotheses space, and the second player has to find a dichotomy that prevents the first player to shatter. If the first player is able to shatter at least one set with N samples it means that the VC dimension is at least N , so it tries with $N + 1$ examples. If for $N + 1$ examples there is always a way for the second player to choose the dichotomy such that the

first player is not able to shatter it, the the VC dimension is N (e.g. $N + 1 = 4$ for linear classification in 2 input features). Pay attention that even if exist a dichotomy for which $N + 1$ examples can be shattered (e.g. $N + 1 = 4$ 3 vertex + and 1 -) the second player will choose a dichotomy that prevents the first player to shatter. Hence, the VC dimension is the maximum number of samples for which exists a disposition in the input space such that is not possible to find a dichotomy that cannot be shattered (e.g. for 3 examples in two input features, if the points are not aligned the linear classifier always shatter any dichotomy; furthermore with 4 examples always exist for each disposition a dichotomy that does not let the linear classifier to shatter the dichotomy).

NOTE For commonly used methods we have the result from people that studied the problem and found the VC dimension, that is not always easy to find. Once you have the VC dimension you can use the theoretica result to upperbound the variance of the model, knowing the number of training error and the confidence we want to achieve.

Theorem[section] Definition[section]

7 Kernel methods

Kernel methods is the dual of parametric methods. Parametric methods work in the feature space, while **kernel methods work in the sample space**.

Kernel methods is **a family of non-parametric techniques**. To better explain what it means, we start from what we have already seen in the previous chapters. With parametric method a certain hypothesis in the hypothesis space is defined by the combination of values of the learnable parameters. For example, linear regression is a parametric method, where each hypothesis is defined by the value of the parameters associated with each feature plus the constant term. **With non-parametric methods we have no explicit parameters**.

- In **parametric methods** the training set is used in the **training phase** to learn the parameters. Then for the prediction phase the training set is not used because all the relevant information are encoded in the learned model.
- In **non-parametric methods** the training set is used also in the prediction phase because the model is implicitly encoded in the dataset.

Kernel methods are **memory-based** (like nearest neighbour) i.e. they use data directly, combining the training data in some way they use them in the prediction phase. Hence this kind of methods are **fast to train, all the effort is moved on the prediction** (this is true for most of the non-parametric methods): during prediction you have to **search the samples in the memory and combine them to produce the output**. Due to this fact kernel methods need the definition of a metric of similarity among samples: while the **effort of the engineer in parametric methods is to find a good selection of features, here is substituted with the choice of the metric of similarity between samples**.

NOTE As we can see **for real time prediction parametric methods are in general preferable**, since all the knowledge from the training set is encoded in the parameters of the model. While for kernel methods you **cannot never really discard the training set since the prediction is computed over it**.

NOTE Depending on the problem may be easier to find a good feature selection or a good metric of similarity, there is not a general rule.

The **goal** we want to have, as before, is to **capture non-linear patterns of the data**

- Non-linear regression: input-output relationship may not be linear.
- Non-linear classification: classes may not be separable by a linear boundary.

since as we have seen we cannot work on models linear on the input variables but we have to work with models that are linear in the feature space that derives from the linear transformation of the input variables, indeed linear models are just not rich enough so we need to introduce non-linear features that encode relevant information for our model.

The **idea** of kernel methods is the same, by defining **kernels that measure the similarity among samples**. Using kernel functions is the same as mapping our data in a new feature space implicitly, without the need of computing the features. The result is that it is like we are working in a very complex feature space, being able to define a projection over spaces with infinite dimensions, being able to work in a space with infinite features but with a finite complexity (i.e. you can compute everything in a finite time). Hence kernels make linear models work in non-linear settings by mapping (i.e. changing the feature representation of) data to higher dimensions where it exhibits linear patterns and applying the linear model in the new representation. Kernels are used when you want to work in a very complex feature space, with many dimension, usually with a number of features higher than the number of samples. Indeed kernels can learn a high dimensional feature space without the need of never computing the feature space. Furthermore such mappings can be expensive to compute, but kernels give them for (almost) free (kernel trick). Hence again the objective is to work in a high dimensional feature space but we want to do this in a very efficient way.

Example - KNN (K-Nearest Neighbours) A very famous example of non-parametric method is the k-nearest neighbour²⁰. This method is used for classification. In practice when we have a new sample, we search for the k nearest training data samples in the training data. Then we assign a class to the new sample equal to the most frequent class between the k nearest training samples. Once classified, the new sample becomes part of the training data.

K-nearest neighbour doesn't utilize parameters, but introduce the concept of "distance" for evaluating the new samples. The distance, more formally, is called **metric**. As in parametric method we need to define the features, in non-parametric methods we need to define a metric. We can notice that in the k-nearest neighbour example we have no training time because we haven't a model. But this comes with a price, in fact we have a time penalty during the prediction phase because we need to review each training data to make our assumption, instead of just use our model.

- Parametric: long training time, short prediction time
- Non-parametric: short training time, long prediction time

So far, all the parametric methods were linear. So in the vanilla configuration they can

²⁰It's worth mentioning that k-nearest neighbour is not a kernel method. It's used only as an example for non-parametric methods

only solve linear problems. We have also seen how we can extend those linear models to non-linear problems, for both regression and classification through the introduction of the basis functions. In the following sections we will see how we can **extend the capability of the linear models to non-linear problems with the kernels**.

NOTE Higher dimension means high number of features.

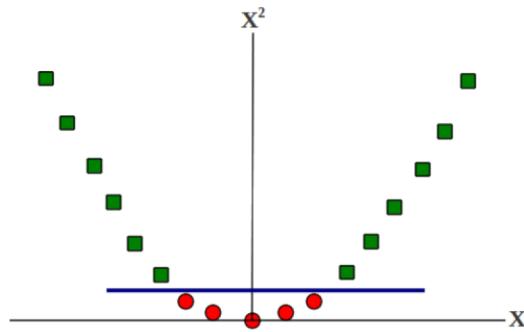
7.1 Kernels

Kernels make linear models work in non-linear settings, by mapping data to higher dimensions, where it may exhibits linear patterns and so linear models are applicable. Another good characteristic of kernel methods is their complexity. In fact, parametric methods complexity is based on the number of features, instead, **kernel methods complexity is based on the number of samples**. This gives us some advantages in some situation. For example, when we have **more features than samples**, kernel methods are much more efficient, both complexity and performance wise. A **mapping to higher dimensions can be very expensive to compute, but kernels can give such mapping almost for free**. This process is called **kernel trick**. In practice we can find a **dual representation of a linear model using kernels**.

Example - Linearize by dimensions augmentation Consider this binary classification problem

$$\text{---} \square \square \square \square \square \square \bullet \bullet \bullet \bullet \bullet \bullet \square \square \square \square \text{---} x$$

Each example is represented by a single feature x . It clearly doesn't exist a linear separator between the two classes. But if we map $\{x\} \rightarrow \{x, x^2\}$ each example now has two features and the data are linearly separable.



This was the standard approach for extending linear models to non-linear problems and as we have seen in general the problem of finding a proper feature space is much harder than this.

NOTE What we are trying to achieve in general is to project the points in a such high dimensional space that they become linearly separable. The problem to do this is that we may consider a large number of features with the problems that comes with it such as overfitting and also computationally speaking, since computing a large number of features can be very expensive and prohibitive.

Remember There is always a feature selection that makes the problem linearly separable (can be seen as there is a space where the VC dimension is ∞ so any dichotomy could be shattered) but then there is the problem of overfitting, so the model is so complex that does not generalize. The problem of overfitting exists due to the fact that we have only a finite number of data.

NOTE Now the focus is not on the overfitting aspect of high dimensional spaces, but on the computational efficiency to find those features.

7.1.1 Kernel functions

Consider the following mapping Φ for an example $x = \{x_1, \dots, x_M\}$

$$\Phi : x \rightarrow \{x_1^2, x_2^2, \dots, x_M^2, x_1x_2, x_2x_3, \dots, x_1x_M, \dots, x_{M-1}x_M\}$$

This particular mapping is called **second order monomial**, indeed each new feature uses a pair of the original features. We can observe that the **mapping will increase quadratically the number of features**, i.e. for each input added the number of features will increase quadratically in number. This will have an **impact on complexity** because computing the mapping itself can be inefficient. Moreover, **using the mapped representation could be inefficient too**.

NOTE The problem is not only overfitting but also computing them because it takes a lot of time since for each example of the dataset the input variables must be expanded into the features. Hence, days may be needed to transform the input without even starting training.

So the problem is that mapping usually leads to the number of features blow up, exponentially grow. Thankfully, kernels help avoid both these issues because the mapping doesn't have to be explicitly computed and the computations with the mapped features remain efficient with a low computational effort.

NOTE I want to move from feature space to sample space because in this way the complexity does not depend on the number of features but it depends on the number of samples, so working in a feature space that is much larger than the number of samples kernel methods are preferable.

Many linear parametric models can be re-cast into equivalent **dual representation**

where predictions are based on a kernel function evaluated at training points. A kernel is a function which takes as input two data samples, and performs the scalar product between the feature expansions (mapping) of the two samples.

$$k(x, x') = \Phi(x)^T \Phi(x') \quad (56)$$

This kernel function is the metric of our non-parametric method and it measure the similarity between the points x and x' of the dataset. A good consequence of being a scalar product is the symmetry ($k(x, x') = k(x', x)$). Looking at the definition of kernel function seems that we are not avoiding the computation of the feature vector as we said, but is possible to define kernel function that have a much faster way to be computed and that can be also represented as the scalar product of feature vectors. So the kernel functions must be computable as their definition says (i.e. by the scalar product of the feature vector) but you do not need to compute them in that way.

NOTE We say that it measure the similarity since the scalar product of the definition can be rewritten as:

$$k(x, x') = \Phi(x)^T \Phi(x') = \|\Phi(x)\| \cdot \|\Phi(x')\| \cdot \cos\theta$$

so the two features vector are similar when the cosine is close to 1, i.e. the two vectors of features point more or less in the same direction. Instead if the cosine is close to 0 it means that the feature vectors are orthogonal. Hence we are trying to understand if there is correlation, similarity, between two input points.

NOTE Encoding the similarity of two points with the kernel function is useful since if the two points are similar the target value of one of the point can be used to predict the target value of the other. If two point are not similar I will put a low (or 0) weight to the target value for the prediction.

The rationale behind the kernel model is to measure the similarity to know how much weight to set to target values of the points of the training set to predict the value of the point in the test phase.

Example - Linear kernel Let's consider the simplest kernel possible. The linear kernel correspond to the identity, in fact $\Phi(x) = x$. Given this $k(x, x')$ will be simply the scalar product²¹between the two original samples. The result of the scalar product is maximum when the two vector are pointing in the same direction. . This kernel simply represent the linear model in the input space.

If I'm able to rewrite existing algorithms (e.g. ridge regression) in a way which I'm able to have input vectors x only in the form of scalar product between features vectors, I can

²¹ $x \cdot x' = \|x\| \|x'\| \cos\theta$, where θ is the angle between x and x'

replace all these scalar products with kernels. Since, as we will see, computing kernels is much more efficient than computing the scalar product of the feature vectors is possible to obtain a much faster algorithm. This technique, known as Kernel trick, is widely used (e.g. ridge regression, perceptron, non-linear variance of PCA, Support Vector Machines, ...).

NOTE Can I use the solution found by the kernel representation and then going back to the solution in the parametric case? Yes, the problem is that usually when you use kernel methods you are using kernel (dual problem) that implicitly define a very large number of features, so if we want to go back to the feature space (primal problem) you have to compute a very large number of features, hence you do not have a large advantage in the prediction time since every time you read the input of the new sample to predict the target you have to do the feature expansion that can be very expensive. Indeed, **a very simple kernel to compute, can implicitly define an infinite number of features** and of course you cannot generate infinite features. For this reasons this is never done.

There are different type of kernel

- **Stationary kernel:** Function of difference between arguments. It is called stationary kernel since **invariant to translation in space**

$$k(x, x') = k(x - x')$$

- **Homogeneous kernel:** Known as radial basis functions, it **depends only on the magnitude of the euclidean distance between arguments** (features)

$$k(x, x') = k(\|x - x'\|)$$

i.e. depends on how much the two points are close in the euclidean space.

Should be noted that while the input x of kernel is a M-dimensional vector the output is a scalar value that tells how much the two inputs are similar. Furthermore kernel functions are valid if can be expressed as $k(x, x') = \Phi(x)^T \Phi(x')$.

7.1.2 Dual representation

Many linear models for regression and classification can be reformulated in terms of **dual representation**, in which the kernel function arises naturally (i.e. **perform the kernel trick**). We want this in order to be able to apply the kernel trick. In practice we want to **describe our model not using features but with a kernel** (i.e. **scalar product of features**). For every linear model exist a dual representation involving kernels.

NOTE This plays an important role in SVMs.

Let's take as an example ridge regression. We recall that the loss function for ridge regression, where the parameters are obtained by minimizing regularized sum-of-squares error function

$$L_w = \frac{1}{2} \sum_{n=1}^N (w^T \Phi(x_n) - t_n)^2 + \frac{\lambda}{2} \underbrace{w^T w}_{\|w\|_2^2 = \sum_{i=1}^M w_i^2}$$

and its gradient w.r.t. w is

$$\begin{aligned} \stackrel{w}{\nabla} L &= \frac{1}{2} 2 \sum_{n=1}^N (w^T \Phi(x_n) - t_n) \Phi(x_n)^T + \frac{\lambda}{2} 2w^T \\ &= \sum_{n=1}^N (w^T \Phi(x_n) - t_n) \Phi(x_n)^T + \lambda w^T \end{aligned}$$

Putting $\stackrel{w}{\nabla} L = 0$ we have

$$\begin{aligned} -\lambda w^T &= \sum_{n=1}^N (w^T \Phi(x_n) - t_n) \Phi(x_n)^T \\ w^T &= -\frac{1}{\lambda} \sum_{n=1}^N (w^T \Phi(x_n) - t_n) \Phi(x_n)^T \\ \text{Define } a_n &= -\frac{1}{\lambda} (w^T \Phi(x_n) - t_n) \quad [N \times 1] \\ &= \sum_{n=1}^N a_n \Phi(x_n)^T \quad [M \times N] \times [N \times 1] = [M \times 1] \\ w &= \left(\sum_{n=1}^N a_n \Phi(x_n)^T \right)^T \\ &= \sum_{n=1}^N (a_n \Phi(x_n)^T)^T \\ &= \sum_{n=1}^N (\Phi(x_n) a_n^T) \\ \text{Define } \Phi &= \begin{bmatrix} [\Phi^T(x_1)] \\ \vdots \\ [\Phi^T(x_N)] \end{bmatrix}, \text{ and } a = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \\ &= \Phi^T a \quad [M \times 1] \end{aligned}$$

Now we can substitute w in the loss function. To make the computation simpler, we switch to full matrix notation

$$L_w = \frac{1}{2} (\Phi w - t)^T (\Phi w - t) + \frac{\lambda}{2} w^T w$$

$$\begin{aligned}
&= \frac{1}{2}(\Phi\Phi^T a - t)^T(\Phi\Phi^T a - t) + \frac{\lambda}{2}(\Phi^T a)^T\Phi^T a \\
&= \frac{1}{2}(\Phi\Phi^T a - t)^T(\Phi\Phi^T a - t) + \frac{\lambda}{2}a^T\Phi\Phi^T a \\
&= \frac{1}{2}((\Phi\Phi^T a)^T - t^T)(\Phi\Phi^T a - t) + \frac{\lambda}{2}a^T\Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T - t^T)(\Phi\Phi^T a - t) + \frac{\lambda}{2}a^T\Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T\Phi\Phi^T a) + \frac{1}{2}t^T t - \frac{1}{2}a^T\Phi\Phi^T t - \frac{1}{2}t^T\Phi\Phi^T a + \frac{\lambda}{2}a^T\Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T\Phi\Phi^T a) + \frac{1}{2}t^T t - a^T\Phi\Phi^T t + \frac{\lambda}{2}a^T\Phi\Phi^T a
\end{aligned}$$

where $t = (t_1, \dots, t_N)^T$. Furthermore, we can notice that the number of feature M is no more present since $a \sim [N \times 1]$, $\Phi\Phi^T \sim [N \times N]$, $t \sim [N \times 1]$.

We can observe how Φ is present only when multiplied by its transpose. In this way we can use the kernels we have defined before to substitute the features. In order to have a simpler notation, we can observe that the kernel function is a **Gram matrix** $K = \Phi\Phi^T$ $[N \times M] \times [M \times N] = [N \times N]$, where each element is

$$K_{nm} = \Phi(x_n)^T \Phi(x_m) = k(x_n, x_m)$$

$$K = \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{bmatrix} \quad (57)$$

Hence given N vectors, the **Gram Matrix** is the matrix of all inner products, i.e. contains the similarity between all the pair of samples in the training set. Since the kernel function is symmetric also the Gram matrix is symmetric so its transpose is always a Gram matrix.

NOTE

- Φ $[N \times M]$ and K $[N \times N]$
- K is a matrix of similarities of pairs of samples (metric).
- K is symmetric.
- K is related to the cosine similarity and not exactly a cosine similarity since the latter needs a normalization.

Now we can substitute $\Phi^T\Phi$ with K . We write L_w as L_a because w is no longer present in the equation so L_a

$$L_a = \frac{1}{2}(a^T K K a) + \frac{1}{2}t^T t - a^T K t + \frac{\lambda}{2}a^T K a$$

so L_a is the new loss function that must be minimized with respect to a , hence the new unknowns are a and the number of unknowns is equal to the number of samples and no more to the number of features (a is $[Nx1]$). This, is a new problem, the dual problem, where features and weights of features are disappeared substituted by samples and weights of samples.

NOTE This new loss function L_a is still convex, so it has a unique optimal value, i.e. a unique global minima.

Solving for a by combining $w = \Phi^T a$ and $a_n = -\frac{1}{\lambda}(w^T \Phi(x_n) - t_n)$

$$a = (K + \lambda I_N)^{-1} t \quad [Nx1] \quad (58)$$

that is the closed formula of the new optimal weights a very similar to the one we find in the past for ridge regression

$$w* = (\Phi^T \Phi + \lambda I_M)^{-1} \Phi^T t$$

in particular there are some relationship between the formulas:

- $\Phi^T \Phi$ ($[MxM]$) corresponds to $K = \Phi \Phi^T$ ($[NxN]$).
- $\Phi^T t$ ($[MxN] \times [Nx1]$) corresponds to t ($[Nx1]$), and Φ^T ($[MxN]$) is needed to pass from the sample space of t ($[Nx1]$) to the feature space M , so this is not needed for a since t is already in the sample space.
- I_M ($[MxM]$) correspond to I_N ($[NxN]$)

Solution for a can be expressed as a linear combination of elements of Φ , whose coefficients are entirely in terms of kernel $k(x, x')$, from which we can recover original formulation in terms of parameters w . This means that the **loss function is convex thus it has only one global minimum**. We can observe how **all the element** in equation (58) have **dimension dependent only on the number of samples**. This is exactly what we were looking for, because we have said that the **complexity of kernel methods depends on the number of samples and is completely independent on the number of features M**. It practically means that **now we have to invert a $[NxN]$ matrix instead of a $[MxM]$ so the complexity will be $\sim O(N^3)$ and no more cubic in M** . In a case where the number of features is much larger than the number of samples there is great advantages using this approach. **Furthermore, the closed formula requires only the computation of the similarities K and not the computation of the features** (we will see later that K i.e. the computation of kernels can be done without computing the features).

NOTE a is a weight associated to each sample of the training set.

When we make a prediction for a new x we have our linear regression model. If we substitute $w = \Phi^T a$.

$$\begin{aligned} y(x) &= w^T \Phi(x) \\ &= a^T \Phi \Phi(x) \end{aligned}$$

but noticing that we obtain the kernel expression where Φ is the matrix of features over the training samples

$$\text{Define } k(x) = \begin{bmatrix} k(x, x_1) \\ \vdots \\ k(x, x_N) \end{bmatrix} \quad [Nx1]$$

$$y(x) = k(x)^T (K + \lambda I_N)^{-1} t \quad [1xN][NxN][Nx1]$$

As we can see, the prediction is **a linear combination of the target values from the training set**. We can get a very nice intuition of this. Making a linear combination of the target values is like **taking a weighted average over the target samples, based on the similarity between the new samples and the target samples in the training data**. In this way, we have completely **eliminated the parameters**, so we have found a **dual representation of the parametric model eliminating the need of parameters and features, for describing the model, by using kernels**. So the "model" is now **implicit in the data**. This approach have several advantages

- Solution for a is entirely described in terms of kernel functions. Once we get a we can recover w as a linear combination of Φ using $w = \Phi^T a$
- When computing the solution we need to invert a $[NxN]$ matrix and not a $[MxM]$ matrix. This is good when the number of features is very high w.r.t. the number of samples.
- The **true advantage is that for some kernel, we don't even need to compute Φ** . Doing so we **resolve** a lot of **issues revolving around the high number of features**. We will see how we can work even with infinite features.
- Kernel functions **can be defined not only over simply vectors of real numbers, but also over objects as diverse as graphs, sets, string, and text documents**.

NOTE It exist a feature space where there is an inner product between the feature vectors that produce exactly the same result of the kernel, but you do not need to compute that feature space. For example $e^{-\|x-x'\|^2}$ can be a kernel since its values go from 0 to 1, and in particular this is a case where the number of features that encode the same function is

infinite. Is only important that someone has proved that the kernel function we are considering respect the definition, i.e. exists a feature vector that encodes that function and $k = \Phi\Phi^T$.

Our **goal** is to find reasonable kernel functions, i.e. **kernel functions that encodes well for our problem what is the concept of similarity** in our example and it also need to be a valid kernel function, i.e. that can be represented as a scalar product of feature vectors: once we verified this we can **compute them quickly with their formula without computing the features since I am working in the sample space**.

NOTE Anytime we **define a metric in the sample space and we work in the sample space, is implicitly the same as working in the parametric space with the corresponding feature space.**

NOTE The switch to the dual problem does not solve all the problem since if before the problem **was to find a good feature space, now it becomes to find the proper kernel that encodes well the similarity for the considered problem.**

NOTE What is changed a lot is the **computational aspect**. While in the parametric method the computational aspect is dominated by the number of features instead in kernel method is **dominated by the number of samples**. Hence kernel methods **are not really good for big data problem since complexity depends on the number of examples, but really shine in problem with small data where you need to find really complex relationship between inputs and output.**

NOTE Hence, kernel methods **are instance methods**, that **directly use the training examples** for building the **prediction** without passing through a parametric model.

Summary Kernel methods can be **seen as a dual representation of parametric methods**, where the latter works on features and parameters of the features and kernel methods work instead directly on the samples, **replacing the concept of feature with the one of kernel that measure the similarity of known samples**. Two input points are **similar** according to some kernel measure, we are implicitly saying that the **target value associated to this two input points are correlated (not that the two input are similar themselves)** and **higher is the value of the kernel higher is the correlation of the target of the two samples**. Usually when two points are close in the input space their target value is similar, but in many cases also very far points in the input space are strongly correlated in the target space, and is this behaviour we want to capture using the kernel functions. (For instance consider the problem of finding the number of sales of a certain product for each month of the year. For many problems the sales have a seasonality, so even if the sells are far in time they are in the same season, so the target value for the product in a certain season are correlated).

NOTE The best kernel is the one that express the similarity among samples that have an high correlation of the target value. Instead, the best features are the one that express the relation between input variable and target variables.

NOTE Instance based method, memory based method are synonymous. Kernel methods are an instance based method.

NOTE The model $y(x)$ is obtained by a linear combination of the target values, so $k(x)(K + \lambda I_N)^{-1}$ represent the weights of the target values t . The effect of this weights is to give more weights to target values associated to examples x_n that are similar according to the kernel measure k to the query point x , hence the **weights are a measure of the correlation between the target value of training points and the target value where you want to make the prediction**. Is not necessary that the weights sum up to one.

Depending on the type of kernel functions the correlation between the target of x_n and x can be found in different ways, so the **knowledge about the problem could be used to find the best kernel function for the considered problem**.

NOTE Should we normalize the kernel function? In many cases the kernel functions are normalized, but this is not required, because for instance, if you want to compute extrapolation you have not to normalize the kernel because you are express something different, so I may want to have a value that may grow beyond the minimum value and the maximum value (e.g. a linear model may predict something that is outside the range).

Remember The similarity is used to weight the target value associated to each examples in the training set. For this reason the training set (both examples and targets) must be kept in memory and never be discarded to compute the similarity k , while with a feature based model we only have to store the weights for the features. Hence, in kernel methods there is not an explicit model, $y(x)$ represent the prediction done using training data, i.e. how we compute the value for a query input. In parametric methods the training set is no more useful for the prediction once the model has been learned, indeed the prediction is done using the learned weights and the feature expansion of the query input as $w^T \Phi(x)$. From this point the advantages in the model training , since kernel methods do not have to learn the model, but the only thing they can do in the training phase is to pre-compute once for all $(K + \lambda I_N)^{-1}$, that must be used for all predictions.

Remember The kernel matrix K (Gram matrix) encode using values in the input space how much is the correlation in the target space, indeed kernel methods work in the input space and have to output a value in the target space.

History Kernel methods are the reason of the winter of neural networks (parametric method) in the 90's since are very powerful tools that works very well for small data problems and have many theoretical properties that NN do not have. At that time the problems were characterized by a small amount of data (no Big data). With the incoming of problems where a lot of data is available, an approach such as NNs that is able to compute efficiently the solution, even with less theoretical guarantees, parametric methods returned to shine.

7.2 Advantage of dual representation

What are the advantages of this dual representation w.r.t. the feature representation?

- The solution for a is expressed entirely in terms of kernel function $k(x, x')$.
- Once we get a we can recover w as linear combination of elements of $\Phi(x)$ using $w = \Phi^T a$. So we would be able to compute the weights and the features.
- In parametric formulation the solution is $w_{ML} = (\Phi^T \Phi)^{-1} \Phi^T t$. Hence, instead of inverting an $M \times M$ matrix we are inverting an $N \times N$ matrix, i.e. an apparent disadvantage but actually is an advantage in case we want to use a very high dimensional feature space, more than the training samples. Then I use other measures to avoid overfitting, like regularization or other techniques. Indeed, is quite cheap computationally speaking to use very large features spaces since I do not have to compute the features explicitly, so I am able to have a bias almost to 0 because the features spaces the method allow to consider are very big.
- The advantage of dual formulation is that we can work with kernel function $k(x, x')$ and therefore
 - Avoid working with a feature vector $\Phi(x)$
 - Avoid problems associated with very high or infinite dimensionality of x
 - kernel functions can be defined not only over simply vectors of real numbers, but also over complex objects as diverse as graphs sets, string, and text documents, indeed the only thing you have to provide is a similarity measure, something that express the correlation of the target values given the inputs, so this measure can be for example how much two strings are similar.

7.2.1 Kernel construction

To exploit kernel substitution we need valid kernel functions so we must be able to construct a kernel. In particular we are interested on how we can avoid to compute Φ directly, even if $K = \Phi^T \Phi$.

The most naive way to construct a kernel is through the scalar product of Φ by its transpose. Nothing special, in fact we don't have any special gain doing this because we still need to

use input feature Φ . Indeed after we have choose the feature space we map $\Phi(x)$ and use it to find corresponding kernel. In one dimensional input space:

$$k(x, x') = \Phi(x)^T \Phi(x') = \sum_{i=1}^M \Phi_i(x) \Phi_i(x')$$

Where $\Phi(x)$ are basis functions such as polynomial.

It exist a second and much more interesting method to **construct directly kernels**. The kernel function we choose has to correspond to a scalar product in some space (perhaps infinite dimensional space). We make an example to better understand what it means.

Example - Kernel construction Suppose to have the kernel function $k(x, z) = (x^T z)^2$. Since not all the functions are kernel functions is this function a valid kernel function or not? To be a valid kernel we need to find a features expansion that is able to provide the same result as the initial kernel definition. Suppose we are in two dimensional space.

$$\begin{aligned} k(x, z) &= (x^T z)^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ &= [x_1^2 \quad \sqrt{2}x_1 x_2 \quad x_2^2] [z_1^2 \quad \sqrt{2}z_1 z_2 \quad z_2^2]^T \\ &= \Phi(x)^T \Phi(z) \end{aligned}$$

So what's the point? We have checked if $k(x, z)$ is a valid kernel by finding a feature expansion whose inner product is equal to the kernel. Furthermore we can notice that the feature mapping takes the form $\Phi(x) = [x_1^2 \quad \sqrt{2}x_1 x_2 \quad x_2^2]$, comprising all second order terms with a specific weighting. The very nice thing about this is that we can avoid to compute Φ and then perform the scalar product, because we can directly estimate the kernel function (metric) since **computing $k(x) = (x^T z)^2$ is computationally less expensive than computing $[x_1^2 \quad \sqrt{2}x_1 x_2 \quad x_2^2] [z_1^2 \quad \sqrt{2}z_1 z_2 \quad z_2^2]^T$** . Hence, computing directly the kernel is way cheaper than computing Φ . In this case we have

- **Naive kernel construction:** compute 6 features values (two squares and 2 multiplication each feature vector) and 9 multiplication plus a summation for inner product.
- **Direct estimation of valid kernel:** 2 multiplication and a squaring

This is a very simple example and the gain is very small. **If we consider the kernel $k(x, z) = (x^T z + c)^p$ (where c is a constant), we can demonstrate that the feature expansion that represent this kernel includes all the possible monomial from degree 0 to p , so the implicit feature space has all monomial from degree 0 to p .**

- **Naive kernel construction: exponential grow in number of operation, since the number of features, i.e. the number of monomials grows exponentially with p .**

- **Direct estimation of valid kernel:** linear grow in number of operation since we are multiplying for p times

We can represent features expansion that include billions of elements (that would require a lot of time to be computed) with very simple kernel which need few operation to be computed. In this way we have constructed a memory based method which doesn't use both features and weights, but it exploit the training data only to predict new samples.

NOTE The most used kernel functions are:

- **polynomial kernel:** $(x^T z + c)^p$
- **radial basis kernel** $e^{-\|x-z\|^2}$, i.e. based on a gaussian like shape, since is the one that encodes the euclidean distance test.

NOTE Is important to bear in mind that **the knowledge of the problem can be used to engineer the kernel function** that is the focal point of the kernel method, so **a well build kernel function could make the difference**, instead of using the commonly used ones. Obviously using a crafted kernel you **need too prove that is a valid kernel**.

NOTE Kernels are able to encode very large features spaces with requiring a low computational effort.

Now we can **define more formally** how we can demonstrate that a given kernel is valid, without having to explicitly construct the function $\Phi(x)$. Necessary and sufficient condition for a function $k(x, x')$ to be a kernel is that the gram matrix K , whose elements are given by $k(x_n, x_m)$, is positive semi-definite²²for all possible choices of the set $\{x_n\}$ i.e. for any possible for any possible training set.

NOTE Being positive semi-definite is not the same thing as a matrix whose elements are non-negative. It means that $x^T K x \geq 0$ for non-zero vectors x with real entry i.e. $\sum_n \sum_m K_{n,m} x_n x_m \geq 0$ for any real numbers x_n, x_m .

Theorem 7.1 (Mercer's theorem). *Any continuous, symmetric, positive semi-definite kernel function $k(x, y)$ can be expressed as a dot product in a high-dimensional space.*

New kernels can be constructed from simpler kernels as building blocks. Be aware that a really meaningful kernel is the one that define a good metric for representing the similarity between two inputs. So given kernels $k_1(x, x')$ and $k_2(x, x')$, the following new kernels will be valid (i.e. **this operations preserve validity**)

1. $k(x, x') = ck_1(x, x')$

²²Positive semi-definite means that $x^T K x \geq 0$, $\forall x : x_i \in \mathbb{R}^+$

2. $k(x, x') = f(x)k_1(x, x')f(x')$, where $f(\cdot)$ is **any function**
3. $k(x, x') = q(k_1(x, x'))$, where $q(\cdot)$ is a **polynomial with non-negative coefficients**
4. $k(x, x') = \exp(k_1(x, x'))$
5. $k(x, x') = k_1(x, x') + k_2(x, x')$
6. $k(x, x') = k_1(x, x')k_2(x, x')$
7. $k(x, x') = k(\Phi(x), \Phi(x'))$, where $\Phi(x)$ is a **function from x to \mathbb{R}^M**
8. $k(x, x') = x^T Ax'$, where A is a **positive semi-definite matrix**
9. $k(x, x') = k_a(x_a, x'_a) + k_b(x_b, x'_b)$, where x_a and x_b are variables with $x = (x_a, x_b)$
10. $k(x, x') = k_a(x_a, x'_a)k_b(x_b, x'_b)$

Remember The kernel function must be equal to the dot product of feature vectors for two reasons:

- **the scalar product is related to the similarity of two vectors**, indeed we have the product of the norms of the vectors and the cosine of the two vectors.
- **the kernel trick is applied to parametric method to make features disappear**. So we want to have the features to be replaced with the kernel. And we are able to do this if we find the scalar product between two feature vectors, avoiding the explicit computation of the feature vectors

Example - Gaussian kernel A commonly used homogeneous kernel is the Gaussian kernel.

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) \quad (59)$$

Being an homogeneous kernel, it's based on a distance metric. In particular $\|x - x'\|^2$ represent the euclidean distance between x and x' .

Gaussian kernel validity. We can demonstrate its validity through kernel composition. We can expand the square as

$$\|x - x'\|^2 = x^T x + x'^T x' - 2x^T x'$$

that can be seen as the summation of linear kernels. To give

$$\begin{aligned} k(x, x') &= \exp\left(-\frac{1}{2\sigma^2}k_1(x, x')\right), \quad \text{where} \\ k_1(x, x') &= x^T x + x'^T x' - 2x^T x' \end{aligned}$$

We know that the exponential of a valid kernel is still valid (rule 4) so we need to demonstrate that $-\frac{1}{2\sigma^2}k_1(x, x')$ is valid. For composition (rule 1) we need to demonstrate that $k_1(x, x')$ is valid because $\frac{1}{2\sigma^2}$ is only a coefficient. We also know that for (rule 5) the sum of valid kernel is still valid, so we need to verify the components of $k_1(x, x')$. All three components are just linear kernels with some coefficient (rule 1), so they are valid. \square

Here we can appreciate the power of kernel composition. In fact, **we don't know which is the feature expansion of the Gaussian kernel, but we are sure that it exists.** This is once more a demonstration of how kernel methods don't need to define the features. Still, they correspond to the dual representation of a parametric model, where, in the **case of Gaussian kernel, the number of features is infinite.**

We can also expand the Gaussian kernel to non-Euclidean distances

$$k(x, x') = \exp\left(-\frac{1}{2\sigma^2}(k_i(x, x) + k_i(x', x') - 2k_i(x, x'))\right) \quad (60)$$

7.3 Kernels for symbolic data

We have said how kernels can be defined over real vector numbers but also object as diverse as sets, graph, strings and text. Indeed, **kernels can be expanded to inputs that are symbolic** (e.g. graphs, strings,...), rather than simply vectors of real numbers.

Example - Sets To define a simple metric over sets we can use

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|} \quad (61)$$

Indeed is a reasonable way to measure the similarity of two sets looking for how much they overlaps, the more they are overlapping the more their target values are correlated. In practice, we find the number elements included in both A_1 and A_2 , i.e. common elements. Then we use this number as an exponent.

Example - Generative models We define the generative model $p(x)$ which is a mapping in a one-dimensional feature space. The kernel is defined as

$$k(x, x') = p(x)p(x') \quad (62)$$

where $p(x)$ represent a probability over the input space, i.e. the probability that I would be asked to predict for a certain point, the probability of generating a new sample in the input space. Performing the multiplication between $p(x)$ and $p(x')$ is like doing a inner product in a one-dimensional space so the kernel is valid. In practice we are multiplying the probability of x and x' . So the kernel defines the probability of having both x and x' and so their "similarity" considering that **points more likely to be generated are more similar**, i.e. are similar if they have high probabilities.

7.4 Radial Basis Function Networks

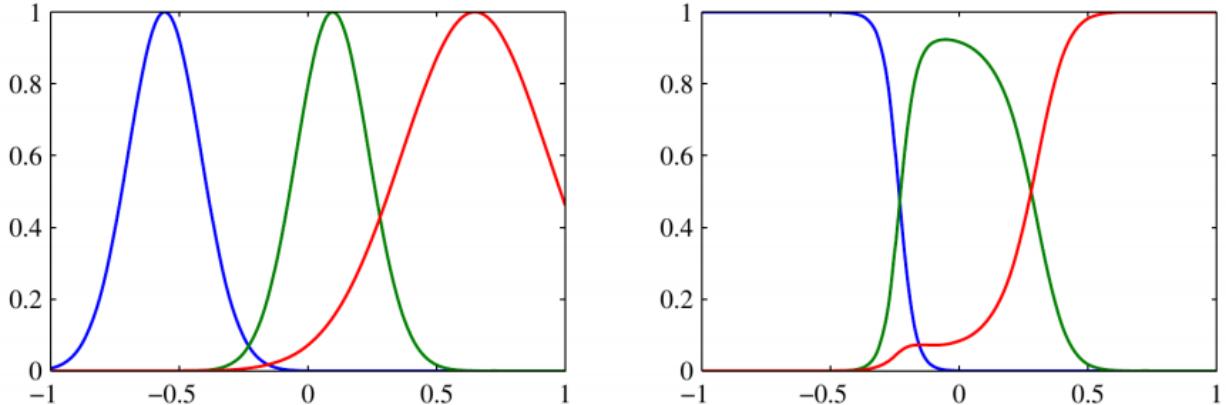
This **method** is used to make **regression**. Each basis function depends only on the radial distance (typically Euclidean) from a center μ_j :

$$\Phi_j(x) = h(\|x - \mu_j\|_2)$$

One of the popular radial basis function h is the Gaussian function (Bell Φ centered in μ_j). Originally it was used for interpolation:

$$f(x) = \sum_{n=1}^N w_n h(\|x - x_n\|_2)$$

where the interpolation is done using the summation of radial basis function centered in the points of the point to interpolate, each one considered with a certain weight. Because the data in ML are generally **noisy**, exact interpolation is not very useful, but **inspired** by the approach we can obtain a **kernel method** based on radial basis functions. To do that the **first** thing we have to do is to **normalize** the basis functions, having that the **sum of the basis function over the input space** is always equal to 1 (i.e. in each point of the input space). Normalization is used sometimes in practice as it avoids having regions of input space where all basis functions take small values, which would necessarily lead to predictions in such regions that are either small or controlled purely by the bias parameter.



We can notice this fact on the left picture where in **the point $(-1, 0)$** : without normalization you would have low power of representation since all the basis function in this region have small values, so unless having large weights to learn something that, you are not able to learn something that give an high prediction. The normalization instead allow to have high values in all the input space, **having enough power of prediction in all the input space**.

The idea of the **Nadaraya-Watson model**, inspired by the radial basis function interpolation, is the following. Given a training set x_n, t_n what we **want to learn** is the **joint**

distribution of the two variables (i.e. of the input-output space) can be estimated with a **kernel density estimation (KDE)**: Kernel density estimation is a **popular method** to estimate density using an **instance based method**. In particular it place kernels over each example of the dataset, as we did in the interpolation, and **then we average the density expected by the kernel using all the examples**:

$$p(x, t) = \frac{1}{N} \sum_{n=1}^N f(x - x_n, t - t_n)$$

where f are the kernel functions centered in (x_n, t_n) , so the density in each point is computed by **averaging** the values of the function over each point of the input output space (they are **averaged since we do not know if there are some outliers**, we know that they belong to the correct distribution). Hence by using the samples of the training set we **estimate the density in each point of the input-output space**. Having the **joint distribution** we can **use it to make a prediction of t about a specific value of x** . By fixing x_q from $p(x, t)$ we know the distribution of t that is $p(t|x_q)$:

$$y(x) = E[t|x_q] = \int_{-\infty}^{+\infty} tp(t|x_q)dt = \frac{\int tp(t, x)dt}{\int p(x, t)dt}$$

from the **theorem of total probability**. So using the above expression of the kernel density estimation of $p(x, t)$, knowing that the kernel basis function f has the **property** $\int tf(x, t)dt = 0$, which means that the kernel basis function have zero mean if **centered in zero**, we can write:

$$\frac{\frac{1}{N} \int \sum_{n=1}^N tf(x - x_n, t - t_n)dt}{\frac{1}{N} \int \sum_{n=1}^N f(x - x_n, t - t_n)dt}$$

after some passages:

$$= \frac{\sum_{n=1}^N g(x - x_n)t_n}{\sum_{m=1}^N g(x - x_m)} = \sum_{n=1}^N k(x, x_n)t_n$$

So the prediction is given by the target value of the examples t_n each weighted by a kernel defined as:

$$k(x, x_n) = \frac{g(x - x_n)}{\sum_m g(x - x_m)} \quad g(x) = \int_{-\infty}^{+\infty} f(x, t)dt$$

being g the **radial basis function defined over the input space** (obtained by summing $f(x, t)$ over the target) while f is the **radial basis function defined over the input output space** (can be noticed since f depends on both x and t while g depends only on x). Hence the **kernel k weight of the prediction is simply a radial basis function over the input space, normalized**.

NOTE The kernel is a radial basis function over the input space, since $g(x) = \int_{-\infty}^{+\infty} f(x, t) dt$, and is normalized since $g(x - x_n)$ is divided $\sum_m g(x - x_m)$

The main takeaway from this is that if you want to **express using a kernel density estimation (KDE)**, the joint probability, you get a kernel based model where you **average** the target value of your training dataset, weighted by the similarity measure between the queried point and the examples that is a **normalized version of the radial basis function centered in the training point**.

NOTE Beyond having the point distribution of the conditional expected value I have the probabilities of all the target values given all the input points ($p(x, t)$).

Remember In linear regression when you are **minimizing the quadratic loss function** you are looking for the **conditional expected value**, so the prediction can be done using the empirical definition of the **expected value**.

The Nadaraya-Watson model is also called kernel regression. Using a localized kernel function, it has the **property of giving more weight to the data points x_n that are close to x** (indeed $g(x - x_n)/\sum_m g(x - x_m)$). As already outlined the model defines not only a **conditional expectation**, but also a **full conditional distribution** ($p(t|x)$), hence we can have something like this:

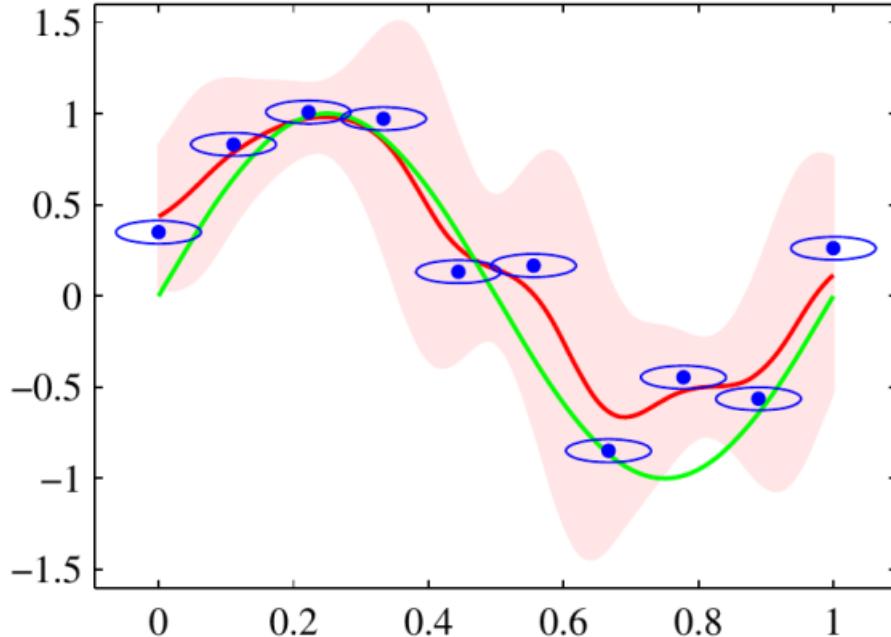


Figure 26: Isotropic Gaussian kernels centered around the data points $z_n = (x_n, t_n)$

The blue dots are the examples. **In each example we put a radial basis f represented**

by the ellipse centered in the samples. Using the kernel density estimation (KDE) we obtain the red line that represent the conditional expectation and the pink area represent a part of the predictive distribution: in the region where there are more points ($\sim 0.5, 0.8$) the uncertainty is smaller and where the points have very different target values the uncertainty is much higher ($\sim 0.38, 0.62$). The interesting part of the KDE method is that do not give only the prediction but also the probability distribution induced by the choice of the radial basis function over all the target, that allow us to express the uncertainty over the prediction.

NOTE As said several times is very important to know how much we are confident of our prediction because this can help people that use the prediction of the model to understand if they can be trusted or not.

7.5 Gaussian processes

In the previous sections, we have seen how we can find the dual representation of ridge regression based on kernels. Gaussian processes are the **kernel version of the Bayesian linear regression** when we **assume a Gaussian distribution for both prior and likelihood**. So far, we have seen how to find the dual model of a non-probabilistic (i.e. direct) model for regression like ridge regression. We can extend the **dual formulation to probabilistic discriminative models**, that **allow to have the prediction but also the uncertainty about the prediction**, since through them we **compute the distribution of the predictions** given the input. Furthermore, as the name suggest Gaussian Processes **assume a Normal distributions**.

In Bayesian linear regression we have introduced a **prior** distribution over the parameters w , that **specify where we believe the solution can be**. Given a training dataset, we evaluate a new distribution over the parameters w (posterior) by combining the prior and the new data observed (likelihood). **From the posterior we can find a predictive distribution $p(t|x)$ (i.e. the conditional distribution) for a new input point x** . With Gaussian processes, **since we are talking about instance based methods**, we define **directly a prior distribution over functions** (i.e. models)²³. Each set of parameter is associated to a function in the input output space that is your model of prediction. So a **distribution over the weights in the dual representation becomes a distribution over the functions**. Indeed, **if before we defined a prior over the parameters associated to the function, now we define a prior directly on the functions, bypassing the parameters**. If we recall that Gaussian processes are actually kernel method we can make sense of it. With parametric method the function (model) is defined by its parameters, which decide the "shape" and characteristics of the function. **In non-parametric method, the function is defined by the samples**. Now we can observe that **to define a func-**

²³With function we mean the model in the parametric world. So in linear regression it would be the hyperplane defined by the parameters

tion with samples, we would need a infinite amount of them. So what **Gaussian processes** are doing is **considering an infinite collection of variables, one for each input point, and considering them as jointly distributed as a infinite-dimensional Gaussian distribution**. OK, don't panic now we will see some graph and it will be clearer.

NOTE A single object in Gaussian processes is a function, that in regression contains infinite points, hence we are defining a prior with infinite dimensions, indeed if I want to specify the distribution over a function I want to specify the distribution for each input point, but the input points are infinite. (In other words the prior define the probability for each point, to form a function, but a function has infinite points so the prior has infinite dimensions).

Might seem difficult since we are working with a distribution over the uncountable infinite space of functions. So **we have to give a probability to each function that can be realized**. We can do this by simply considering the finite training set, indeed for a finite set we need to consider the values of the function at the discrete set of input values x_n of the training set. So we define the distribution over all the function expressed as a function of the examples of our training set, i.e. specifying this distribution over a finite set of N examples. So in practice we can work in a finite space.

Example - Gaussian processes intuition We plot a function. To define the function we should consider infinite points. For this example we only consider two points x_1 and x_2 . For each point we define $p(t|x)$ as a **Gaussian distribution**, which means that in the target space (t_1, t_2) the target values probability is Gaussian. Then, we **join the two points distribution in a multivariate Gaussian**. This distribution describe the values of our function based on the inputs x . Note that the **variables are not independent**, because the value of consequent inputs are likely to have similar values, indeed the distribution has an high variance over the diagonal and low covariance over the orthogonal dimension, so we have an high probability of taking points with that are on the same height, i.e. with similar values t_1 and t_2 respectively of x_1 and x_2 .

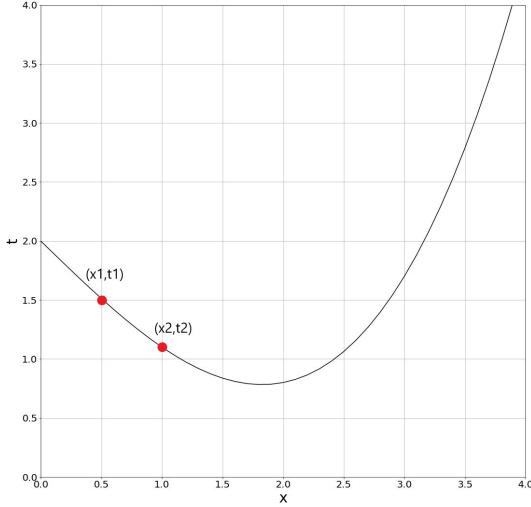


Figure 27: (a) Function we want to estimate $y(x)$ in the space (x, t)

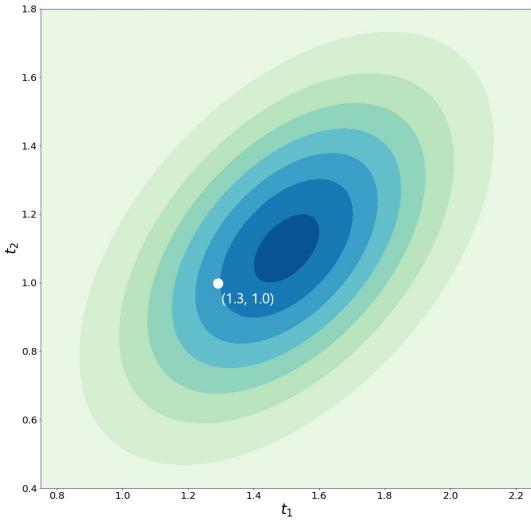
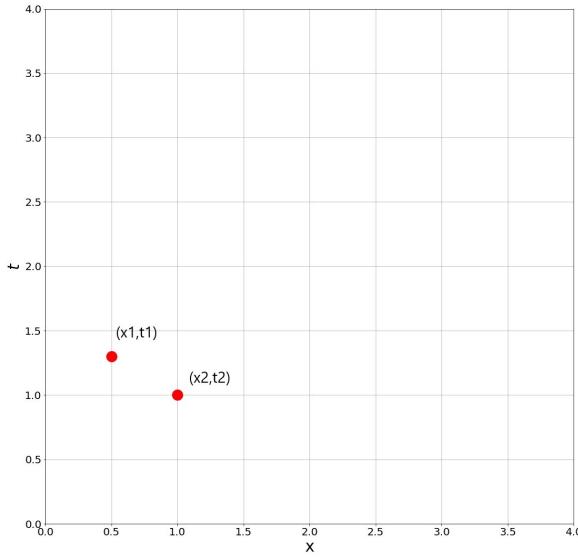


Figure 28: (b) Multivariate Gaussian over t space $p(t_1, t_2)$

If we draw a **sample from the multivariate Gaussian** we would define a new function. In fact, picking a sample is the same as picking an infinite sequence of points in the (x, t) system. So our goal is to define a multivariate Gaussian which describe as well as possible the values of every point of our original function. The white dot in figure (b) is a sample of the multivariate Gaussian representing a guess on the original function.



(c) Function estimated from the multivariate Gaussian. Correspond to the white dot in (b).

Adding points the target space dimension increases, so the dimension of the Gaussian increases. Considering all the infinite points I have the Gaussian defined over a infinite dimension, and every sample from the distribution is a realization of the infinite dimension Gaussian, i.e. a function that passes from the points drawn by the infinite dimension Gaussian distribution (as the one defined from a Gaussian distribution of dimension 2 passes from the two drawn points). Hence we define it over the training points we have.

NOTE Defining a model i.e. **the choice of the kernel function**, that is the way we weights the samples of the kernel function, we are implicitly defining the distribution over the other points since you are able to do prediction in points where you do not have samples. So is true that the distribution is defined by only the samples in the training set but then the distribution is complete for all the points so we are able to make prediction over the input points, not only for the one in the training set. Hence **implicitly we are defining a distribution over the function, over an infinite points and the parameter of this distribution depends only on the points that you measure**. In the example above we have x_1 and x_2 as examples and we want to now the target value in another point x_3 , and you can do this since we have the **kernel function that says how much x_3 is related to x_1 and x_2** .

In practice we can't work in an infinite space. In fact, we operate over the finite set of the training data. This process will produce a **distribution** which describe t . This distribution **can be used to make prediction for never seen inputs**. Now we will

explain how we can **define a prior distribution over the functions**. To do so we recall what we did for linear Bayesian regression. So taken a generic parametric model we have

$$y(x, w) = w^T \Phi(x)$$

In the case of ridge regression we have that the prior over w is

$$w \sim \mathcal{N}(w|0, \tau I)$$

$$y = \Phi w$$

where we **assume there is no correlation between the different parameters and we have some kind of uncertainty about w specified by τ** . From $w \sim \mathcal{N}$ comes the name of Gaussian processes. So, such distribution of the parameter uncertainty w which kind of distribution induces over the model y ? **How is y distributed?** For what we said as **$p(w)$ is the prior for linear Bayesian regression, $p(y)$ is the prior for Gaussian processes**. We know that the linear combination of Gaussian is still Gaussian. So knowing that y is a **linear combination Gaussian distributed variables given by the elements of w , we are sure that it is distributed as a Gaussian**. Now we can calculate its mean and variance.

$$\begin{aligned} E[y] &= \Phi E[w] = 0 \quad (\text{Uncertainty is only on } w \text{ and } \Phi \text{ is not random}) \\ Cov[y] &= E[yy^T] - E[y]^2 = E[yy^T] = \Phi E[ww^T]\Phi^T = \tau\Phi\Phi^T = K, \quad (\text{Gram matrix}) \\ K_{nm} &= k(x_n, x_m) = \tau\Phi(x_n)^T\Phi(x_m) \end{aligned}$$

so the **marginal distribution $p(y)$ is defined by a Gram matrix so that:**

$$y \sim \mathcal{N}(y|0, K) \tag{63}$$

To justify why K is the covariance matrix of y , we can observe that the Gram matrix components are the kernel function values of input pairs. We have said that the kernel function measure the similarity between two inputs. So K **measure the similarity between all inputs pairs and so the correlation of the outputs (i.e. predictions) y , i.e. represent the correlation we have in the prediction model**.

NOTE The expected value of y is null since the expected value of the prior was assumed to be null. Depending on where is centered the prior this value could change.

NOTE Remember that a Gaussian distribution over the weights reflects in a Gaussian distribution over the models.

NOTE We are talking about a vector y defined only on the training points, since K is defined only on the training set.

Now we can define a more formal and organized definition of Gaussian processes.

Definition 7.1 (Gaussian process). A Gaussian process is defined as a probability distribution over function $y(x)$, such that the set of values of $y(x)$, evaluated at an arbitrary set of point $\{x_1, \dots, x_N\}$, jointly have a Gaussian distribution

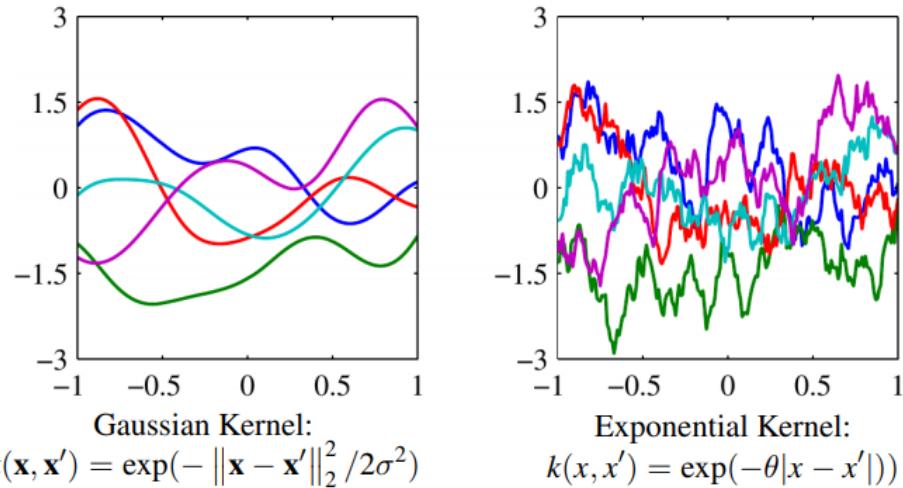
A Gaussian process, is a probability distribution over functions hence over infinite objects. So we have to imagine the function as the realization over infinite dimensional Gaussian distribution. Being a Gaussian, the distribution can be completely specified by the mean and covariance.

- Usually we do not have any prior information about the mean of $y(x)$, so we'll take it to be zero.
- The covariance is given by the kernel function:

$$E[y(x_n)y(x_m)] = k(x_n, x_m)$$

so once we define the kernel we are defining the distribution over the functions, indeed we are defining the Gram matrix which encodes the correlation between the training samples.

Example - Gaussian Processes The kernel function can be defined directly rather than indirectly through a choice of basis function.



In the latter figures the functions are samples from the distribution that works over an infinite dimensional space with zero mean and covariance defined by the kernel function choice. The regularity of the function depends on the kernel, that is the main parameter that define the distribution, indeed the kernel defines the covariance of the function distribution.

Note - Fitting As for every kernel method, the **choice of the kernel is very important**, because it defines how the inputs are correlated. A hyperparameter which control the under-overfitting (i.e. the smoothness of the function realizations) of the method is the **bandwidth of the Gaussian (σ)**. A narrow bandwidth means that inputs near each other will be highly correlated and the correlation between inputs will decrease very fast as we increase the "distance" between the inputs. On the other hand, for wider bandwidth even slightly far away inputs will be correlated. In case of overfitting we would like to use wider bandwidth because the sample will be less correlated locally, thus decreasing the probability of fitting the noise. From another point of view, if we increase the bandwidth every inputs will "see" more inputs, and so it will have more samples to estimate the function value. In the same way, if we are underfitting we can have narrower bandwidth. A narrower bandwidth indeed means that the inputs are less correlated to the other points, so the extreme situation is that the examples are uncorrelated to each other. A narrower σ would mean a more oscillating function since there is no propagation of information between close points (neighborhood points), instead a wider σ means a propagation of information so that for the prediction of a point the information of neighboring points can be used leading to smoother realizations. Recalling the multivariate Gaussian over t image:

- Narrowing σ we are reducing the correlation, so the diagonal of the covariance matrix K is increasing, and less covariance with the other points. Hence the **kernel measure the similarity only for few nearest neighbors**. Which on the image means a reduction of the orthogonal direction to the main one and an increase of the main direction length.
- Making σ wider we are increasing the correlation, so the non-diagonal elements of the covariance matrix K increase meaning more covariance. Hence the **kernel measure the similarity using more nearest neighbors**. Which on the image means a increase of the orthogonal direction to the main one.

Even with other type of kernels different than Gaussian, as the exponential kernel in the latter example, we still have an hyperparameter θ , whose choice could help us to avoid underfitting and overfitting by respectively reducing or increasing it. This hyper-parameters help to regulate the bias variance trade-off.

Remember Is very important, when studying a new technique, to ask yourself what are the hyperparameters which you can play with to regulate the variance-bias trade-off. This indeed must be done in the **model selection phase**, where you locate the points where you can leverage on to obtain the trade-off.

7.5.1 Prediction

In this section we will see how we can predict the value of our function in input points never seen before. We consider the case in which we use **Gaussian processes for regression**.

As usual for every regression method we define our target value as

$$t_n = y(x_n) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

Under the assumption that the noise is distributed as a Gaussian, we can say that the conditional distribution

$$P(t_n|y(x_n)) = \mathcal{N}(t_n|y(x_n), \sigma^2) \quad (64)$$

We can also **assume that the noise is independent on each data point**. So the **joint distribution of t is still Gaussian**

$$p(t|y) = \mathcal{N}(t|y, \sigma^2 I) \quad (65)$$

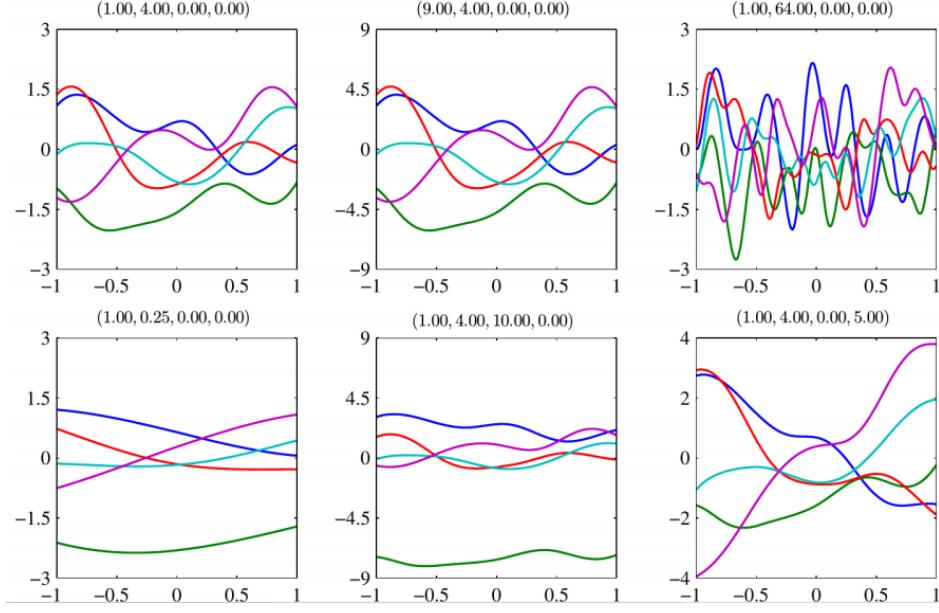
Since as we have seen a Gaussian distribution over the prior $p(w)$ is the same as considering a gaussian distribution over the model $p(y) = \mathcal{N}(0, K)$, we can compute the **marginal distribution $p(t)$ over the target space**

$$p(t) = \int p(t|y)p(y)dy = \mathcal{N}(t|0, C), \text{ where } C = K + \sigma^2 I_N \quad (66)$$

so from two Gaussian $p(t|y)$ and $p(y)$ we obtain a Gaussian, whose covariance simply add since the two Gaussian are independent. We can notice how a part of the convariace is due to the uncertainty of the model (K) and the other one due to the intrinsic noise we have in the dataset (σI_N).

NOTE We are making the homoscedastic assumption by considering that all examples have the same amount of noise. In some cases this is not true because you have different variance for different points in the input space.

Example - Gaussian Processes In the following example is clear how to change the several hyperparameters θ_i is possible to obtain various hypotheses space, i.e. function that can be realized using that kernel function. Is clear how the choice of the kernel, by setting θ_i is like choosing the features in the primal method:



Hence, as already pointed out, the selection of the kernel of the dual substitute the choice of the features in the primal.

Now suppose we want to **make a prediction** t_{N+1} for a new data input x_{N+1} , given the training data. Our **goal** is to **evaluate** the predictive distribution $p(t_{N+1}|t^{(N)}, x_1, \dots, x_{N+1})$ ²⁴. We can calculate that

$$p(t^{(N+1)}) = \mathcal{N}(t^{(N+1)}|0, C^{(N+1)}), \text{ (prior distribution) where}$$

$$C^{(N+1)} = \begin{bmatrix} C^{(N)} & k \\ k^T & c \end{bmatrix}, \quad [(N+1)\times(N+1)]$$

$$k = [k(x_1, x_{N+1}) \dots k(x_N, x_{N+1})]^T, \quad [N\times 1]$$

$$c = k(x_{N+1}, x_{N+1}) + \sigma^2, \quad [1\times 1]$$

where k is a vector that measure the kernel between the query point x_{N+1} and all the training points, and c is a scalar value that adds the term that consider the similarity between the query example and itself plus the irreducible noise (since $C = K + \sigma^2 I$). **Hence we are adding to the covariance matrix C a row and a column that consider the new point (x_{N+1}, t_{N+1}) .** We can observe that for the Gaussian distribution properties, **being the prior a Gaussian distribution the predictive distribution is still a Gaussian**. From that, we can apply the properties over conditional Gaussian distribution to obtain $p(t_{N+1}|t^{(N)}, x_1, \dots, x_{N+1})$.

$$p(t_{N+1}|t^{(N)}, x_1, \dots, x_{N+1}) \sim \mathcal{N}(t_{N+1}|k^T C^{(N)^{-1}} t, c - k^T C^{(N)^{-1}} k) \quad (67)$$

²⁴ $t^{(N)}$ is the t vector when we have N sample. The professor in the lecture uses \mathbf{t}_N but I found it a little misleading, because the difference between the bold character and the normal one can be easily missed. Also putting the number of sample as a superscript reminds of the iterations count in other notation

that derives from the formulas to obtain a conditional distribution from a joint distribution with Gaussian.

- The mean:

$$m(x_{N+1}) = k^T C_N^{-1} t$$

so the **mean prediction** is computed **averaging the target values weighted by the similarity** $k^T C_N^{-1}$.

- The variance:

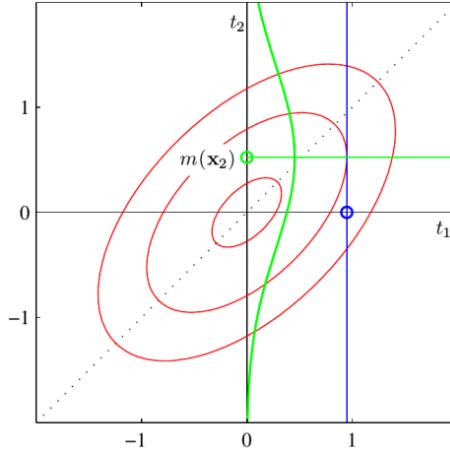
$$\sigma_{N+1}^2 = c - k^T C_N^{-1} k$$

Where C_N^{-1} can be computed once for all while k needs to be computed once I know the sample for which I want to know the prediction x_{N+1} . We can observe two things. The **mean and the variance depend on** x_{N+1} . More interestingly, **the mean of the prediction is actually what we obtain for the kernel version of ridge regression**. This shouldn't be a surprise, because we have already said that **in the parametric world, ridge regression is a particular case of linear Bayesian regression, when the prior is Gaussian and centered around zero**. So this particular case holds also in the kernel world. We see that **to calculate our prediction we need to invert C** . This operation is always possible because by definition K is a Gram matrix and so its semi-definite positive. If we add to K the term $\sigma^2 I$, we are **sure that C is positive definite** (adding a term that is surely grater than 0, and so are the eigenvalues) and so it is for sure invertible.

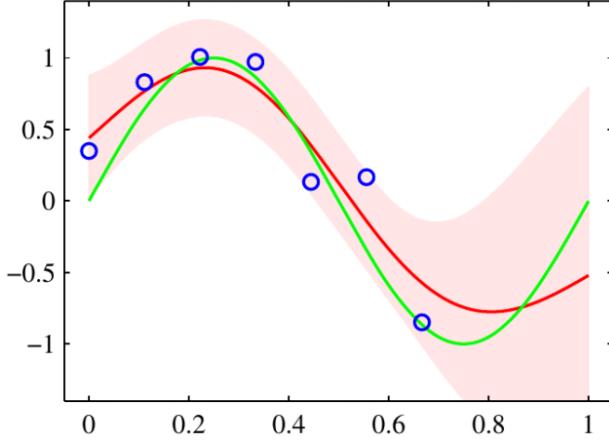
NOTE The prior is defined over the vector of target, so is the joint distribution over all the target values. The posterior $p(t_{N+1}|t^{(N)}, x_1, \dots, x_N + 1)$ instead is related to a single point t_{N+1} , and is conditioned to observing the data.

Note - computational cost As usual, inverting a matrix is the most intensive operation of the solution. In our case, the complexity of inverting C will be $\mathcal{O}(N^3)$. Luckily we need to compute this only once for the given training set. We can also observe that the **complexity depends only on the number of samples, as it should be for kernel methods** (and not on the number of features that implicit and never computed). When we obtain a new sample, we can use the already calculated $C^{(N)}{}^{-1}$ to simplify the complexity of calculating the mean and variance of the predictive distribution. Indeed, we have that the **computational cost of the mean is $\mathcal{O}(N)$ and of the variance is $\mathcal{O}(N^2)$** . For large dataset is very expensive to calculate exactly the result, so we resort to approximated methods like random sampling and clustering (try to cluster data in groups and then pick a representative sample from each group to reduce the number of samples that are used in the prediction).

Example - Prediction Suppose to have a regression problem we want to solve with Gaussian processes. For simplicity we assume that our function is described by t_1 and t_2 , which are the function value for x_1 and x_2 . First, we **construct** our **prior** $p(t) = p([t_1, t_2])$ (red ellipses). We assume a zero mean distribution (red ellipses centered in $(0, 0)$), where the **shape of the multivariate Gaussian prior depends on the covariance matrix, and so on the kernel we choose**. Now we haven't observed any data, so our best guess for both t_1 and t_2 is zero (i.e. the mean value of the prior). Now we **observe** in x_1 a **value for t_1** (blue dot) i.e. measured. We know that t_1 and t_2 are correlated, so observing t_1 will give us some information about t_2 . We know that the **predictive distribution** $p(t_2|t_1, x_1, x_2)$ (green Gaussian) is a **Gaussian**. In the prior plot below we can visualize this distribution through **cutting the prior $p(t)$ parallel to t_2 through t_1** (blue line). This slice of $p(t)$ will be $p(t_2|t_1, x_1, x_2)$ (green Gaussian) that **tells the value of likelihood of t_2 value for each point along the blue line**. Now we can **take the mean to estimate** $t_2 \leftrightarrow m(x_2) = \int t_2 p(t_2|t_1, x_1, x_2) dt_2$ (green point). So the **prediction of $t = t_1, t_2$** would be in the **intersection between the blue line and the green line**. So the **correlation among the training points allowed me to change the prediction looking at the training set**, predicting the value in all the other points. Having no observation I predict 0 since the prior says that the target value without any information is 0 (**mean value of the prior**), i.e. I have no data correlated to the one I want to predict so the only thing that I could say is the mean value.



Using a specific kernel we will have the blue points as examples of the dataset. Given the points we would **predict the red line and also the uncertainty related to the covariance estimated in each point given the examples**. The **uncertainty cannot be reduced below the intrinsic noise**, but we also have the **problem** of having an **high uncertainty in the region where there are no examples**, instead in region with many samples and in region where the samples are very similar the uncertainty reduces. As the definition outlines $C = K + \sigma^2 I_N$, so when computing C^{-1} the **contribution of K goes to 0 with N that goes to ∞** , while the term of the **intrinsic noise depends on the number of examples and is not eliminable**. The green line represent the true model, i.e. that generates the blue circles.



NOTE The matrix $C^{-1} = (K + \sigma^2 I_N)^2$ is taking into consideration the density of the examples, so you will have **high uncertainty in the region where we have few examples and instead lower uncertainty in the region where we have more examples**. Hence, **in the regions where you have a lot of samples the contribution of K goes to 0 ($N \rightarrow \infty \implies C^{-1} = I/\sigma^2$)**, instead in region with fewer example it increase the **uncertainty**.

NOTE Not observing any couple (x, t) the only information we have are given by the **prior**, so mean (i.e. prediction) 0 and uncertainty equal to the covariance matrix of the prior, i.e. maximally uncertain.

As we have said before, like we need to define features in the parametric world, we need to define a kernel in the non-parametric case. Indeed kernels have some **hyper-parameters**, that define the hypotheses space. I.e. the **performance** of a **Gaussian process** are heavily affected by the choice of the hyper-parameters for the kernel. How can we find the optimal values for this hyper-parameters? The choice of kernel hyper-parameters is a **model selection problem**. We have already seen how **cross validation** can be used to do model selection. This method is **very robust**, but at the **same time** it is **slow (due to the cubic complexity of the Gaussian processes)**. Another approach uses the **maximization of the marginal likelihood** using gradient optimization. In practice, you want to find the hyper-parameters for which the observed target variables are more likely, i.e. that tries to explain well data that you observe. This is **faster**, but **unfortunately it is not a convex problem so there are local minima and the solution can be suboptimal**. Usually the gradient approach is the go to method for hyper-parameters optimization.

NOTE The **marginal likelihood is the likelihood of the model given the input** (e.g. in the case of log marginal likelihood $\log p(y|x)$). So by maximizing it we change the parameter of the kernel so that your model has a higher likelihood.

Other tricks for model selection are:

- Use **domain knowledge** wherever possible
- **Standardize input data** and set length scales to ~ 1
- **Standardize targets** and set function variance to ~ 1
- Set **initial noise level high**, even if you think your data have low noise, such that the optimization surface for your other parameters will be easier to move in.

Theorem[section] Definition[section]

8 Support Vector Machines

Support Vector Machines (SVMs) is a **kernel method** that were invented in the present form in the late 90's. It is **one of the best methods for classification**. Is one of the most theoretical complete method in the machine learning world. Its **theoretical guarantee** are **very good**, and they reflect also in **very good performance in practice**. It is one of the most mathematical and difficult problems in machine learning since you have to understand the learning theory, the kernel theory and the constrained optimization. As a consequence people use SVMs as black boxes, without having a real knowledge of how they really work. So our goal is to have a basic understanding of how SVMs work and what are the important parameters. In particular we will see the main use of SVMs, i.e. the classification tasks, but there are also SVMs for other things such as the SVRegressor for regression, or clustering, feature selection, dimensionality reduction.

SVMs as said are an **instance based method** and are **composed by three elements**

1. **Subset of training data** Being a kernel method the complexity depends on the number of samples we consider. SVM **works on particular subset S of training samples to reduce complexity and find sparser solutions**. This subset is called **support vectors**, the **other samples are discarded**. This **helps to lower the complexity of the method**.
2. **Vector of weight α** This vector is used to **weight the training samples subset (i.e. the support vectors)**.
3. **Kernel** SVMs are a kernel method, so they need a **similarity function** (kernel) to work.

SVM is mainly used for classification. From now on we will **consider a binary classification problem**. In this case, after we have defined the three elements specified above, the **class prediction for a new example x_q** is

$$f(x_q) = \text{sign} \left(\sum_{m \in S} \alpha_m t_m k(x_q, x_m) + b \right), \text{ where} \quad (68)$$

S set of indices of the **support vectors** (i.e. which are the samples considered)

α **vector of weights** (i.e. which are the weights of these samples)

t **target vector**

$k(x_q, x_m)$ **kernel** (i.e. **measure the similarity**)

hence **the class of the input is computed considering the weighted sum of support vectors target where the weight comes from the multiplication of the support vector weights and the similarity of the input with the support vectors**. Furthermore,

b is the bias term (in the sense that is not multiplied by any other term), a constant that could be added.

NOTE α are not user defined, but defined by the method, as was for a in its kernel version that were defined by a formula.

NOTE This is valid for binary classification but can be extended to multiple classification by applying what we saw for linear classification.

NOTE The SVMs are a kernel method indeed in the last formula there is no sign of the input feature, everything depends only on the training samples.

This is a very smart way of doing instance based learning. They are usually presented as a generalization of the perceptron indeed the solution formulation is somewhat similar to the one of the perceptron. But what is the **relation between perceptrons and instance based learning**? Now we will try to explain how we can derive this solution by revisiting the perceptron, a parametric method. We know that the prediction for perceptron is define as

$$f(x_q) = \text{sign}(w^T \Phi(x_q)) = \text{sign}\left(\sum_{j=1}^M w_j \Phi_j(x_q)\right)$$

We can also recall that the various weight are updated using gradient descent as

$$w^{(k+1)} = w^{(k)} + \alpha \Phi(x_n) t_n$$

where the superscript indicate the iteration step. If we assume that every weight start from zero, every weight can be calculated as

$$w_j = \sum_{n=1}^N \alpha_n t_n \Phi_j(x_n)$$

(in particular in the perceptron the update was done only over misclassified examples, but here we suppose to do that over all the examples). Now we can put our new formulation of w_j into the perceptron function

$$\begin{aligned} f(x_q) &= \text{sign}\left(\sum_{j=1}^M \left(\sum_{n=1}^N \alpha_n t_n \Phi_j(x_n)\right) \Phi_j(x_q)\right) \\ &= \text{sign}\left(\sum_{n=1}^N \alpha_n t_n \sum_{j=1}^M (\Phi_j(x_n) \Phi_j(x_q))\right) \\ &= \text{sign}\left(\sum_{n=1}^N \alpha_n t_n (\Phi(x_n) \cdot \Phi(x_q))\right) \end{aligned} \tag{69}$$

We can observe that the only elements dependent on m are the features Φ_j . We can rewrite the sum over m as a dot product between $\Phi(x_q)$ and $\Phi(x_n)$. Doing so, the sum over features has been rewritten as a sum over the samples. Furthermore, the feature appears only as dot product, so we can find a kernel representation for this function. What before was a parametric method, now is an instance-based method.

$$f(x_q) = \text{sign} \left(\sum_{n=1}^N \alpha_n t_n k(x_q, x_n) \right)$$

Our prediction is now become a weighted average over the target value and the similarity between the input and the training data samples, so we rewrited the sum over the features as a sum over the samples. What we are doing is like a weighted kNN.

NOTE Now, one can argue that this method has parameter even though it is non-parametric. **Non-parametric** it doesn't mean that our method doesn't have parameters, but that they are related to the samples, instead of the features, as we can see in the example above.

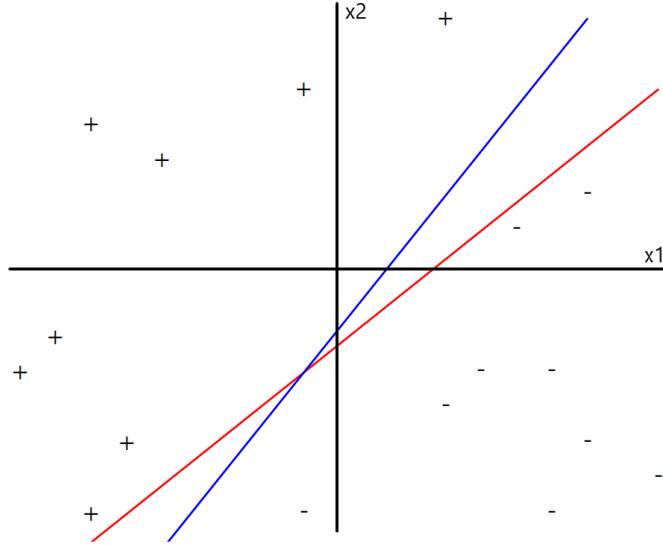
Hence, to obtain a SVM from the perceptron we can replace the dot product with an arbitrary kernel $k(x, x')$. **So the perceptron can be seen as a special case of instance-based learning.** Indeed, now we have a much more powerful **learner** that is able to work into **very large dimensional spaces**. Having the ability of working into high dimensional feature spaces increases the probability that a linear classifier is able to correctly classify your data since we are projecting it into a very high dimensional space. So (for the theorem??) if a symmetric matrix K is positive semi-definite (i.e. has no negative eigenvalues) then $k(x, x')$ is still a dot product, but in a transformed space $k(x, x') = \Phi(x)^T \cdot \Phi(x')$. **A very good property of SVM is that the usage of kernels guarantees a convex weight optimization problem.** In practice, we can always find the global optimum (no local optima). All of this at the **expense of very small computations**.

8.1 Learning phase

In the **learning phase** of SVM we need to define three things

- **Kernel** To choose a kernel we don't have a general approach, because it is **highly dependent on the problem formulation**. For this reason is here that you have to put the knowledge about the problem and your experience.
- **Training subset** We will see how this is implicitly obtained by choosing the weights. In short, if a weight relative to a sample is zero, it will be excluded from the subset and all the ones with weights different from zero are called **support vectors**.
- **Weights** The weights are calculated maximizing the margin.

Now we don't have a clue on what the margin is. Let's define it. To better understand the concept consider the figure below. We want to find the line that separate the two classes (+) and (-). We have found two decision boundaries corresponding to the red and blue lines (actually there are infinite number of boundaries). Which is the better solution? One can argue that the blue solution is better because is more "centered" between the two classes. In a more formal way, the **blue line have a greater distance to the nearest point comparing to the red line**.



2-dimensional input space with two classes (+) and (-).
Both red and blue line are linear separators of the two classes

We have to consider the fact that we can have **noise in the measuring of our information (i.e. on training examples)**. So there is the probability that the points are not precisely positioned, so if I perturb the points there may be a boundary that have an higher risk of miss-classification, like the (+) point in the bottom left of the figure considering the red boundary. For this reason is **better to consider the linear boundary that maximizes the minimum distance w.r.t. all the points in the training set**, allowing the classifier to be more robust to noise because all the points of the training set that are close to the boundary are as far as possible.

Definition 8.1 (Margin). *The margin is the minimum distance between the decision boundary hyper-plane and the nearest point.*

in other words the distance of the closest point to the hyper-plane. In formulas we have

$$\text{margin} = \min_n(t_n(w^T\Phi(x_n) + b)) \quad (70)$$

where the separating hyper-plane equation is $w^T\Phi(x_n) + b$, and to compute the distance to a point of the training set to the hyper-plane, is enough to substitute the value of the example

x_n . The multiplication for t_n is due to a problem of sign of the distance, so better than using the absolute value ($|w^T\Phi(x_n) + b|$), that is a non-linear operator, is better to multiply for t_n , indeed under the assumption that the hyper-plane perfectly classify every sample multiplying for t_n always returns a positive quantity. As we have seen in section 1.1.1, the distance between a point and the decision boundary is expressed as $\frac{y(x_n)}{\|w\|_2}$. In our case $y(x_n)$ is simply $w^T\Phi(x_N) + b$. We can see that $\|w\|_2$ doesn't depend on x_n , so it can be dropped for points comparison. Furthermore, we can, for the moment, make the assumption that all the points are classified correctly by the boundary. Knowing that, we can **multiply $y(x_n)$ by t_n , to ensure that the margin is always positive.**

NOTE $|w^T\Phi(x_n) + b|$ is not a euclidean distance indeed it has to be normalized by $\|w\|_2$, so $|w^T\Phi(x_n) + b|/\|w\|_2$ is the euclidean distance of x_n with respect to the boundary we have defined by the weights w and the feature Φ .

To find the **optimal weights** we need to **maximize the margin**.

$$w^* = \underset{w,b}{\operatorname{argmax}} \left(\frac{1}{\|w\|_2} \min_n(t_n(w^T\Phi(x_n) + b)) \right)$$

indeed I want the **weights that maximize the distance of the closest point, so it is a max-min problem**. Another reason why we removed $\|w\|_2$ from the margin, is to ensure that it later reappear in w^* formulation. The **direct solution computation** from the formula above is **very complex, because nesting a minimization inside a maximization is very computational intensive**. So we need to consider an equivalent problem that is easier to be solved. Another reason why the **complexity is high**, it is due to the fact that an hyper-plane (decision boundary) can be expressed in an infinite number of ways²⁵. We can solve this by **fixing the scale of the parameters, which means fixing the margin**. We do so by imposing the margin equal to 1. Doing so, we are sure that only one combination of w will satisfy this condition. This solves also the min-max nesting, because in the **new problem formulation we can drop the min computation and consider the margin as a constraint where $\text{margin} \geq 1$** .

NOTE As we remember from the perceptron what define the position of the hyper-plane is not the module of the vector w but only the direction, so scaling the weights that define the hyper-plane we are not defining another hyper-plane. Hence this means that we have one degree of freedom that we can use as we want. In particular in the perceptron we chose the learning rate equal to 1, since it was simply a scaling factor and we could select a value useful for us. Here, since the magnitude of w is not changing the orientation (indeed it does not appear in the quantity that define the distance from the hyper-plane $w^T\Phi(x_n) + b$) we can fix the scaling factor. This is done by **fixing the non-normalized margin to 1**. This choice does not mean that the solution has the minimum euclidean distance equal to

²⁵For example $3x_1 + \frac{1}{2}x_2 = c$ is the same hyper-plane as $6x_1 + x_2 = 2c$

1, because maybe it does not exist, but it will have the margin equal to 1. Since now the length of the margin have been chosen we can rephrase the problem as follows.

NOTE The problem is that among the infinite values that w can have, and define the same hyper-plane that minimizes the margin, I have to choose only one scaling w . Hence, I choose the one that puts the margin equal to 1. The space can be zoomed in or out, so when I decide the value to which I want to fix the margin I am blocking the zoom for a certain value. In this way the value of the l_2 norm of w will be related to the choice of the hyper-plane implied by the choice of fixing the margin equal to 1.

NOTE Remember that w define a vector that is perpendicular to the hyper-plane, so its scaling does not change the orientation of the hyper-plane itself, so they are all associated to the same solution. **What I am saying is choose the value of w such that the margin is 1, so the distance of the points from the hyper-plane is at least one.** Of course by normalizing for w the distance, by changing w the distance does not change, but if I do not normalize changing w just means we are looking at the problem at different scale, so by fixing the margin to 1 I am **choosing the scale for which the margin (not normalized is equal to 1)**. So it is **a way to choose one of the infinite solution that I have**. In other words by putting the margin equal to one we define a visibility region inside which we minimize the l_2 norm squared of w .

NOTE At the moment we are assuming the problem to be linearly separable, i.e. exist a solution to the problem. After we will see what happens if that is not the case.

From this considerations we can formulate the new problem. First, we can eliminate the margin (i.e. the minimization) from w^* (indeed we are fixing it to 1).

$$w^* = \max_w \left(\frac{1}{\|w\|_2} \right),$$

where $t_n(w^T \Phi(x_n) + b) \geq 1$

the \geq depends from the fact that the minimum distance is fixed to 1, so the distance of the points of the training set is at least one. For notations purposes, we switch from a maximization problem to a minimization one. We also slightly modify it to have simpler calculi later on

$$\begin{aligned} w^* &= \max_w \left(\frac{1}{\|w\|_2} \right) \\ &= \min_w \left(\|w\|_2 \right) \\ &= \min_w \left(\frac{1}{2} \|w\|_2^2 \right) \end{aligned}$$

$$= \min_w \left(\frac{1}{2} \|w\|_2^2 \right)$$

Finally the new problem is

$$\begin{aligned} & \text{Minimize}_w \quad \frac{1}{2} \|w\|_2^2 \\ & \text{Subject to} \quad t_n(w^T \Phi(x_n) + b) \geq 1, \quad \forall n \end{aligned}$$

At this point we are still in the primal problem, a parametric one, so we need to rewrite it in the dual representation, making the weights disappearing and the dependence will be on the training samples and on the kernel. To do this we have to recall the basis of constrained optimization.

Note - Constraint optimization basics Suppose we have

$$\begin{aligned} & \text{Minimize}_w \quad f(w) \\ & \text{Subject to constraints} \quad h_i(w)(= 0), \quad i = 1, 2, \dots \end{aligned}$$

Where $f(w)$ is a convex function. If f and h_i are linear we can use linear programming, but in our case the constraints are linear in w but the function to minimize ($\|w\|_2^2/2$) is actually quadratic (is the sum of the square of the components). In this case, we need to use quadratic programming. To solve this we use a Lagrangian formulation using the Lagrange multiplier. We will give only an intuition of the method, because the complete formulation is outside the scope of this summary. In practice, we want to transform a constrained optimization problem into an unconstrained optimization problem, where the constraints are encoded into the objective function as follow

$$L(w, \lambda) = f(w) + \sum_i \lambda_i h_i(w) \tag{71}$$

The λ_i s are called Lagrange multiplier and $L(w, \lambda)$ is called Lagrangian function. From the constraint theory, we know that to calculate the optimal solution, we need to find a point that satisfies

$$\nabla L(w, \lambda) = 0$$

hence we are optimizing both the objective function plus the value of the constraints multiplied for new parameters λ_i , over w and λ . We can see that now we need to find both the optimal w and optimal λ . Let's see an example to better understand what's going on. Suppose we have

$$\begin{aligned} & \text{Minimize} \quad \frac{1}{2}(w_1^2 + w_2^2) \quad (\sim \frac{1}{2} \|w\|_2^2) \\ & \text{Subject to} \quad w_1 + w_2 = 1 \end{aligned}$$

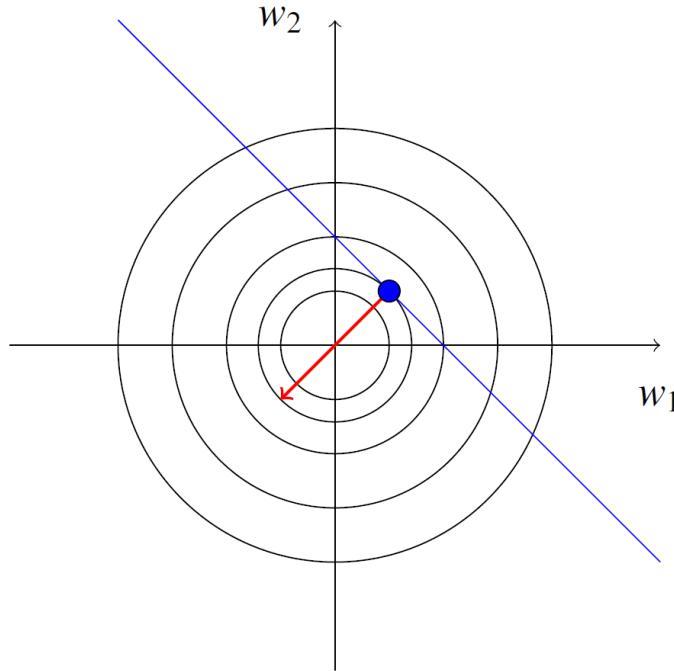
The only feasible solution are the one on $w_1 + w_2 = 1$ that is represented by the blue line in the below figure, so the origin point cannot be returned. What I have to do is to **define a new unconstrained problem using the Lagrangian function** would be

$$L(w, \lambda) = \frac{1}{2}(w_1^2 + w_2^2) + \lambda(w_1 + w_2 - 1)$$

So we find the gradient of $L(w^*, \lambda^*) = 0$ both w.r.t. w and λ (there are as many w s as parameters and as many λ s as constraints)

$$\nabla L(w, \lambda) = 0 \rightarrow \begin{cases} \frac{\partial L}{\partial w_1} = w_1 + \lambda = 0 \\ \frac{\partial L}{\partial w_2} = w_2 + \lambda = 0 \\ \frac{\partial L}{\partial \lambda} = w_1 + w_2 - 1 = 0 \end{cases}$$

Dual: $\begin{cases} w_1 = \frac{1}{2} \\ w_2 = \frac{1}{2} \\ \lambda = -\frac{1}{2} \end{cases}$



Black line are the isoline of our objective function (paraboloid so increases for higher ws).

The blue line is our constraint $w_1 + w_2 = 1$.

The red arrow is the gradient of the objective function

The blue point is the solution $(1/2, 1/2)$

We can notice from the figure above, that the gradient of the objective function in the optimal point is orthogonal to the constraint. **The only point that can be optimal is the one for which the gradient of the original objective function is orthogonal to**

the constraint, indeed this means that **being in the optimal point would mean that I cannot move towards point with lower value since the component of the gradient along the constraint line is null**, which is not true for the other points of the constraint which have gradient pointing towards the origin, so with a non-null component along the constraint. Having a gradient perpendicular to the constraint is the same as saying that the constraint is tangent to a isoline. We know that the solution of the optimization problem must lie on the constraint. **If we consider the tangent point, we can see how moving along the constraint will surely get to higher values of the objective function.** This is **due to the convexity of the objective function**.

NOTE λ is only an **auxiliary parameter used to find the solution of the constrained problem**. Furthermore, another use of λ is to write the dual formulation. Indeed **since w_1 and w_2 can be expressed using λ**

$$\begin{cases} w_1 = -\lambda \\ w_2 = -\lambda \\ w_1 + w_2 = 1 \end{cases} \implies L = -\lambda^2 - \lambda$$

we obtain that the Lagrangian function is $\frac{1}{2}(\lambda^2 + \lambda^2) + \lambda(-\lambda - \lambda - 1) = -\lambda^2 - \lambda$, only in function of λ .

This is not exactly what we were looking for, because the **constraint in the example was an equality and not an inequality as in our case**. We need to expand our formulation to handle the inequalities. Suppose to have

$$\begin{aligned} & \text{Minimize} && f(w) \\ & \text{Subject to} && g_i(w) \leq 0, \quad i = 1, 2, \dots \\ & && h_i(w) = 0, \quad i = 1, 2, \dots \end{aligned}$$

We have to define a **Lagrange multiplier α_i for the inequalities**. To find the optimal solution we can exploit the **KKT conditions (necessary conditions for being an optimal solution**, i.e. optimal points satisfy this conditions but also other non-optimal points satisfy these conditions because is a necessary condition)

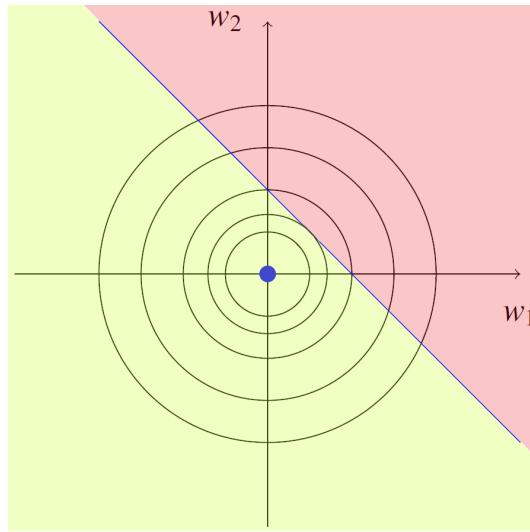
$$\begin{aligned} \nabla L(w^*, \alpha^*, \lambda^*) &= 0 \quad \text{with } L = f(w^*) + \sum_i \alpha_i^* g_i(w^*) + \sum_i \lambda_i^* h_i(w^*) \\ h_i(w^*) &= 0 \quad \text{equality constraints} \\ g_i(w^*) &\leq 0 \quad \text{inequality constraints} \\ \alpha_i &\geq 0 \\ \alpha_i^* g_i(w^*) &= 0 \end{aligned}$$

where w^* , α^* and λ^* are the optimal values. The **most interesting constraint** are the last two:

- $\alpha_i \geq 0$ says that all the Lagrangian multipliers of the inequality constraints need to be not negative.
- $\alpha_i^* g_i(\mathbf{w}^*) = 0$ the Lagrangian multiplier of the inequality constraint times the function of the inequality constraint must be equal to zero.

Particularly important is the **last one** and it is called **complementarity condition**. We can have two cases:

- $\alpha_i^* = 0 \wedge g_i(w^*) \leq 0$. If $\alpha_i^* = 0$, $g_i(w^*)$ can assume infinite different values (≤ 0 due to the other necessary condition)

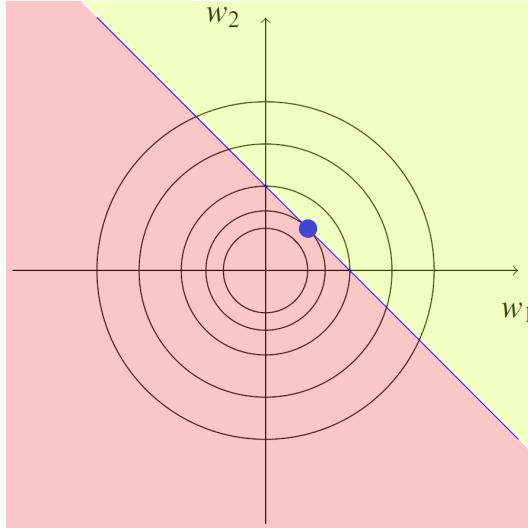


so if the optimal w is inside the region of the inequality constraint, then the Lagrangian multiplier must be 0. In the previous example:

$$\begin{aligned} \text{Minimize } & \frac{1}{2}(w_1^2 + w_2^2) = \frac{1}{2}\|\mathbf{w}\|_2 \\ \text{Subject to } & w_1 + w_2 \leq 1 \end{aligned}$$

We can see from the picture above, that the solution is in the global optimum. This is due to the fact that the **constraint doesn't play any role in limiting the solution**. In fact, we can notice that $g(w^*) = -1$. As a consequence, we know that $\alpha = 0$, **this totally make sense because the constraint is useless**.

- $\alpha_i^* \geq 0 \wedge g_i(w^*) = 0$. If $g_i(w^*) = 0$, α_i^* can assume infinite different values (≥ 0 due to the other necessary condition)



so the **optimal w** is on the boundary of the constraint since $g = 0$, and the weight (α) is not 0. In the previous example:

$$\begin{aligned} \text{Minimize } & \frac{1}{2}(w_1^2 + w_2^2) = \frac{1}{2}\|w\|_2 \\ \text{Subject to } & w_1 + w_2 \geq 1 \end{aligned}$$

Here the solution is no longer in the global optimum, but **lies on the constraint**. In fact, we have that $g(w^*) = 0$ and $\alpha \geq 0$.

Hence, when the solution lies on the constraint boundary like in the case (b), it is called **active constraint**. When a constraint is active its **Lagrange multiplier** is **positive**, indeed it means that the constraint is **really constraining the solution**. On the other hand, if the solution is inside the region defined by the constraint, its Lagrange multiplier will be 0. If the constraint is not active, removing the constraint the solution does not change, instead if the constraint is active it means that it may be influence the optimal solution, indeed if we remove the constraint the optimal solution may change. So the **complementarity condition** says that either a constraint is active ($g(w^*) = 0$) or its multiplier is zero ($\alpha^* = 0$) i.e. is useless.

NOTE As we can guess the **active constraints** are directly connected to the concepts of support vectors. Indeed as **only active vectors** are constraining our optimization problem, support vectors are the only training examples meaningful.

8.2 Dual representation

Until now the problem was over w (weights over the features) so it is the primal. We solve the equations for w as a function of the lagrangian multipliers and substitute in the Lagrangian

function resulting a problem of optimization over the Lagrangian multipliers α , that is the dual. The problem is now function of the coefficients that multiply the inequality constraints. If it is easier we can solve the dual instead of the primal. So as in the kernel methods seen, in SVMs:

- the primal problem is over feature weights
- the dual problem is over instance weights, so α will be the weights over the instances of our training set.

Furthermore, the nice effect of SVMs is that the solution over the dual problem will have a lot of zero weights α , in particular the weights over the samples over the non-active constraints.

Now we have a way to solve our problem. We can observe that we have a constraint for every training sample. Now comes the very interesting part. Every sample related to a constraint with positive Lagrange multiplier will be in the support vectors set. Hence, through weights optimization, we have found the training data subset to use for finding the solution, as we anticipated early in this section.

The optimization problem we have defined so far is called the **primal**. What we have here is still a parametric method with parameters and features.

Primal

$$\begin{aligned} \text{Minimize } & \frac{1}{2} \|w\|_2^2 \\ \text{Subject to } & t_n(w^T \Phi(x_n) + b) \geq 1, \quad \forall n \end{aligned}$$

Now our objective is to find its dual kernel representation. Let's consider the Lagrangian function of our primal problem²⁶

$$L(w, b, \alpha) = \frac{1}{2} \|w\|_2^2 - \sum_{n=1}^N \alpha_n (t_n(w^T \Phi(x_n) + b) - 1) \quad (72)$$

where the constraint are applied to all the example of the training set, indeed the summation is done over the multiplication of the weight of the example and the constraint associated to the example x_n . As we did before, we put the gradient of L equal to zero, with respect to w and b ²⁷.

For w we have,

$$\frac{\partial L(w, b, \alpha)}{\partial w} = \frac{1}{2} 2w - \sum_{n=1}^N \alpha_n t_n \Phi(x_n) = 0$$

²⁶Here we have a minus sign. The sign change is due to the fact that the constraint must be \leq

²⁷ $\frac{\partial \|w\|_2^2}{\partial w} = \frac{\partial \sum_{n=1}^N w_n^2}{\partial w} \rightarrow \frac{\partial}{\partial w_j} \sum_{k=1}^N w_k^2 = \sum_{k=1}^N \underbrace{\frac{\partial}{\partial w_j} w_k^2}_{\begin{array}{l} =0, \text{ if } j \neq k, \\ =2w_j, \text{ else} \end{array}} = 2w_j$. It follows that $\frac{\partial \|w\|_2^2}{\partial w} = 2w$

$$\implies w = \sum_{n=1}^N \alpha_n t_n \Phi(x_n)$$

For b we have,

$$\frac{\partial L(w, b, \alpha)}{\partial b} = \sum_{n=1}^N \alpha_n t_n = 0$$

The **formulation of w is equal to what we have seen in the perceptron case** at the beginning of this chapter. Now, we **replace the new formulation of w in the Lagrangian function $L(w, b, \alpha)$** obtaining

Dual

Maximize $\tilde{L}(\alpha) = \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m t_n t_m k(x_n, x_m)$

Subject to $\alpha_n \geq 0, \quad \text{for } n = 1, \dots, N$

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

In this new formulation we don't have **neither parameters w nor features $\Phi(x)$** . It is an **instance based formulation**, indeed it contains the **weights over the examples (α)**, the target value of the examples (t) and the similarity between all the pairs of the examples in our training dataset (k). Hence we want to find the weights of the examples α under the constraints:

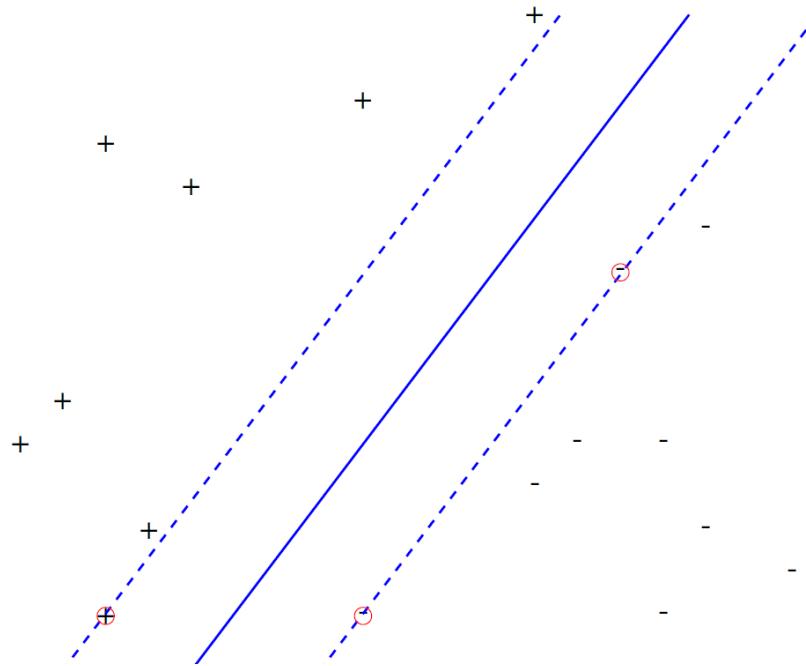
- $\alpha_n \geq 0$ derives from the KKT conditions.
- $\sum_{n=1}^N \alpha_n t_n = 0$ derives from the gradient w.r.t. to b . It means that the sum of all the weights of the points classified positively ($t_n = +1$) must be equal to the weights of the points classified negatively ($t_n = -1$). This constraint is due to the fact that this two sets of weights have to **put the separating boundary in equilibrium**, hence if the sum is zero is **not unbalanced towards the positive class or towards the negative class**. Indeed it comes from the bias b that controls the position of the boundary.

Now this **new problem is all instance based**, i.e. no more features and weights of features, is **still quadratic**, because we have $\alpha_n \cdot \alpha_m$, while the constraints are linear in α . Again we have the advantage of working in the kernel space, so there is **no need to explicitly build the feature vector associated to the points**.

Hence, the **problem of maximizing the margin that is a constrained quadratic problem**, can be rephrased using the Lagrangian function. The latter is associated to the primal problem. By putting the gradient of the Lagrangian function equal to 0 both for w and b , and replacing the expression of w we **obtain the dual problem** that is still a **quadratic optimization problem no more in w but in the parameter α** , one for each of the training examples, with two constraint.

8.3 Prediction

It's worth recalling what SVMs are doing in the input space. SVMs are linear classifier, which operates in the space defined by the features correlated to the kernel we have chosen. This highlights even more the importance of the kernel selection, because if the feature space we are defining by the kernel is not linearly separable, SVMs will fail to generate correct predictions. Another crucial point in the SVM is the optimization of the weights α . This process is performed by applying the Lagrangian method we have seen before on the dual problem representation. Once we have α we can start the prediction phase (remember that α are not user defined, but analytically computed solving the dual problem of constrained margin maximization). As we said various time before, α defines which training samples become support vectors by giving each samples a weight. If the weight is zero, the training samples will be ignored in the boundary computation. This is a behaviour we want. Let's see an example to better visualize the problem.



Blue line: Decision boundary, Dashed lines: Margin, Red circles: Support vectors

In the figure above we can see an important result. All the samples that don't lie on the margin are not contributing to the boundary, so even if removed the boundary we find is does not change. So all the support vector, i.e. vectors that influence our constrained problem, must lie on the margin. Suppose to delete a (+) sample on the top left corner. Should the boundary be changed? No, because that point doesn't affect the separation between the two classes. In practice, every samples is interpreted as a constraint. If the region they define includes the solution, the constraint is not active, and so the sample can be discarded. This is what we meant before with sparser input space. We are keeping only the samples which can contribute the most to the

boundary, all the others are discarded. The ratio between the support vectors number and the number of training samples can give us some information about the performance of the SVM. The less support vectors we have the better. For example, if all the samples are also support vectors the SVM is strongly overfitting. The sparsity gives SVMs more robustness to noise and outliers, because they will be ignored.

NOTE As we saw, for the complementarity condition the point that are not active have a zero weight in the optimization vector, instead the one that are active have a **weight greater or equal to zero, which means that are the only one to contribute in the prediction.**

NOTE Support vectors define the position of the hyper-plane, all the other points are useless, all the other points are not involved in the positioning of the hyper-plane and the absence of one or more of them won't change the boundary hyper-plane.

NOTE Is now more clear how the constraint $\sum_n \alpha_n t_n$ that balances the weights is creating the equilibrium between positive and negative samples that lie on the margin. Indeed, the weights that do not belong to support vector will be null, so the constraint is balancing the weight of the support vectors, making the hyper-plane to be positioned in the position where is more robust from noise, maximizing the margin.

The classification of new points is performed as

$$y(x) = \text{sign} \left(\sum_{n=1}^N \alpha_n t_n k(x, x_n) + b \right) \quad (73)$$

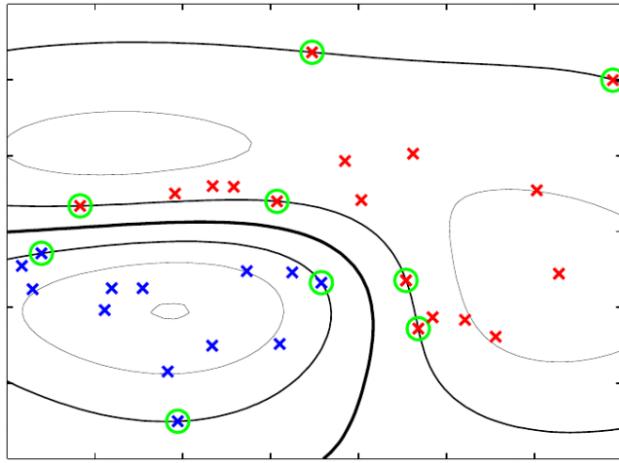
found **using the quadratic optimization solver**. Since a lot of points, by construction, have a weight that is zero, **many terms disappear** from this summation, and for this reason the SVM is a **sparse method**. This closed formula is very similar to the perceptron case. Better analyzing **the formula** what it does is it takes the support vectors weights, multiply them for their target value and multiply for their kernel with the query point x . If the query point is on the boundary (i.e. the separating hyper-plane) the summation will be null.

NOTE All the points on the hyper-plane have $\sum_{n=1}^N \alpha_n t_n k(x, x_n) + b = 0$. This is possible since the evaluation is done on new data, not used during "training".

NOTE All the weights of the support vector associated to the positive class must be equal to the weights associated to the support vector associated to the negative class, due to the optimization constraint $\sum_{n=1}^N \alpha_n t_n = 0$

Remember The **kernel function** encodes the similarity of two input points, **measured in the correlation of the target of the points**.

NOTE SVMs perform, in this case, linear classification in the feature space, that being a kernel method is the **feature space implicitly defined by the choice of the kernel function**. The features are the ones whose scalar product define the kernel function. For instance using a Gaussian kernel function, that is just like working in a feature space with infinite number of features, after we have found a linear classifier and we want to project back to the **input space we can achieve a non-linear boundary**, still having support vectors that define the position of the boundary.



hence, as it was for linear classifiers, if we use kernels that maps the results in the feature space we can **learn arbitrarily complex separating surfaces**, indeed **we are linear in the feature space and not in the input space**.

We can notice that **from the optimization problem b remains uncomputed**, indeed b **does not appear in the Lagrangian function that we optimize**. Be aware, that **we don't have a fast way to find the optimal value for b** . We can estimate it with

$$b = \frac{1}{N_S} \sum_{n \in S} \left(t_n - \sum_{m \in S} \alpha_m t_m k(x, x_m) \right)$$

where $\sum_{m \in S} \alpha_m t_m k(x, x_m)$ is the computation of the support vector machine, present of course in the prediction formula, and so $t_n - \sum_{m \in S} \alpha_m t_m k(x, x_m)$ represent the **error of the prediction**. Hence b represent an **average of the error of the prediction**, working as a **correction term computed after the weights α** , added in the prediction formula to correct the bias.

NOTE N_s is the number of support vectors, and is usually much smaller than N .

8.4 Curse of dimensionality

Sadly, as for every method we have seen so far, the **curse of dimensionality** hits also the SVMs. When the **number of dimensions increases, the percentage of support vectors increases too**, indeed to fix the position of the boundary you may need a lot of support vectors. This **leads to**:

- **poor generalization guarantees**, indeed **an high percentage of support vector can be related to overfitting**. Indeed the boundary is shaped by the support vectors (as it is clear from the Gaussian kernel example picture), so if a great percentage of samples are support vector the boundary has high degrees of freedom such that the boundary will interpolate tend to interpolate the noise of the training set following the high number of support vectors that fix the position of the boundary.
- some **scalability issue**, indeed it can be very **expensive to compute the solution** since we are talking about **a memory-based method**. SVM are a sparse method so the **non-active points could be discarded**, but if the **percentage of support vectors increase** the **advantages vanishes**.

We have seen how we are able to construct bounds about the loss function of our models. The margin bound of SVM is affected by VC dimension, and there are many theoretical results that clarify this. The **bound on VC dimension decreases with the margin, the larger the margin, less capacity to overfit and so less VC dimension**. Furthermore the **margin bound is quite loose**. But, for SVMs exists a very **handy way to construct a bound on the error** called **Leave-One-Out Bound**. As for classic LOO, we use $N - 1$ samples for the training phase and one sample for validation. We repeat this operation N times. We have already observed how **eliminating a sample from the training set, which is not a support vector, doesn't change the solution**. In this cases we will not miss-classify any of the validation points, so the cross validation error would be 0. When we "leave out" a support vector, in the **worst case** we can miss-classify a point. Hence if we want an **upperbound (worst case) of the classification error of the SVM** we can simply use the **percentage of the support vector over the number of training examples**, i.e. the **loss function will be upperbounded by the probability of miss-classify a point**, which in our case is ²⁸

$$L_h \leq \frac{E[|\mathcal{S}|]}{N} \quad |\mathcal{S}| \text{ is the number of support vectors} \quad (74)$$

Hence the desired situation is to have a very low number of support vectors with respect to the number of samples. In the case we have a low number of support vector

²⁸ $|\mathcal{S}|$ is the number of support vectors

is good both for computational reasons (memory-based method) and for the robustness of the solution (low error). Instead if almost of the examples are support vectors it means that you are overfitting, indeed it means that as you remove one sample the solution changes, i.e. a symptom of high variance and high variance means overfitting. So:

- Small number of support vectors means robust solution i.e. high generalization, and low computational effort
- Large number of support vectors high variance, i.e. risk of overfitting, and high computational effort

This bound have several computational advantages. It's like performing LOO, but without the need to train at each iteration the model, because we already know for which samples we will have a potential loss, so we have directly the upperbound of the loss looking at the percentage of support vector as indication of the generalization capability of SVM. This eliminates the major problem of LOO, but keeps the fact that is the less biased way to perform cross-validation.

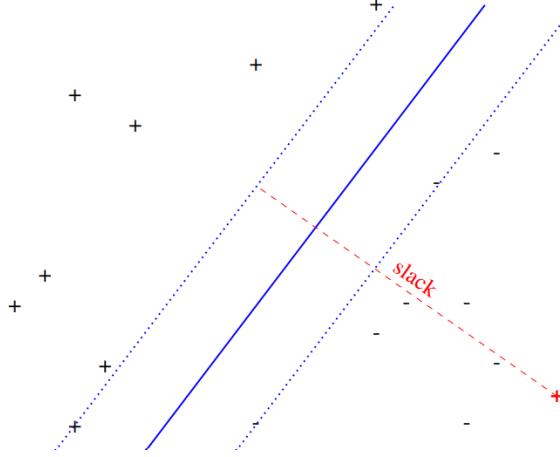
Note - Solution technique We have seen previously how we have to solve the quadratic problem of finding the weights α . It can be done using generic quadratic programming solver but they do not scale very well with a lot of samples (indeed million of samples means millions of constraints). There are more efficient specialized optimization algorithms to calculate the weights. For example SMO (Sequential Minimal Optimization). The simplest idea would be to update the weights of each examples one at a time independently, instead of calculating all the weights at the same time. But if we calculate one α_i at the time we would violate the equality balancing constraint being the new summation different from zero after the single update. So we can find them in couples. We can apply the following iteratively until convergence,

1. Find example x_i that violates KKT conditions (e.g. associated to a value of the gradient different from 0, or do not satisfy the complementarity condition).
2. Select second example x_j heuristically, that non-necessarily violates the KKT conditions. The heuristic has to suit the problem, indeed the number of couples is quadratic in the number of samples, so is better to choose the correct one to obtain a faster convergence.
3. Jointly optimize α_i and α_j in order to satisfy the constraints.

8.5 Handling Noisy data

So far we have assumed that the data are always linearly separable, i.e. we were able to correctly classify everything. This assumption was encoded in the fact that the

margin was always greater than one. Now we want to allow some samples to have a margin smaller than one, in some cases even negative. Obviously we want to minimize this behaviour, giving this points a penalization. At the same time we will relax our assumption in order to handle noisy data. The **penalization** is given by the slack variables ξ_i .



Using the previous formulation on this problem, it won't be able to find the solution since the visibility region is empty, i.e. there isn't a hyper-plane that satisfy the constraints, so there is no solution to the problem indeed until now we did not let a point to be miss-classified. **Now we have to reformulate our primal problem to include the slack variables, allowing to violate the margin constraint, but adding a penalty.**

$$\begin{aligned}
 & \text{Primal} \\
 & \text{Minimize} \quad \frac{1}{2} \|w\|_2^2 + C \sum_i \xi_i \\
 & \text{Subject to} \quad t_i(w^T \Phi(x_i) + b) \geq 1 - \xi_i, \quad \forall i \\
 & \quad \quad \quad \xi_i \geq 0, \quad \forall i
 \end{aligned}$$

In practice ξ_i shift the margin relative to a sample x_i in order to respect the constraint, allowing, for some points, the margin to be smaller than one and even negative. The effect of the slack variables ξ_i is to allow some point to be miss-classified. The boundary correspond to a slack value equal to 1 for both classes (indeed the boundary is defined as $t_i(w^T \Phi(x_i) + b) = 0$). Instead if the slack value is greater than 1 the point is miss-classified since will be on the other side of the boundary (notice that in the above figure the slack of negative class is still positive for the condition but the direction is towards the positive class area). But ξ_i also appears in the minimization formula as a penalization, in order to limit the exploitation of the margin shift. The term C is a coefficient that influences how much the slack variables

(i.e. points inside the margin or even miss-classified) will **penalize the objective function**, indeed in a minimization problem we are adding a positive term. Hence C is a parameter that can control the regularity of the solution. If we put C close to zero, we are probably making a lot of mistake, because the miss-classification are not penalized. This will produce simpler models with a lot of bias and little variance. Practically we are underfitting. If C tends to infinity, we are not allowed to violate the constraints since any violation would be infinitely penalized. The risk is that no solution can be found, because the problem is not linearly separable. In practice, we can control the bias-variance trade-off by manipulating this coefficient, making the separating surface simpler or more complex. We can find the value of C with cross-validation, i.e. it is an hyper-parameter.

NOTE With $C = 0$ the constrained optimization problem becomes an unconstrained optimization problem since we are allowing miss-classification without any penalization, so the constraints do not mean anything. This will produce the $\min \|w\|_2^2$ that is $w = 0$ i.e. complete underfitting.

As we did before we can find the dual representation

Dual

$$\begin{aligned} \text{Maximize } & \tilde{L}(\alpha) = \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m t_n t_m k(x_n, x_m) \\ \text{Subject to } & 0 \leq \alpha_n \leq C, \quad \text{for } n = 1, \dots, N \\ & \sum_{n=1}^N \alpha_n t_n = 0 \end{aligned}$$

We can see how the **objective function and the balance weights constraint are exactly the same** we had considering only linearly separable problems. Furthermore, C has become the upper bound of α_n (before it was only $\alpha_n \geq 0$).

Based on the value of α_n we can have three cases

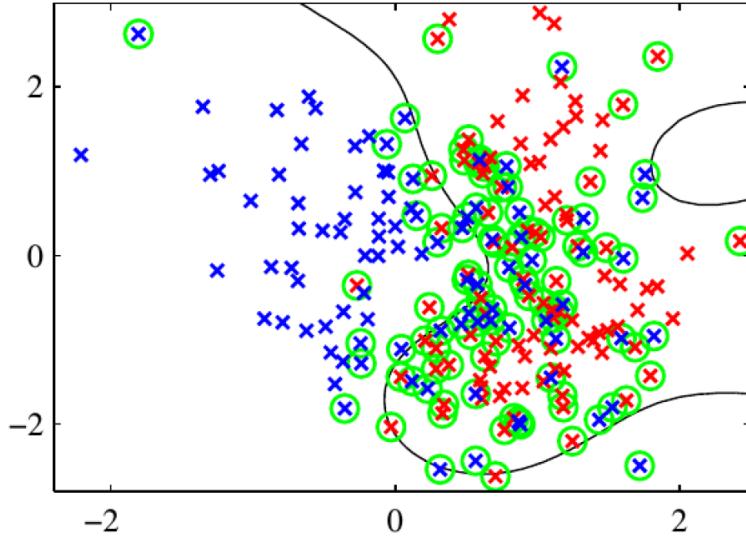
- $\alpha_n = 0$: the point associated to α_n is **not a support vector**.
- $0 < \alpha_n < C$: the point is a support vector and lies on the margin ($\xi_i = 0$)
- $\alpha_n = C$ the point lies inside the margin, and it can be either correctly classified ($0 < \xi_i \leq 1 \implies$ between the class margin and the boundary) or miss-classified ($\xi_i > 1 \implies$ after the boundary). We do not do this consideration by only looking at α_n because it depends on the slack variable value. Remember that the distance between the boundary and the margin is always 1. We know that ξ_i represents how much we shift the margin only for the sample i . If we shift it less than one, we are still correctly classifying the sample. Otherwise we are going over the boundary, and so the sample will be classified wrongly.

Considering again the consideration about the **hyper-parameter** C we did above, we can see how reducing C will force the weights α to be equal to 0, instead having a C that tends to ∞ we are allowing weights that do not let the violation of the constraints, going back to the past definition of the problem that has a solution only for linearly separable problems.

NOTE Being inside the margin means that the constraint $t_n(w^T\Phi(X) + b) \geq 1$ is not satisfied so the point is on the wrong side of the dashed line in the above figure. Then we can distinguish the situations by considering even the positioning of the point w.r.t. the boundary.

NOTE For the negative class the margin is the one below the boundary in the above figure, so the slack goes in the other direction to the one in the figure.

NOTE The value of the weights of examples inside the margin is $\alpha_n = C$ since we have to introduce a slack variable due to the fact that the margin must be corrected to make the constrain valid also for that point i.e. we are relaxing the constraint. So since to obtain a solution the constraint must be respected, the weight of this point is set to the maximum allowed value C . Hence increasing C we are increasing the weight of the possibly (if they are over the boundary) miss-classified sample in the attempt to obtain a model able to correctly classify them, which means increasing the bias i.e. overfitting the problem^(*).



Non linear classification problem with noisy data. Green circles: Support vectors

In the figure above we can see an example of **non-linearly separable problem**. We can notice that the two classes are pretty shuffled together. This may be due to a poor kernel choice. This is also reflected in the high ratio of support vectors.

NOTE A SVM that do not allow miss-classification is called **hard margin**, while the ones that allow miss-classification are called **soft margin** (like the one above).

NOTE - other SVM uses So far, we have seen the SVMs applied to classification problems. They are **not used only for classification but can also be used for**

- **Regression** (SV Regressors)
- **Ranking**
- **Feature selection**
- **Clustering**
- **Semi-supervised learning** i.e. a technique in which you have some samples for which you have the target value and some examples for which you do not have the output value and also the distribution of the examples is used to generalize better over the input space.

Theorem[section] Definition[section]

9 Markov Decision Processes

9.1 Reinforcement Learning

Reinforcement learning is learning what to do in specific case, i.e. how to map situations to actions, in order to maximize a numerical reward signal, i.e. is the part of learning concerned in the sequential decision making problem. The maximization objective is not on the immediate utility but in the long term utility, i.e. the accumulation reward along a time horizon.

NOTE For example in the number positioning game, is easy for us to change the expectations, that is a kind of learning that allow us too improve a little bit the choice policy. In this kind of game you can compute the optimal policy, but of course this take you some time to compute the probability. On the other hand by continuously playing the game simply trough experience you can learn the effect of your action, converging to play to something that is similar more and more to the optimal policy. The optimal policy is not the one that obtains each time the largest possible number (e.g. best 75421 and the optimal policy achieves 75142), indeed the optimal policy plays with a little bit of uncertainty not seeing all the five numbers together, and you can always select the best number only after seeing the 5 numbers aposteriori.

9.2 Sequential Decision Making

The **goal** of sequential decision making is to select action, i.e. **identify a policy that maps states to actions, that maximize the cumulative rewards**. The difficulty comes from the fact that actions may have long-term consequences in the sense that in the immediate the action has not a positive effect but is necessary to reach our goal, hence **immediate utility may be discarded in order to achieve much larger utility in the (uncertain) future**. So there is the credit assignment problem due to the fact that I have to find which were the actions that allowed me to achieve that goal. The learner is not told which actions to take, 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, indeed the **reward may be delayed. It may be better to sacrifice immediate reward to gain more long-term reward**. These two **characteristics—trial-and-error search and delayed reward** are the two most important distinguishing features of reinforcement learning.

Examples - Decision Making Problem

- Financial investment (may take months to mature)
- Refueling helicopter (might prevent a crash in several hours)
- Blocking opponent moves (might help winning chances many moves from now)

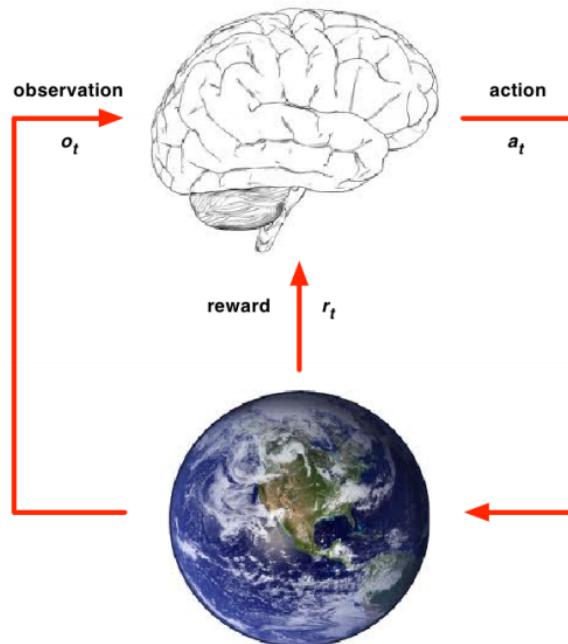
Reinforcement learning is different from supervised learning, the kind of learning studied in most current research in the field of machine learning. **Supervised learning is learning from a training set of labeled examples provided by a knowledgeable external supervisor**. Each example is a description of a situation together with a specification (the label) of the correct action the system should take to that situation, which is often to identify a category to which the situation belongs. The objective of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. This **is an important kind of learning, but alone it is not adequate for learning from interaction**. 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**.

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development.

- A master chess player makes a move. The choice is informed both by planning or anticipating possible replies and counter replies and by immediate, intuitive judgments of the desirability of particular positions and moves.

- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.
- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 30 kilometers per hour.

These examples share features that are so basic that they are easy to overlook. **All involve interaction between an active decision-making agent and its environment**, within which the **agent seeks to achieve a goal despite uncertainty about its environment**. We can also see how the problems involves sequential decision making to achieve a given goal. The **agent-environment interface** can be described as follows:



At each time step t :

1. the agents can execute an **action** a_t .
2. as a result the **environment will provide** an **observation** o_t about its state and a **reward** r_t to the agent
3. **on the basis of the observation** the **agent decide** which is the **next action** to perform.

The **effect of an action is double**:

- **changing the state of the environment**

- **getting reward signal**, a numerical signal that tells how good or bad was to take that action in that state.

The goal of the agent is to maximize the accumulation of reward, **trading-off immediate utility with the one you will get in the far future**. The interaction between the agent and the environment produces a **history**, where each step is characterized by the **action, observation and reward at a given time t** .

$$h_t = a_1, o_1, r_1, \dots, a_t, o_t, r_t$$

The history **influences what will happen next**, because future choices, and so rewards, are dependent on the past decisions. Hence it represent in some way **what the agent knows about the problem, and is the maximum information that I have of the current situation**. To predict what will happen, instead of using directly the history, we **can use the state**. Formally, the state is a function of the history

$$s_t = f(h_t) = f(a_1, o_1, r_1, \dots, a_t, o_t, r_t)$$

according to the assumption that we make in MDP, we get that **the state of the problem can be only a part of history or all the history**. With other assumption the state is a more complex function of the history. The state is defined as the minimum amount of information of the current situation that is needed to take the optimal decision.

NOTE So the first thing that we have to do working with a sequential decision making problem, is try to predict the effect of my action with respect to the reward that I can get.

Example - State vs. History Suppose you are running an experiment in a lab. You have a guinea pig (**agent**) named Bisc, which is able to pull a lever (**action**). Bisc receives a feedback of its actions by a flashing light and a bell (**environment, observations**). Based on what happens, Bisc will receive an electroshock or a tasty piece of lasagna (**rewards**). Suppose to have the following three histories,

- $h_2^1 = (\underbrace{\text{do nothing}}, \underbrace{\text{double flash}}, \underbrace{\text{null}}, \text{pull lever}, \text{bell rings}, \text{electroshock})$
- $h_2^2 = (\text{do nothing}, \text{bell rings} + \text{single flash}, \text{null}, \text{double lever}, \text{null}, \text{lasagna})$
- $h_2^3 = (\text{pull lever}, \text{single flash}, \text{null}, \text{pull lever}, \text{bell rings}, ?)$

If you were Bisc, what would you expect? The answer changes based on **how we define our state, i.e. on the definition of the model, the function you build over the history**. If our state is the last step of the history, we would expect an electroshock. Instead, if we consider as a state the number of times we have pulled the lever, we would predict that we will eat a nice lasagna. As we can see, **the state is a function of the history, but**

it may not be equal. From the state definition the ability of generalization over histories that have never seen before may change.

We can distinguish between **two states**.

- The **environment state** s_t^{env} is the environment's private representation. It's **used to produce the next observation/reward based on agent action**. It's worth mentioning, that **in many real applications** the environment state is **usually not visible**, because for example we have not many information about the problem. In the case the **agent is not able to observe the state of the environment**, but **only something that is correlated to the state of the problem** we talk **partial observability of the problem**, instead if we are able to perfectly observe the state of a problem we talk of **fully observable problem**. The **non-observability of the real state** greatly increases the difficulty of the problem. But even if s_t^{env} is visible it may contain irrelevant information.
- The **agent state** s_t^a is the **agent internal representation of the environment**. It's **used by the agent to select the next action**. It's **build upon the observation coming from the environment and can be any function of the history** ($s_t^a = f(h_t)$). Of course **may not be exactly equal to the environment state**, indeed if the agent observe the state of the problem then the state of the agent and the one of the environment are the same, in other cases where the agent observes only some part of the information of the state of the environment the agent will build an **approximation of the state of the environment**. **Ideally, the observation are enough to build a correct agent state, which can be used to behave efficiently in the environment**.

From now on we will make a very important and limiting **assumption**. The **observations that the environment provides to the agent are its internal state at a given time**

$$(t) \quad o_t = s_t^{env}$$

As a consequence the state of the agent is the same of the environment

$$s_t^a = s_t^e$$

when this happens we have a **fully observable environment**. Formally, this are **Markov Decision Process (MDP)**.

In practice the environment doesn't have hidden information. For example, chess is fully observable, because all the pieces and squares can be seen by both players. Poker is not, because a player don't know the cards of the other players.

NOTE - When RL is useful? The goal of RL in many cases is to **build a controller** which can **perform some task in an environment**. RL is **useful when the dynamic of the environment is unknown or difficult to be modelled** (e.g. trading, betting).

This is a different approach compared to the classic control theory. Historically, we build manually a model that describe the environment and the effect of an agent in it. **RL tries to learn the environment from the experience, without the need of formalizing from the beginning how the environment behaves.** Hence RL works **inductively**, learning simply by experiencing the actions, by trial and error. Another advantage of RL compared to classical control theory is the ability of **generating approximate, but yet efficient, solutions**. Hence is an advantage even if the model is available, but too complex, such that an exact solution would require too much time to be computed. For example, if we want to learn a controller to make a humanoid robot walk, an exact solution using control theory would be very complex, due to the shear amount of degree of freedom and sensors.

Examples - Sequential decision making problems Let's see some examples of SDM problems trying to classify them:

- Rubik's cube:

- the state of the problem is any disposition of the tiles of the cube. The space is discrete $\sim 4.33 \times 10^{19}$
- Actions are 12 for each state
- Deterministic state transitions, since you can deterministically predict the state of the problem since there is no randomization happening
- Rewards are -1 for each step, since the idea is to punish the agent for each moves such that the agent tries to solve as fast as possible in order to minimize the punishment received. So in this case you are solving the cube with the minimum number of moves.
- Undiscounted problem, i.e. the utility of two rewards distance in time is the same, so the value of the reward is not changed in the future.

The cube problem is quite simple for computers, but not so much to solve for humans.

- Blackjack:

- the state space is ~ 800 , or ~ 104000 if we consider even the composition of the hands
- Actions are 2 to 4 according to the state
- The state transition are stochastic since the card is drawn from the deck so it comes from some kind of probability distribution.
- the rewards depend on the outcome of the game: 0 for each step and $\{-2, -1, 0, 1, 1.5, 2\}$ at the end.
- Undiscounted since the game last a very short time.

Having a small state space is easy to solve for computers. What makes it difficult for humans is the fact of having stochastic transitions. Learning this by repeating play can be done but takes a lot of repetition, indeed in order to understand which is the best strategy you have to average the effect of the randomness intrinsic in the problem. Trying to understand which is the best action, having a lot of noise on the reward associated to this actions you have to repeat many times in order to understand the best one. Humans tend to do biased action giving more weights to negative rewards so we are not able to compute the real average, meaning that we do not behave in an optimal way. A machine instead by repetition can learn the best policy.

- Pole balancing:

- State space is a four continuous state variables $x, \dot{x}, \theta, \dot{\theta}$ i.e. the position and velocity of the cart and the angle and the angular velocity of the pole.
- Two actions $\{-N, N\}$ that correspond to the maximum force applied on the left and maximum force applied on the right.
- we can assume that if the physics is simulated it is deterministic so the state transition are deterministic. Instead in case of a real motor it is stochastic due to the noise on the sensors and actuators.
- Rewards can be 0 when the pole is in the goal position, -1 when the pole is outside the goal region and -100 when it is outside the feasible region.

- Robot navigation:

- The state space is continuous and it represents the robot coordinates.
- the actions are moving actions that express how to move the robot in the environment.
- being in a real world the states are stochastic. The stochasticity can come even from other agents that can change the configuration of the environment.
- Rewards can be set to -1 until the goal is reached, pushing the agent to reach the goal as fast as possible
- according to the problem it can be undiscounted or discounted.

Often the state cannot be observed, because in many cases the mobile robot is not able to observe the position but only what is around it, that is not enough to understand which is the position of the robot in the environment. Hence this is no more a MDP but a Partially Observable MDP (POMDP), where **since you are not able to observe the state of the environment** (position of the robot) **but only something that is related, you have to keep a probability distribution of the states that is induced by your observation**. Collecting more and more observations over the environment the distribution probability can be refined, so that you can improve the knowledge of the state.

- Web banner advertising: choose which banner ad showing in a certain slot of the web page, the best is the most profitable.
 - the state space is a single state since the choice of the banner usually does not influence the effect of the next action, since in the next time the age is requested by another user. So having a single state, it won't influence the state that comes after. Hence there is no dynamics in the state space. Sometimes I have some information about the user so in that case the problem have multiple states but again no dynamic, the chosen banner does not influence the one of the next user
 - the action is one for each banner
 - there is no dynamic in the state space
 - since each time we display the non-optimal banner we are loosing some money we want to learn by doing the least amount of trials. The performance of the banner are highly stochastic. So the reward is the probability of click times the cost per click.

These kind of problems are called (**contextual**) **Multi-Armed Bandit** problems that are a **subcase of the general RL problem**, in which you have a single state (or multiple) but your action does not influence the transition from a state to another state, but the state comes from a fixed stationary probability distribution. So the concept of exploration is important since you have to understand if an action is good or not, otherwise you cannot understand which is the best alternative. On the other hand you cannot explore indefinitely because exploring means taking action at random, and random is not so good for the reward, so is important to exploit the information you have gathered so far. If the information is not enough exploiting it can be risky. So we must find a balance of exploitation and exploration, and is of paramount importance in RL.

- Chess
 - the state space is very large $\sim 10^{47}$ configuration of the board.
 - the number of actions depending on the state ranges from 0 to 218 actions
 - the transition of the state is deterministic but is also opponent-dependent, indeed the next state you will see is deterministic but depends on how the opponent plays. So if the opponent plays with some kind of randomization the state transition model perceived by your agent is a stochastic model. Indeed depending on how the opponent plays change how the problem can be formalized. If the opponent plays with a fixed deterministic or stochastic policy you can solve the problem as a single agent MDP, but instead if the opponent is using RL actually there is a multiagent reinforcement learning problem, which is much more difficult since the environment is changing due to the fact that the opponent is changing as a function of the action you have performed.

- the rewards are 0 at each step and depending on the outcome of the game the final reward is $\{-1,0,1\}$. The reward is very delayed so it is difficult to learn which were the moves that led to the final output of the game. Usually such sparse reward function makes the problem very difficult and expensive to be learnt. Giving higher reward to some moves may introduce some bias, since in some way you are encoding your opinion on the good and bad moves, and not really what is optimal to the game, preventing of learning the optimal solution of the game.
- Undiscounted

The size of the game tree is 10^{123} .

- Texas hold'em

- The state space can be huge and not observable because you can have a lot of combination of the card and the card of the opponents are not known (partial observability)
- the actions are fold, call and raise. But the raise action imply the choice of the amount of money. In this case the number of actions are many, and makes the problem more difficult.
- the transitions are stochastic and opponent dependent, since you are drawing cards from a deck and the next state your agent will see depend on the strategy of the opponents.
- the rewards are sparse, giving 0 at each step and $\{-1, 0, 1\}$ depending on the final outcome of the turn.
- undiscounted

The difficulty of the resolution of this problem comes from the fact that the states are partially observable so the agent have to try to understand what is the distribution of the strength of the card it has. Hence you have to model the way your opponent is playing trying to make an inference on the cards it has. Bluffing introduces a randomization in order to make the process of prediction the value of the cards considering the player actions.

NOTE Passing from pure exploration to pure exploitation is very tricky, but is the key of success of all this kind of algorithms. Finding the optimal way of transitioning from exploration to exploitation. The best way to do this is to move gradually from pure exploration to pure exploitation. In general exploration cannot be totally eliminated, otherwise you cannot give guarantees, hence it goes to 0 only asymptotically.

NOTE What makes POMDP hard to solve is that when you take action you have to take into consideration that some action may help you in getting information about your state, so the policy that maximize your utility does not have to

consider only the reward but also how informative the actions are with respect to the knowledge you have of the current state.

NOTE Usually the reward function is fixed, changing through time would make the problem non-stationary, going outside the traditional MDP framework.

NOTE Discounting: in many cases you want to specify that a reward that comes very far in the future have a lower reward than the one you have currently. This is done for the reason that you do not know if that future will never come. So the discounting encodes how much you want to risk about the future instead of looking at the earnings in the present or in the close future (e.g. investment. one gives you 10% in one day or 1000% in 1000 years, so the future is too far)

9.3 MDP

9.3.1 MDP model

Here we introduce the formal problem of finite **Markov Decision Processes**, or finite MDPs. MDPs are a mathematically idealized form of the reinforcement learning problem, for which precise theoretical statements can be made. MDPs are **constructed on two principles**. The problem is **fully observable** and **Markovian**.

Definition 9.1 (Markov assumption). A stochastic process X_t ²⁹ is said to be **Markovian** if and only if

$$P(X_{t+1} = j | X_t = k_t, X_{t-1} = k_{t-1}, \dots, X_0 = k_0) = P(X_{t+1} = j | X_t = k_t)$$

This means that **the future is independent of the past given the present**.

Which means that by observing the present variables I can predict the value of the future states without the need of storing in memory the past observation. The present is enough for future prediction and the past can be forgotten, since it does not bring any information on what will be happening in the future. In other words what is **important for the prediction is the current state and not how it was reached**, i.e. the history. This can be noticed in the mathematical definition of the Markovian assumption since the conditional probability can take into account only the current state value discarding the past ones.

Example - Markovian problem A practical example of a Markovian problem is chess. To make the next move, is the current state of the board enough or we need also the previous move/states to take a decision? The current state is more than enough, because how we reached it doesn't affect the states itself and so the next moves (same for the rubik's cube).

²⁹A stochastic or random process can be defined as a collection (sequence) of random variables that, in our case, is indexed by time. We can see it as a sequential observation of a time series or model

An example of **non-Markovian problem** is black jack. Knowing only the card in your hand is not enough, because the cards previously drawn affect which cards will appear in the next rounds, so you are missing some information (same for texas hold'em).

If a problem is **Markovian**, the current state captures all the information from history. As a consequence, once the state is known, the history can be thrown away, i.e. the state is a sufficient statistic for the future. So by limiting only to the current information you can synthesize the **dynamic of the problem** (conditional probabilities) by simply specifying the transition probabilities from the current state to the future state. If the transition probabilities are stationary (like the MDPs we will analyze) we can write that:

$$p_{ij} = P(X_{t+1} = j | X_t = i) = P(X_1 = j | X_0 = i)$$

which encodes the **probability of going to the state $i \rightarrow j$ in all time steps** (since we said the probabilities are stationary). It means that the environment doesn't change through time, so the **problem is time invariant**, i.e. stationary environment (e.g. rubik's cube, black jack are the same game no matter when you play it). So the **transition probabilities can be encoded in a table that express which is the probability to go from one state to another and is valid for each point in time**.

Let's define formally what **Discrete-time Markov Decision Processes** are. A Markov decision process (MDP) is a **Markov reward process with decisions**. It models an environment in which all states are Markovian and time is divided into stages. Hence "discrete time" since the **decision are not continuously** (like in control theory) done but are **done at each fixed interval of time** and "finite" since we have a number of states and actions that are finite.

Definition 9.2 (Finite MDP). A Markov process is a tuple $\langle S, A, P, R, \gamma, \mu \rangle$

- S , finite set of states
- A , finite set of actions
- P , state transition probability matrix $P(s'|s, a)$. Describe the probability of arrive in state s' taking action a from starting state s
- R , reward function $R(s, a) = E[r|s, a]$. The immediate reward in performing action a in state s
- γ , discount factor. Weights how important are future rewards. $\gamma \in [0, 1]$
- μ^0 , set of initial probability. Describe the probability for every state to be the starting state: $\mu_i^0 = P(X_0 = i) \quad \forall i$

NOTE So the probability of reaching a state is not only influenced from the starting state, but also from the action that I take in that state. So the transition matrix is a rectangular matrix (no more a square one of the markovian assumption $|S| \times |S|$).

NOTE Usually the reward function is specified as $R(s, a, s')$, where

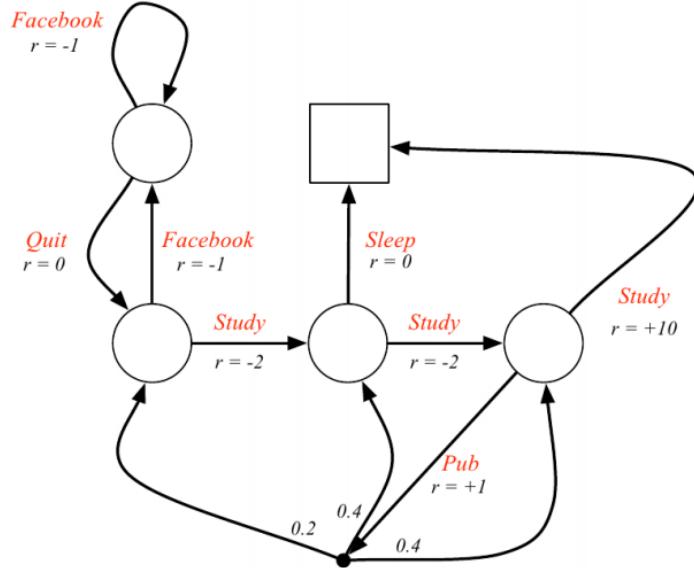
$$R(s, a) = \sum_{s'} p(s'|s, a)R(s, a, s')$$

that is the expected value with respect to the transition model p . So $R(s, a)$ is equal to the average over all the possible reachable state multiplied (weighted) for the probability of reaching that state considering that I was in (s, a) .

NOTE The discount factor tell how the reward decrease with time. If it is 1 the rewards remain fixed, instead if it is 0 the reward in future time points is null. **If the discount is 0 we are trying to maximize the immediate utility, since you do not have hope that there will be a future** (e.g. in a game you are sure that the game will stop after the first step). **Usually the discount factor is in between 0 and 1 but closer to 1, encoding the fact that you prefer to have larger rewards sooner with respect to the future.** Hence with very low γ the agent is interested only in **immediate rewards or rewards in few steps**, and if its value increase the agent want to optimize also for a longer future. It encodes how much you want to sacrifice **immediate utility looking for a much larger reward in the far future.**

NOTE The transition matrix of a Markov chain is instead squared. A Markov chain is a sequence of states determined maybe by the fact that you have fixed the way of choosing the action, i.e. in each state you know which action will be chosen. Fixing the way of behaving will produce a random process of states, so there is the need of formalizing only the probability of going to one state to another given that you have decided which action you will take in each state (in each state you know you take a certain action but you model the state you will go to). **t I can take different actions in different states.**

Example - MDP visualization Suppose we want to model a student's life during an exam session.



*Finite MDP: circles = states, arrows/red words = actions, square = goal,
 r = rewards, black dot = transition probability*

What you would like to know are what are the best action in each state. The optimal way of behaving depends also on discount factor, i.e. which function you want to optimize with respect to the future. We can represent a MDP with a directed graph. We can see how taking an action sometimes have deterministic consequences (Facebook), or non-deterministic (stochastic transition) consequences (Pub) so the probability of taking actions must be specified.

9.4 Goals and rewards

A very important phase of every reinforcement problem is goal definition. Is a scalar reward an adequate notion of goal? When we use MDPs the goal can be encoded in the reward definitions. In particular we have

Definition 9.3 (Sutton hypothesis). *All of what we mean by goals and purposes can be well thought of as the maximization of the cumulative sum of a received scalar reward.*

Sutton hypothesis is **not universally correct** (i.e. **not correct for all the problem**), but so simple and flexible we have to disprove it before considering anything more complicated for our problem. A **goal** must be shaped so to **specify what we want to achieve, not how we want to achieve it**. This is very important, because **how we achieve a given goal is not dependent on the goal itself, but depends on the environment where we are operating**. Furthermore, goal **definition must be outside the agent's direct control** (thus outside the agent). From a practical point of view, a goal can be defined with many (infinite) different reward functions (e.g. scaling or summing a constant to the reward function we are simply specifying the same problem). For example, if we consider the previous

student example, if we multiply by a coefficient every rewards, the optimal behaviour and the goal would remain the same. The **success of a reward function depends on**

- how much the reward function is shaped³⁰. Indeed if the reward function is **too sparse the learning process becomes too hard**.
- **how frequently the agent receive feedback.**

hence the **agent must be able to measure success**:

- **explicitly** with rewards
- **frequently** during her lifespan (i.e. connected to the sparsity of the reward function)

NOTE The **Sutton hypothesis is not correct for every problem**, indeed the goal sometimes is not formalized as the maximization of some kind of sum of the past rewards, but as the maximization of a more complex function. **But for most of the problem the Sutton hypothesis holds.**

9.4.1 Return

As we have said **the goal of a problem can be encoded in the reward function**. What we are **interested** in is to **maximize the return**. The return describe the **expected cumulative reward we will get from a given time t to the end of the episode**. We need two ingredients to define our return,

- **Time horizon** The time horizon define **how many step we will make until we stop an episode**. There are different cases:
 - **Finite** We perform a **finite and fixed number of steps known to the agent** at the beginning of the process (i.e. each time we take an action the time horizon diminishes of 1). The **solution found is non-stationary** since the **policy will depend on how many steps there are before the end of the game**, so the policy will **take into account explicitly how far is the deadline of the game** (e.g. choose the pitstop strategy with a certain type of tyre)
 - **Indefinite** We don't know how many steps will take to reach the goal but **we know we will reach them in a finite number of steps**. We **stop** when a stopping criteria is met (**absorbing states**). For example in the black jack game the absorbing states are stop to request a card and exceed 21, furthermore the horizon length depends on the randomicity of the card that come out from the deck. **Is possible to learn optimal policy that are stationary, so do not depend on time.**

³⁰Shaped means that the reward are **different from zero**. Shaped reward functions lead to faster learning

- **Infinite** We will perform an infinite number of steps. So for instance in the pole balancing ideally the problem is defined for infinite time, so you want to keep the pole balanced forever. **Is possible to learn optimal policy that are stationary, so do not depend on time.**

- **Cumulative reward** Define how we aggregate all the rewards from each step.

- **Total reward**

$$V = \sum_{i=1}^{\infty} r_i$$

it works well with finite problems, since you can sum the reward with no problem due to the limited horizon, and also in the indefinite case even if the length of the horizon is so long that can have instability problem. But does **not work with infinite horizon since the risk is that the sum over infinite elements it may diverge**, so you cannot estimate the cumulative utility. For infinite horizon other cumulative reward must be defined.

- **Average reward**

$$V = \lim_{n \rightarrow \infty} \frac{r_1 + \dots + r_n}{n}$$

that **if each reward is bounded the average is bounded so there is no problem of divergence**. But **estimating it is quite hard** so there are only few approaches that use this framework.

- **Discounted reward**

$$V = \sum_{i=1}^{\infty} \gamma^{i-1} r_i$$

you sum the reward at each step weighting them for the discount factor raised at the power of the number of step the rewards happen in the future ($\gamma^0 r_1 + \gamma^1 r_2 + \dots$). Being γ between 0 and 1 the **weights decrease exponentially to zero with time, giving less and less importance to the reward taken far into the future**. The value of γ defines how quick the rewards go to 0, and by taking $\gamma = 1$ we go back to (undiscounted) total reward case. Supposing that all the rewards are bounded

$$R_{min} \leq r \leq R_{max}$$

how can we bound V of the discounted reward?

$$V = \sum_i^{\infty} \gamma^{i-1} r_i \leq R_{max} \sum_i^{\infty} \gamma^{i-1}$$

but due to the fact that that is a geometric series and $0 \leq \gamma \leq 1$:

$$V \leq R_{max} \frac{1}{1 - \gamma}$$

the same can be done with R_{min} . Hence we find that the **discounted reward return can be bounded as**:

$$\frac{R_{min}}{1 - \gamma} \leq V \leq \frac{R_{max}}{1 - \gamma}$$

- **Mean-variance reward** is a more complex objective function where we are not only trying to maximize the return but also there is a minimization of the variance of the reward.

NOTE With a finite and indefinite horizon problems can be used all the types of cumulative rewards being the steps finite. Instead for infinite time horizons the total reward may lead to diverging return, so that definition cannot be used.

The most used cumulative reward is the discounted reward. We make use of the **discount factor γ to penalize future rewards**. We can give a definition of the **infinite-horizon discounted return**

Definition 9.4. *The return v_t is the total discounted reward from time-step t .*

$$v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (75)$$

hence v_t is the return obtained by certain way of selecting the actions (policy) by taking the rewards produced by the sequential decision making problem discounted, starting from the time step t . So this represent the **concept of the return i.e. the realization of the sequence of the reward summed together with the discount factor**. We know that $\gamma \in [0, 1]$. γ^k can be interpret as a measure of the importance of a reward at time $t + k + 1$. The discount factor will penalize more the rewards far in the future because it is monotonically decreasing.

- If γ is close to zero we will prefer immediate rewards, i.e. it leads to a "myopic" (miope) evaluation.
- If γ is close to 1 we will equally evaluate immediate and future rewards, i.e. it leads to a "far-sighted" evaluation.

Ideally, γ equal to one will represents the true problem, but it makes the learning process very complex from a computational point of view. Sometimes, it's worth simplifying the problem reducing γ , in order to find an optimal solution of a simplified version of the true problem. Another way to interpret γ is from a probabilistic point of view. γ could represents the probability that the process (decision making problem) will go on, so that we will play another step. On the other end $(1 - \gamma)$ represent the probability that the problem ends in the next step. We invest in future rewards, only if we are confident that we will play future steps.

The **discount factor** has **many advantages**. It is mathematically convenient to discount rewards and **avoids infinite returns in cyclic or infinite Markov processes**. From a model point of view, we can model uncertainty about the future easily. For example, we can represent animal/human behavior preference for immediate reward. Sometimes it is possible to use **undiscounted Markov reward processes** (i.e. $\gamma = 1$) for examples if all sequences terminate (e.g. rubik's cube, black jack).

The **discount factor** is a **property of the problem we want to solve (model)**(***), not a parameter of the solution, so by **changing it we would change the problem and so the solution**. Sometimes researchers **wrongly treat it as a parameter of the problem**, indeed by **reducing it we are making the problem easier** since only few steps will give a non negligible reward. Instead a discount factor **close to one makes the problem more difficult** since I have to consider the long term effect of the actions. For this reason many times the discount factor γ is used as a bias variance trade-off parameter, indeed by **taking a smaller γ you are changing the problem thus introducing a bias**, but it also has the effect of **reducing the variance** due to the fact that the problem is easier. So reducing γ you are getting a better solution for another problem that maybe is not so different to the original one and can be a good approximation.

- Increasing γ you are increasing the variance due to the fact that you are giving more importance to future time steps.
- Reducing γ you are reducing the variance since the time horizon decreases.

but since the problem is associated to a specific value of γ **using another value of γ means solving another problem different from the original one** (similar to what we have done with ridge regression starting from the original regression problem, we preferred to obtain a bias-variance trade-off by defining another problem). Here, **again (as in ridge regression) we are introducing the bias at the cost of reducing the variance**.

NOTE γ can be used as bias variance trade-off, indeed by changing it we are changing the problem (imagine that you are changing the tree in which you are looking for the solution) which means you are putting some bias in your solution; furthermore if you are reducing γ the problem is simpler (the tree depth is smaller) so the variance of the solution is smaller. Instead giving importance to much more steps in the future (tree much more deep) so the estimation will have a lot of variance. **In many cases when you have few samples because you cannot have many interactions of the agent with the environment, you prefer to reduce the discount factor of the problem, i.e. simplifying the problem by considering a shorter time horizon, introducing a bias but also reducing the variance.** So with few samples you get a better performance with this new problem (that is not the original one), and **usually if the reduction of the discount factor is not exaggerated you get that the good solution of the new problem is also a good solution on the original problem, and in many cases is better than solving directly the original problem.**

NOTE Changing the discount factor γ during time is not advisable since changing γ means changing the problem, so the problem becomes non-stationary. There are a lot of theoretical result that states how if γ changes through time the problem becomes NP-hard, while the γ is constant the problem is PAC learnable (i.e. polynomial complexity).

9.4.2 Policies

So far we have defined what a MDP is, and how the behaviour of an agent can be evaluated at a given time t . Now we need to formalize the decision making process of our agent. The rules that control which action an agent will take in a given state are called policy. In other words, a policy, at any given point in time, decides which action the agent selects, so it fully defines the behaviour of the agent. A policy can be

- **Markovian \subseteq History-dependent.** Markovian means that policy depends only on current state. History dependent means that policy depends on whole history of interaction with the environment.
- **Deterministic \subseteq Stochastic.** Deterministic means that in a given state the policy will always choose the same action. Stochastic policies could choose different action in the same state
- **Stationary \subseteq Non-stationary.** Stationary policies are time invariant. Non-stationary policies will have different behaviour at different time steps.

so on the left we have the easiest case and on the right side the most complex case, i.e. the most complex policy is history dependent stochastic and non-stationary instead the simplest are Markovian deterministic and stationary. **There is always at least one optimal policy** (i.e. that achieves the best behaviour) **for any MDP that is Markovian deterministic and stationary**, maybe there are policy in the bigger class (history dependent stochastic and non stationary) that are equivalent, with the same performance. The fact that exist at least an optimal policy in Markovian deterministic stationary framework let us to focus our attention to this **class of policies which is the simplest**.

NOTE We are not interested in history dependent since we work in **Markovian framework**. Non-stationary policy are only really needed in finite horizon problems, but we focus on infinite horizon problems. Even if there is always a deterministic (Markovian and stationary) policy we will consider stochastic policies, since we want to use RL, and even if its true that the optimal policy in MDP can be always been found in the set of deterministic policies to learn the deterministic policy I have to explore, which involves randomization and so RL will need to consider also stochastic policies since it will need to explore the actions and not just execute the optimal policy, that has to be learnt.

From now on, we will focus on stationary stochastic Markovian policies. More formally we have

Definition 9.5 (Stationary Stochastic Markovian policies). A *policy* π is a distribution over actions given the state

$$\pi(a|s) = P(a|s)$$

In few words, π describes the probability of doing action a in s .

Once you fix the policy π I can reason about what happens to my Markov decision process. As said before once you fix the policy of the action the MDP becomes what is called a **Markov reward process**, that is nothing else a **Markov chain** (chain that goes from one state to another according to the probability specified in the transition matrix) with rewards associated to each transitions. Indeed by fixing the policy the action disappear from the transition probabilities table, since there is no more the freedom of changing the policy i.e. changing the actions. So now we are interested to select the policy π (i.e. how the agent behave in the environment) and see what happens to the state transition and to the rewards associated to the state transitions. Hence given an MDP $M = \langle S, A, P, R, \gamma, \mu \rangle$ and a policy π

- the state sequence s_1, s_2, s_3, \dots is a **Markov process** $\langle S, P^\pi, \mu \rangle$
- the state and reward sequence $s_1, r_1, s_2, r_2, \dots$ is **Markov reward process** $\langle S, P^\pi, R^\pi, \gamma, \mu \rangle$, where:

$$P^\pi = P^\pi(s'|s) = \sum_{a \in \mathcal{A}} \pi(a|s) P(s'|s, a)$$

$$R^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) R(s, a)$$

hence we passed from a transition model P that involves action ($a \in A$) to another transition model P^π that I want to average the effect of the policy by making the action disappear. In other words since now the policy π (the probability of choosing the action given the state) is fixed which is the probability of going from each state to another state. So P^π is computed as the expected value with respect to the policy of going to each (s, a) to each state s' , i.e. $P^\pi(s'|s) = E_{a \sim \pi(\cdot, s)}[P(s'|s, a)]$. The same is valid for the reward $R^\pi(s) = E_{a \sim \pi(\cdot, s)}[R(s, a)]$. Thus $P^\pi(s'|s)$ is a **square transition matrix** (number of states x number of states), and $R^\pi(s)$ is a **reward vector that states the immediate reward that I can get in state s when I use the policy π** , that of course is the expected reward following the policy π .

NOTE What we did in this last part was imply to make a disappear since we computed the average with respect to the policy given the probability induced by the policy. Indeed the **Markov process was initially defined over A but then is not**.

NOTE P^π are the transition probability from each couple of state when I have the policy π and R^π are the immediate reward in each state when I have the policy π . These values are obtained by averaging on the values across the possible actions weighted by the probability of executing the action.

NOTE The transition model P at the moment is supposed to be known, then we will see what we should do when P is not known and we have to use RL to learn P . So we are seeing algorithm that solve the MDP based on the knowledge of P that are not RL algorithm. Then we will see how to proceed when we have no knowledge of P and we want to replace P with the direct experience from the agent that actually takes samples of P , and then use this samples of P to estimate the solution of this sequential decision problem. Also in cases in which P is known it can happen we may want to estimate it from the data in an approximate way because it is too complex. Hence for now we assume to have the perfect knowledge of the MDP (problem) and we see how to compute the solution; then we will see how to learn the solution having no knowledge of the transition model and the reward function, but having only data, i.e. experience of the interaction between the agent and the environment (RL).

NOTE Solving a MDP means to find a policy that maximize the objective function, i.e. the accumulation of rewards (we will focus on cumulative discounted rewards).

9.4.3 Value functions

Given a policy π (Markovian stationary stochastic), we want to evaluate the goodness, the utility associated to the policy. It is possible to define the utility of each state by doing a policy evaluation. We want to find the expected return for each state. This is very handy when trying to understand which are the optimal actions, and so the optimal policy. The utility can be formalized as

Definition 9.6 (State-Value function). *The state-value function $V^\pi(s)$ of an MDP is the expected return starting from state s , and then following policy π*

$$V^\pi(s) = E_\pi[v_t | s_t = s]$$

where v_t is nothing else than the sum of the discounted rewards ($v_t = \sum_{i=t}^{\infty} \gamma^{i-1} r_i$). Hence this is the expectation with respect to the policy that you have on how much discounted reward you will collect starting from this state s and following the policy π , which is the utility associated to the state s when you use the policy π . Each state of the problem has associated a state-value. For control purposes, rather than the value of each state, it is easier to consider the value of each action in each state

Definition 9.7 (Action-Value function). *The action-value function $Q^\pi(s, a)$ is the expected return starting from state s , taking action a , and then following policy*

π

$$Q^\pi(s, a) = E_\pi[v_t | s_t = s, a_t = a]$$

that is **both function of the state s and the action a** . The meaning is the same as before, the expected value of the discounted sum of the rewards under the policy π but not only starting from s but also taking the action a . The only difference between V^π and Q^π is that

- in V^π at each step in the state s you take action according to the probability specified by π ,
- in Q^π in the first step you take action a with probability 1 (i.e. you take the action specified by the argument of the function) and from the second step on you will follow policy π as done for V^π .

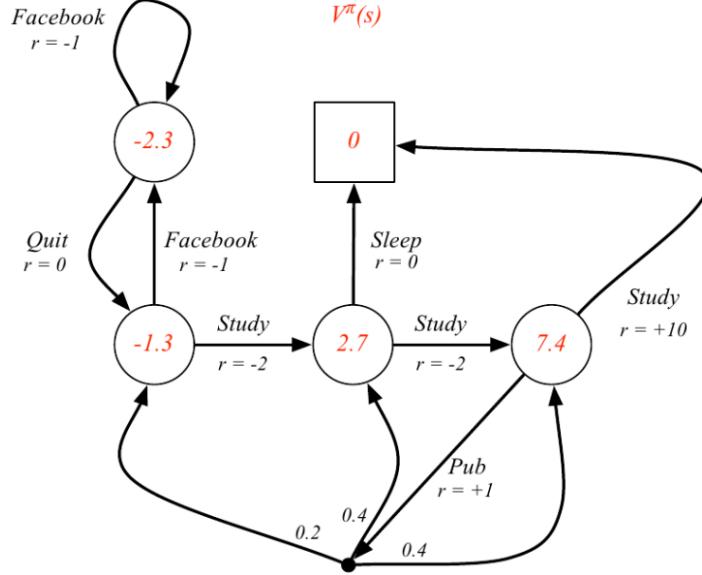
so there is a relationship between the two. It's worth mentioning that we can **calculate $V^\pi(s)$ from $Q^\pi(s, a)$, but not vice versa**. We have that

$$V^\pi(s) = \sum_a \pi(a|s) Q^\pi(s, a) \quad (76)$$

Practically, **we are doing a marginalization**³¹. The only difference between $V^\pi(s)$ and $Q^\pi(s, a)$ is in the first step. $V^\pi(s)$ follows the policy, instead $Q^\pi(s, a)$ takes blindly action a . From the second step the two value functions are the same. **If we want to find $Q^\pi(s, a)$ from $V^\pi(s)$ it would be impossible, because we don't know which action a we want to pursue at the beginning.**

Let's consider again the student MDP example. Suppose to have a random policy, i.e. $\pi(s, a) = \frac{1}{2}$ because from every state we always have to choose between two actions, and a discount factor $\gamma = 1$, i.e. is the undiscounted case. This problem can be modeled as undiscounted even if the time horizon is indefinite, because sooner or later you will reach the absorbing state where the problem ends, so is not possible to obtain the sum of infinite rewards.

³¹Given a known joint distribution of two discrete random variables, say, X and Y, the marginal distribution of either variable, X for example, is the probability distribution of X when the values of Y are not taken into consideration. This can be calculated by summing the joint probability distribution over all values of Y. $P(x) = \sum_y p(y)P(x, y)$



Is not easy to compute the value function $V^\pi(s)$ due to the cumulative utility through time, due to the several path that can be followed. Can also happen to have recursion, so that the reward in a given state depends on itself since the path start from that state but also return again on it. Indeed, to define how much return I will collect from a state you have to simulate the Markov reward process given the policy π for all the possible trajectories, to see how much reward you collect from all the trajectory and what is the probability of this trajectory to be realized. Hence, **a really naive way to calculate the state-value functions is to enumerate all the possible trajectory from a given state, and average all the cumulative rewards.** This is really inefficient, even for MDPs with few states. **A possible solution** is to use the **Bellman expectation equation**. The **state-value function** can again be decomposed into immediate reward plus discounted state-value function of successor state,

$$\begin{aligned}
 V^\pi(s) &= E_\pi[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s] \\
 &= \underbrace{\sum_{a \in A} \pi(a|s) \left(R(s, a) + \gamma \sum_{\substack{s' \in S \\ \text{all the possible next states}}} P(s'|s, a) V^\pi(s') \right)}_{\text{summing over the action to consider stochasticity}}
 \end{aligned}$$

where $R(s, a)$ is the immediate reward of taking action a in s . $\gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s')$ is the **discounted value of the successor of s** , i.e. the **utility of the state that I reach**. We can see that the **Bellman equation is recursive**, because $V^\pi(s)$ appears both on the left and right side of the equation, indeed on the right hand side we have $\sum_{s' \in S} P(s'|s, a) V^\pi(s')$. In this sum, we iterate over all the states, included the state s for which we are calculating the value function. **In this way we do not do directly the**

unrolling, which is contained into the recursion.

Our goal is to calculate the value function for every states. If we build a system of equations with all the value functions, we will have a system with $|S|$ equations. Every equation will have $|S|$ unknowns. Furthermore, the unknowns are linear. This means that if the equations are linearly independent, we can find the solution of the system in closed form. Knowing that it would be nice to have a matrix notation for that. We can start by rewriting the equation condensing some terms together.

$$R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s, a)$$

$$P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s, a)$$

$R^\pi(s)$ can be seen as the immediate reward we expect following $\pi(a|s)$ from s . $P^\pi(s')$ is interpretable as the probability of reaching s' from s in one step, following $\pi(a|s)$. Bellman equation can be rewritten as

$$V^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V^\pi(s')$$

that is called **Bellman expectation equation since is the Bellman equation for a specific policy π** . This equation can be used to compute V^π . In order to do this is better to consider the matrix notation. For every term of the Bellman equation, we can define its matrix version

$$V^\pi = [V^\pi(s_1) \dots V^\pi(s_{|S|})]^T \quad [|S| \times 1]$$

$$R^\pi = [R^\pi(s_1) \dots R^\pi(s_{|S|})]^T \quad [|S| \times 1] \quad \text{Immediate reward with policy } \pi$$

$$P^\pi = \begin{bmatrix} P^\pi(s_1, s_1) & \dots & P^\pi(s_1, s_{|S|}) \\ \vdots & \ddots & \vdots \\ P^\pi(s_{|S|}, s_1) & \dots & P^\pi(s_{|S|}, s_{|S|}) \end{bmatrix} \quad [|S| \times |S|] \quad \text{Transition matrix of the MRP with policy } \pi$$

In matrix notation we can rewrite the **system** of Bellman equation as

$$V^\pi = R^\pi + \gamma P^\pi V^\pi \tag{77}$$

that is a **system of $|S|$ equations in $|S|$ unknowns**, which are linear, hence is a linear system with $|S|$ unknowns and equations. Being a linear system we can solve it in close form isolating V^π

$$\begin{aligned} V^\pi &= R^\pi + \gamma P^\pi V^\pi \\ V^\pi - \gamma P^\pi V^\pi &= R^\pi \\ V^\pi &= (I - \gamma P^\pi)^{-1} R^\pi \end{aligned} \tag{78}$$

This is our **closed form solution to the policy evaluation problem**. Given a policy π we can evaluate the value of every state, **without the need of unrolling since it is implicitly done by the inverse matrix**. Actually the inverted matrix can be interpreted as the limit of the geometric series, obtained by this markov chain.

Existence of V^π solution To calculate V^π we need to perform a matrix inversion. **Is the inversion always possible?** Luckily yes, but **under the condition** that $\gamma < 1$. Hence under the latter condition the matrix $(I - \gamma P^\pi)^{-1}$ is always non-singular. P^π is a stochastic matrix, it means that every row sums up to 1. This can be justify by the fact that each rows represents the probabilities distribution of going from a fixed state s to every state in S . The stochasticity of the matrix ensures that its eigenvalues are between -1 and 1 (spectral radius equal to 1). If we define $A = I - \gamma P^\pi$, the eigenvalues of A will be $\lambda_i^A = 1 - \gamma \lambda_i^{P^\pi}$. If γ is less than 1 we are sure that $\lambda_i^A > 0, \forall i$. This implies that the matrix is **positive-definite and so invertible**. Hence having a discounted problem with $\gamma < 1$ we are sure that we can compute in closed form the function of a given policy by knowing the policy, the transition model and the reward.

So evaluating the performance of a policy in each state can be computed in closed form. The only drawback of this closed form is that if the number of state is **very large the complexity of this computation is cubic in the number of states** (remembering that a NxN matrix inversion is $\sim O(N^3)$). So having for example millions of states the **computational time becomes prohibitive**.

As we did before, we can **define the Bellman equation using the action-value function**

$$\begin{aligned} Q^\pi(s, a) &= E_\pi[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \\ &= R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s') \\ &= R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \sum_{a' \in A} \pi(a'|s') Q^\pi(s', a') \end{aligned}$$

where the immediate reward is the same as the one for the state-value function, but then the second term is expanded considering the relation between the state-value function and the action-value function, showing how it is still a recursive equation.

The question that we want to answer now is that if **exist other ways to compute the utility of each state for a given policy**. Of course if the problem is small we can use the closed form since it compute exactly the value function with a not so large computational effort, but instead with **bigger problems we may want to have some approximation of the computation of the value function that is less expensive**. The expression of V^π can be compacted even more using the **Bellman operator**, which is a **vectorial operator that takes as input a vector and produces as output another vector with the same size using the Bellman expectation equation**. First, we introduce its definition. Don't worry if its concept seems too abstract or not clear. We will better explain it in a moment

Definition 9.8. The **Bellman operator** T^π for V^π is defined as $T^\pi : \mathbb{R}^{|S|} \rightarrow \mathbb{R}^{|S|}$. This

*operator takes as an argument a value function and it returns a value function*³²

$$T^\pi(V) = R^\pi + \gamma P^\pi V$$

or

$$(T^\pi V^\pi)(s) = \sum_{a \in A} \pi(a|s) \left(R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s') \right)$$

i.e. the Bellman operators maps value functions to value functions. This definition alone is not enough to understand the Bellman operator. Let's go deeper. **Consider the space of all possible value functions.** In our case a **value function is defined by the values given to each state of our MDP**. Our value function can be represented as vector of length $|S|$ ($V \in \mathbb{R}^{|S|}$). As we can see from the definition, the Bellman operator maps a value function to another value function, so it is a vector operator, because V^π can be seen as a vector. Applying the operator to V^π we obtain the Bellman equation from which we obtain the following linear equation in V^π and T^π

$$T^\pi(V^\pi) = R^\pi + \gamma P^\pi V^\pi = V^\pi \quad (79)$$

$$\implies T^\pi(V^\pi) = V^\pi \quad (80)$$

We can say that V^π is the **only fixed point**³³ of the operator T^π . Hence, applying the operator T^π to the true value function of the problem V^π , the output of the operator is again V^π , i.e. the operator doesn't change V^π when is taken as input. This implies that **if we apply the Bellman operator to $V \neq V^\pi$, we have $T^\pi(V) \neq V$.** Furthermore, we can say that **applying the Bellman operator to V produces a new value function which is closer to V^π .** This is a great news, because we can **approximate the value function V^π by applying iteratively T^π** , without the need of computing the closed form solution. Formally we have

$$\lim_{k \rightarrow \infty} (T^\pi)^k V = V^\pi, \quad \forall V \quad (81)$$

This is very useful when we have a large number of states because the complexity is dropped from $\mathcal{O}(|S|^3)$ to $\mathcal{O}(\#iteration|S|^2)$, being the computation of the Bellman operator quadratic in the number of states. **The discount factor plays an important role on the convergence, because it regulates the number of iterations needed.** Smaller γ will make the convergence faster, in particular we want $\frac{1}{1-\gamma} \ll |S|$. Be aware that (78) produces an exact solution, on the other hand (81) produces an approximation. Depending on the problem V^π can be reached in a finite number of iteration, instead in other problems it can be reached only asymptotically. Actually in a problem with a finite number of states sooner or later you reach the solution. **Usually what you get is that**

³²The Bellman operator can also be defined as

$(T^\pi V^\pi)(s) = \sum_{a \in A} \pi(a|s) (R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s'))$. I found this definition very unintuitive. In the professor slides, this is the used notation

³³Fixed point it means that, if we apply the operator to the input, the result will be the same as the input

you come closer to the solution, but as you get closer to it your step towards it get smaller and smaller. The **length of the steps can be used as index of precision of the approximation**, indeed if the steps are not so small it means that V^π is not so close, on the other hand when you start making small steps it means that it cannot be too far, thus usually what is done is to **put a threshold on the size of the steps you make between two iteration**, so that **when you stop you can guarantee an approximation error**. Furthermore, this procedure works with any initialization of the value function V (usually the common choice are the zero, a random vector, or if known R^π), but of course the point chosen affect the number of iteration that must be performed. The **convergence is faster for smaller γ , indeed has we have seen it affects the horizon of the problem**. For example being $\gamma = 0 \implies V^\pi = R^\pi$ since the horizon is null, and so the solution is reached in only one application of the Bellman operator no matter where you start from. For **values of γ very small in few steps you reach a good approximation, instead for values of γ closer to one the number of iteration increases**. So the speed of contraction of the Bellman operator is affected by γ , indeed is said that the **operator T^π produces a γ contraction, so with γ close to 1 the contraction is very small** (is a contraction being $0 < \gamma < 1$).

NOTE Let's consider the set of all the possible value function of a problem with a given number of states ($R^{|S|}$). What we want to find is V^π . Applying the Bellman operator T^π to another value-state function V_0 we obtain $T^\pi V_0 = R^\pi + \gamma P^\pi V_0 = V_1$ that is another value-state function closer to V^π w.r.t. V_0 , and this is true for any vector of the space $R^{|S|}$. Being V^π a fixed point when it reaches V^π it does not move anymore.

NOTE The intuition of what does the operator T^π can be obtained by considering as starting point $V_0 = 0$. Applying the operator to 0 we obtain the reward R^π ($T^\pi(0) = R^\pi + \gamma P^\pi 0 = R^\pi$). Applying again the operator to the reward we obtain the reward we expect after one step $T^\pi(R^\pi) = R^\pi + \gamma P^\pi R^\pi$, and so on. Hence each time we apply T^π is just like extending the horizon one step further, so reaching a number of step that is close to the effective horizon of the problem, that depends on the discount factor, and that can be usually approximated by the convergence of the geometric series $\frac{1}{1-\gamma}$ (e.g. $\gamma = 0.9 \implies 10$, $\gamma = 0.99 \implies 100$). So at each iteration is performing one step of prediction in the future **unrolling always more the MDP**. Starting from $V_0 = 0$ only helps to understand what the operator is doing but even starting with other points it reaches the approximation of the solution.

NOTE This is the same idea used in dynamic programming, the usage of this operators.

NOTE - Bellman operators properties Here we recap the Bellman operators properties.

- **Monotonicity** If $V_1 \leq V_2$ component-wise (i.e. in any state).

$$\begin{aligned} T^\pi(V_1) &\leq T^\pi(V_2) \\ T^*(V_1) &\leq T^*(V_2) \end{aligned}$$

i.e. applying a Bellman operator to two value functions that are again \leq in each state, so the relationship doesn't change after the application of the Bellman expectation operator T^π or the Bellman optimality operator T^*

- **Max-Norm contraction** For two value functions V_1 and V_2

$$\begin{aligned} \|T^\pi(V_1) - T^\pi(V_2)\|_\infty &= \|R^\pi + \gamma P^\pi V_1 - R^\pi - \gamma P^\pi V_2\|_\infty \\ &= \gamma \|P^\pi(V_1 - V_2)\|_\infty \\ &\leq \gamma \|V_1 - V_2\|_\infty \end{aligned}$$

where the inequality comes from the fact that we consider all the probability on the worst element so P^π creates the bound. So

$$\begin{aligned} \|T^\pi(V_1) - T^\pi(V_2)\|_\infty &\leq \gamma \|V_1 - V_2\|_\infty \\ \|T^*(V_1) - T^*(V_2)\|_\infty &\leq \gamma \|V_1 - V_2\|_\infty \end{aligned}$$

where

$$\|V_1 - V_2\|_\infty = \max_s |V_1(s) - V_2(s)| \quad (82)$$

is used to measure the precision of the approximation found. Indeed, once you apply one step of the bellman operator the distance that you have from the target is equal to the distance multiplied by γ , so being $0 < \gamma < 1$ the distance is smaller so we are contracting towards the solution, and the speed of contraction is γ (gamma contraction).

- **Fixed point uniqueness** V^π is the unique fixed point of T^π . V^* is the unique fixed point of T^*

$$\begin{aligned} T^\pi(V^\pi) &= V^\pi \\ T^*(V^*) &= V^* \end{aligned}$$

- **True value function convergence** For any value function V and any policy π

$$\begin{aligned} \lim_{k \rightarrow \infty} (T^\pi)^k V &= V^\pi \\ \lim_{k \rightarrow \infty} (T^*)^k V &= V^* \end{aligned}$$

Being an **iterative approach**, it would be nice if we can actually know at which step we had reached a good approximation. We can define $\epsilon = \|T^*(V) - V\|_\infty$ (that is

$\|T^*(V^{(k+1)}) - T^*(V^{(k)})\|_\infty$. We can bound the distance of our approximation to the real value as

$$\|V^* - V\|_\infty \leq \frac{2\gamma\epsilon}{1-\gamma} \quad (83)$$

The **same holds for Q^π** , the only thing is that the **operator is different since the Bellman equation is the one for Q and not for V** .

Definition 9.9. *The Bellman operator T^π for Q^π is defined as $T^\pi : \mathbb{R}^{|S||x|A|} \rightarrow \mathbb{R}^{|S||x|A|}$. This operator takes as an argument a action-value function and it returns a action-value function*

$$T^\pi(Q) = R^\pi + \gamma P^\pi Q$$

or

$$(T^\pi Q^\pi)(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \sum_{a' \in A} \pi(a'|s') Q^\pi(s', a')$$

Again using the Bellman operator, the Bellman expectation equation can be compactly written as:

$$T^\pi Q^\pi = Q^\pi$$

that is a linear equation in T^π and Q^π . Furthermore, Q^π a fixed point of the Bellman operator T^π and if $0 < \gamma < 1$ then T^π is a contraction w.r.t. the maximum norm.

9.5 Optimal Value function

Until now we have discussed how to find the state-value function.

Definition 9.10. *The optimal state-value function $V^*(s)$ is the maximum value function over all policies*

$$V^*(s) = \max_{\pi} V^\pi(s)$$

similarly

Definition 9.11. *The optimal action-value function $Q^*(s, a)$ is the maximum action-value function over all policies*

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$$

Hence the optimal value function specifies the **best possible performance in the MDP**. The MDP is "solved" when we know the optimal value function V^* (or Q^*). **Since once we know the optimal utility we also know how to achieve it**.

NOTE Considering again the problem of the student study we can see that the optimal value function considering again $\gamma = 1$ is the following hence the optimal policy starting from the central left state, is to study, study and study.

Value functions define a partial ordering over the policies

$$\pi \geq \pi' \text{ if } V^\pi(s) \geq V^{\pi'}(s), \quad \forall s \in S$$

so the policy can be ordered according to their performances evaluated using the state-value function. In particular a policy is greater than another if it is better in term of state-value function in any state.

Theorem 9.1. *For any Markov Decision Process*

- There exist an optimal policy π^* that is better than or equal to all the other policies $\pi^* \geq \pi, \quad \forall \pi$
- All optimal policies achieve the optimal value function, $V^{\pi^*}(s) = V^*(s)$
- All optimal policies achieve the optimal action-value function, $Q^{\pi^*}(s, a) = Q^*(s, a)$
- There is always a deterministic optimal policy for any MDP.

In particular from the theorem we can deduce that:

- Always exist a policy that is equal or better to all the other policies in all the states, so is never sub-optimal in any state. This means that we do not need to do any trade-off, i.e. sacrifice utility in some states to gain utility in some other states, because there is a policy that obtains the maximum utility in all the states of the problem. There may be more than one optimal policy but at least one exist.
- All the optimal policy are equal in terms of utility.
- All the optimal policy are equal in terms of utility.
- For any MDP there is always an optimal policy that is in the Markovian deterministic stationary policies set. Hence if I want to solve a MDP I can restrict the attention to this smaller subset of policies.

Hence the **problem is well formed**, and can be solved optimally and solving the problem, i.e. **finding the optimal value function, is our goal**.

NOTE The theorem refers to infinite (or indefinite) horizon with discounting. Indeed as we have seen the finite horizon the optimal policy is non-stationary. Here instead we are considering stationary solutions.

IMPORTANT At the moment we are not doing machine learning, this is optimization. We are saying that knowing the problem the problem has an optimal solution and this is what are the optimal solution properties. We still not have found a way to estimate the solution from data, i.e. machine learning. We are studying the existence of solution of the problem that then will be solved using machine learning. Now we have defined the solution, and we will see how to compute the solution: these are different things to learn the solution from data (ML).

Furthermore, a deterministic optimal policy can be simply found if we have found Q^* , by maximizing over $Q^*(s, a)$

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in A}{\operatorname{argmax}} Q^*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

i.e. we are simply giving probability equal to one to the action that maximize, the Q^* in the state s . Being Q^* a table $|S| \times |A|$, if I want to know the optimal action in a specific state I take the row corresponding to that state in the Q^* table and I will choose the action associated to the maximum value, and that is the optimal action. Notice that knowing Q^* is the only requirement to find the optimal policy (instead as we will see knowing V^* requires also to know the action probability, i.e. the model)

NOTE The optimal policy for student MDP is the following and can be computed looking at Q^* as explained where in each state the optimal policy is determined by considering the highest Q value.

Instead, if I have found the state-value function is not so simple to find the optimal policy since I do not have the utility in each state for each action but the cumulative utility of each state. Hence by substituting the relation between Q and V we have:

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in A}{\operatorname{argmax}} Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s') \\ 0 & \text{otherwise} \end{cases}$$

NOTE Until now we have seen that Q^* exists and there is always a solution that is deterministic Markovian stationary that achieves that performance. How to compute the policy knowing Q^* , but we do not have seen yet how to compute Q^* .

Considering the Bellman expectation equation, used when we want to compute the value function of a given policy, is possible to define **Bellman optimality equation for V^*** , that is the Bellman equation to compute the optimal value function.

$$V^*(s) = \max_a Q^*(s, a)$$

$$= \max_a \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s') \right\}$$

comparing it with the Bellman expectation equation ($\sum_{a \in A} \pi(a|s)(R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V^*(s'))$) the only difference is that while in the Bellman expectation equation we take the expected value with respect to the policy, summing over the action with the probability of taking the action, instead in the optimality equation maximize over the action. The difference is small but makes a significant difference indeed **having the summation the equation lead to a system of linear equations, but having the maximization, we still have a system of equation that is no more linear, i.e. a system of non-linear equation with $|S|$ unknowns and $|S|$ equations**. The consequence of this is that we have **no more closed form solution**: for computing V^π we had a closed form solution but for the computation of V^* does not exist a closed form solution. The same holds for Q^* , for which the Bellman optimality equation for Q^* is:

$$\begin{aligned} Q^*(s, a) &= R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s') \\ &= R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \max_{a'} Q^*(s', a') \end{aligned}$$

where the summation over a' are replaced with the maximum over a' , so the system has non-linear equations, and there is not a closed form solution to the problem.

NOTE The difference between the Bellman expectation equations and the optimality one is that in the expected Bellman equations the actions are fixed by using the policy and summed according their weight in the policy, instead in the Bellman optimality equations we are trying to find the optimal policy that requires the maximization over the action, hence we are looking for the best action not to the expected state-value due to a certain policy π . Hence the expected value operator (linear operator) is replaced with the maximum operator (non-linear operator), implying the non-existence of a closed form solution.

Example - Optimality check in student MDP Doing the same for each state we see that the value-state function satify the optimality equation, so it is the optimal value function.

NOTE The optimal state-value function must be searched in all the states, obviously.

For what we have just said **does not exist a closed form solution** for the optimal (action-) value function, but the **properties of the iterative approach are still valid**. So we cannot use closed formulas but iterative approaches can be used because the **Bellman**

optimality operator can be applied repeatedly and converge to the solution, at a speed again regulated by the discount factor γ

Definition 9.12. *The Bellman optimality operator for V^* is defined as $T^* : R^{|S|} \rightarrow R^{|S|}$ (maps value functions to value functions)*

$$(T^*V^*)(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V^*(s') \right\}$$

hence also T^* has a **unique fixed point** V^* , and T^* is a **gamma contraction** in maximum norm for any vector.

The same holds for Q^*

Definition 9.13. *The Bellman optimality operator for Q^* is defined as $T^* : R^{|S||x||A|} \rightarrow R^{|S||x||A|}$ (maps action-value functions to action-value functions)*

$$(T^*Q^*)(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \max_{a'} Q^*(s', a')$$

Finally, the properties of the Bellman operators, already discussed, are the following:

- **Monotonicity** If $f_1 \leq f_2$ component-wise.

$$\begin{aligned} T^\pi(f_1) &\leq T^\pi(f_2) \\ T^*(f_1) &\leq T^*(f_2) \end{aligned}$$

- **Max-Norm contraction (gamma contraction)** For two value functions f_1 and f_2

$$\begin{aligned} \|T^\pi(f_1) - T^\pi(f_2)\|_\infty &\leq \gamma \|f_1 - f_2\|_\infty \\ \|T^*(f_1) - T^*(f_2)\|_\infty &\leq \gamma \|f_1 - f_2\|_\infty \end{aligned}$$

where

$$\|f_1 - f_2\|_\infty = \max_s |f_1(s) - f_2(s)| \quad (84)$$

- **Fixed point uniqueness** V^π is the unique fixed point of T^π . V^* is the unique fixed point of T^*

$$\begin{aligned} T^\pi(V^\pi) &= V^\pi \\ T^*(V^*) &= V^* \end{aligned}$$

- **True value function convergence** For any vector $f \in R^{|S|}$ and any policy π

$$\begin{aligned} \lim_{k \rightarrow \infty} (T^\pi)^k f &= V^\pi \\ \lim_{k \rightarrow \infty} (T^*)^k f &= V^* \end{aligned}$$

As said **Bellman optimality equation** are non-linear, so there is not a closed form solution, but many iterative solution methods can be applied

- Dynamic programming
 - **Value iteration** i.e. the iterative application of the Bellman optimlaity operator
 - **Policy iteration**
- Linear programming
- Reinforcement learning
 - Q-learning
 - SARSA

where **dynamic** and **linear** programming solve MDP when you know the model of the problem, i.e. the transition model, instead reinforcement learning solves the problem when you have no such knowledge and you want to solve this using direct interaction, i.e. data, and no the knowledge of the model.

9.6 Solving Markov Decision Processes: Dynamic programming

Solving MDP means finding an optimal policy. To find the optimal policy (i.e. the one with the optimal value function) the **naive approach** of policy search consist of **brute force**

- **enumerating all the deterministic Markov policies.** We can limit our attention to deterministic policies since we know that surely exist a policy in the deterministic Markovian stationary set of policies.
- **evaluate each policy**
- **return the best one**

this **simple approach** of enumerating all possible policies is **unfeasible**. In fact, we would need to evaluate $|A|^{|S|}$, so **even for very simple problems the number of policies is too large to be enumerated with the brute force**. Hence we need a more intelligent search for the best policies:

- **restrict the search to a subset of the possible policies**, parametrized in some way using function approximation (e.g. neural networks). Encoding the knowledge with some **parameters you may encode the knowledge you have about the problem** in order to make the policy to make some action in some states, since the policy shape does not allow to take that action.
- **using stochastic optimization** algorithms, **following the gradient of the performance of the policy to improve the performance**.

So to solve exactly an MDP the **unique way in policy search is to use brute force but it is not a viable solution** due to its complexity.

We can use **dynamic programming to find a better way to find π^*** .

- **Dynamic:** sequential or temporal component to the problem
- **Programming:** optimizing a "program", i.e. a policy

Dynamic Programming is a **very general solution method** for problems which have **two properties**

- **Optimal sub-structure** Optimal solution can be decomposed into sub-problems and the principle of optimality applies. A problem is said to satisfy the principle of optimality, if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems.
- **Overlapping sub-problems** sub-problems recur many times and solutions can be cached and reused, i.e. they are somehow interrelated.

Indeed based on the idea of decomposing the problem into sub-problems, easier to be solved, then solved in a recursive way. Markov decision processes **satisfy both properties**.

- Bellman equation gives recursive decomposition for the computation of the value function.
- Value function stores and reuses solutions since the value functions are related to each other through the transition model.

Dynamic programming requires the full knowledge of MDP so you need to know the state transition model exactly and the reward function. Thus dynamic programming is used for planning in an MDP.

We distinguish between two types of problems:

- **Prediction:** estimating the state-value function of a given policy
 - Input: MDP $< S, A, P, R, \gamma, \mu >$ and a policy π (i.e. MRP $< S, P^\pi, R^\pi, \gamma, \mu >$)
 - Output: value function V^π
 that as seen it **can be solved in a closed form or approximately** using a Bellman operator.
- **Control:** estimating the optimal value function, and the optimal policy
 - Input: MDP $< S, A, P, R, \gamma, \mu >$
 - Output: value function V^* and optimal policy π^*

NOTE We focus on application of dynamic programming on Markov decision processes. But they can be used for many other problems.

Even if this is not our main goal let's say something about **finite-horizon problems**, i.e. where your **agent knows how many steps are left before the ending of the problem**. For the principle of optimality, the tail of an optimal policy is optimal for the "tail" problem. Due to this fact to solve this kind of decision making problems, we can use the backward induction: starting from the last time step of the problem, for all the states in the last time step, you compute which is the optimal action. Since in the last step there is no future, the optimal action is the one that maximizes the immediate reward. This is repeated until the first step of the problem is reached. So you have propagated the function through time starting from the last step, going backward to the first step, obtaining a value function for each time step, so that the optimal policy is the one that maximize the value function in each time step. As we have said **since the value function changes with the time also the optimal policy changes with time, meaning that it is in general non-stationary**.

- Backward recursion

$$V_k^*(s) = \max_{a \in A_k} \left\{ R_k(s, a) + \sum_{s' \in S_{k+1}} P_k(s'|s, a) V_{k+1}^*(s') \right\} \quad k = N - 1, \dots, 0$$

note that γ is not present since it is an undiscounted problem.

- Optimal policy

$$\pi_k^*(s) \in \operatorname{argmax}_{a \in A_k} \left\{ R_k(s, a) + \sum_{s' \in S_{k+1}} P_k(s'|s, a) V_{k+1}^*(s') \right\} \quad k = 0, \dots, N - 1$$

The computational cost of this procedure is $N|S||A|$, that compared with $|A|^{|N|S|}$ of the brute force policy search is much lower (where N is the number of steps to be performed).

From now on we will consider infinite-horizon discounted MDPs.

9.6.1 Dynamic programming: Policy iteration

Policy iteration is an iterative method to find the optimal policy of an MDP. It's an extension of what we have seen for the evaluation of a value function. The method can be divided in two steps

1. **Policy evaluation** In this step, our objective is to evaluate the true state-value function V^π of a policy π i.e. solve the prediction problem. We have already

seen how we can estimate it. In closed form with (78), or with an iterative approach (81). Recalling the **state-value function for the policy π** is given by

$$V^\pi(s) = E \left\{ \sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right\}$$

In Bellman equation form (for V^π)

$$V^\pi(s) = \underbrace{\sum_{a \in A} \pi(a|s)}_{\text{summing over the action to consider stochasticity}} \left(R(s, a) + \gamma \sum_{\substack{s' \in S \\ \text{all the possible next states}}} P(s'|s, a) V^\pi(s') \right)$$

that is a **system of $|S|$ simultaneous linear equations**, whose solution in matrix notation is

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

that require a **complexity** of $O(n^3)$. But instead of using the closed form solution we could use an **iterative approach** by iteratively applying the **Bellman operator**, that produces a sequence of state-value functions that converges with the **limit of infinite application of the operator** ($V_0 \rightarrow V_1 \rightarrow \dots \rightarrow V_k \rightarrow V_{k+1} \rightarrow \dots \rightarrow V^\pi$). hence we obtain a full policy-evaluation backup:

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s') \right]$$

where $k+1$ is the new iteration, that is computed using values of the iteration k ($V_{k+1}(s) \leftarrow T^\pi V_k$). A sweep consists of applying a **backup operation** to each state, computing the new value function at each state. Using **synchronous backups** (i.e. updates done at the same time)

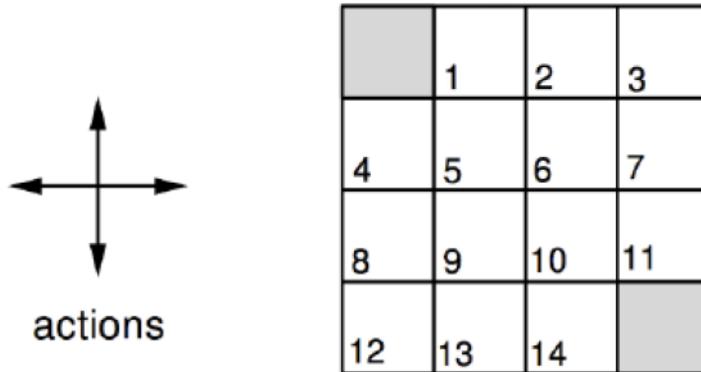
- At each iteration $k + 1$
- For all states $s \in S$
- Update $V_{k+1}(s)$ from $V_k(s')$

instead another approach is an **asynchronous** (i.e. update done as soon as possible) one where you use $V_{k+1}(s')$ instead of $V_k(s')$ for the states for which it was already updated, so you do not wait to compute all the values before changing V_k ($V_k = [V_{k+1}(s_0), V_k(s_1), \dots]$). This last approach could let the **convergence of the procedure** to speed up a little bit.

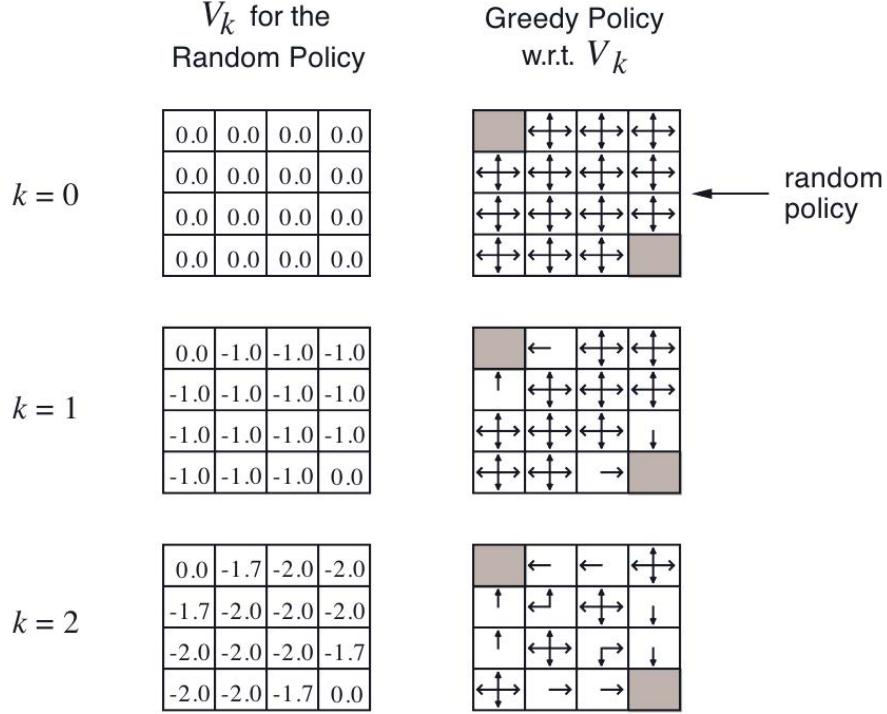
2. **Policy improvement** In this step, we improve our policy, generating a new one, based on the information given by the evaluation of the state-value function of the previous step.

The method start with a given or random policy π_0 . First, we apply the policy evaluation step generating V^{π_0} . Then we apply the policy improvement using V^{π_0} to generate a new policy π_1 . We iterate this two step iteratively ($\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \dots$). The policy improvement step is done in order to obtain a new policy which satisfies $V^{\pi_i} \leq V^{\pi_{i+1}}$. This implies that at each evaluation-improvement cycle the newly generated policy is monotonically better than the previous ones. We reach the convergence when $V^{\pi_i} = V^{\pi_{i+1}}$. This means that we have found V^* and the relative optimal policy π^* .

Example - Grid world policy evaluation Imagine to have a 4x4 grid. This grid represents our MDP states. From each cell we can make four actions. Go up, down, left or right. If the agent make an action which result in reaching a cell outside the grid, the state remains unchanged (e.g. 3, up \rightarrow 3). The action will have deterministic results. The top left and bottom right state are absorbing states (goals), while the other are transient state from which the agent can go in and out. If we reach these states the episode is concluded. The immediate reward is -1 until an absorbing state is reached (then is 0). We use $\gamma = 1$, so it is an undiscounted problem. We also suppose that we choose a policy that sooner or later reach the absorbing state.



What we want to do is to evaluate the performance of the random policy using the application of the Bellman expectation operator.



Notice that here we are doing only policy evaluation. We start ($k = 0$) from our random policy π_0 and state-value function $V_0^{\pi_0}$ initialized to all zeros. This means that in every state we have an equal probability of choosing one of the four possible actions and every action has the same utility. We apply the Bellman operator to find an approximation of V^{π_0} . To update the state-value function in a state we use

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_k(s') \right]$$

To tidy up the notation we remove the policy superscript on V_k . In practice, using synchronous backups at each iteration $k + 1$, for all state $s \in S$, we update $V_{k+1}(s)$ from $V_k(s)$. For example let's take state 5. We know that for every action $\pi^0(a|5) = \frac{1}{4}$

$$V_1(5) = \sum_{a \in A} \frac{1}{4} \left[-1 + \underbrace{\gamma \sum_{s' \in S} P(s'|s, a) V_0(s')}_{V_0(s')=0 \implies 0} \right] = -1$$

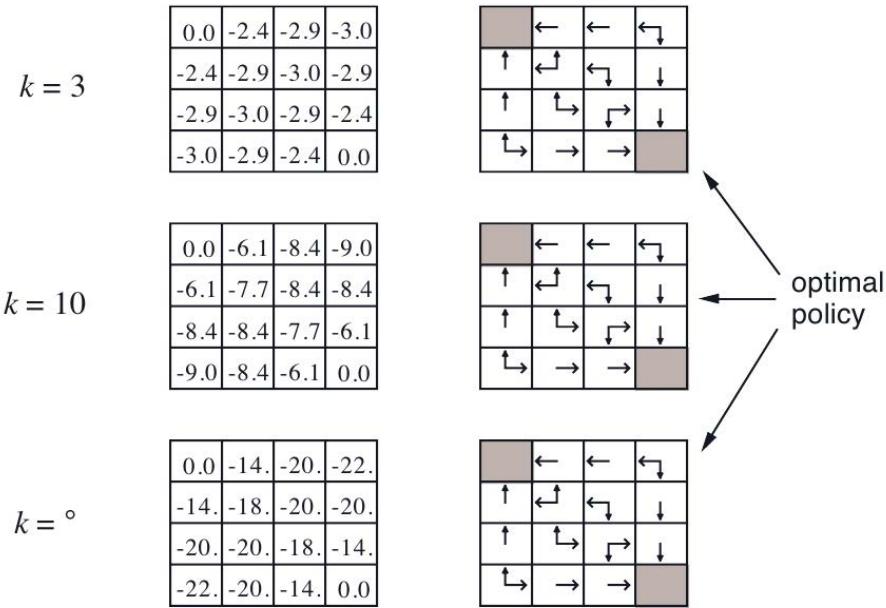
This holds true for every state, **excluded the absorbing state where the state-value function is always zero**. For $k = 2$ we have.

$$\begin{aligned} V_2(5) &= \sum_{a \in A} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_1(s') \right] \\ &= \sum_{a \in A} \frac{1}{4} \left[-1 + \sum_{s' \in S} P(s'|s, a)(-1) \right] \end{aligned}$$

$$\begin{aligned}
&= \sum_{a \in A} \frac{1}{4} \begin{bmatrix} -1 & -1 \end{bmatrix} \\
&= -2
\end{aligned}$$

Notice that if the action would let the agent outside the matrix the state does not change, so the value function to consider is the one of the same state. Applying the Bellman operator on the random policy we are simply averaging the state-value function of the neighbours and adding the immediate reward.

$$V_1(1) = -1 + \frac{1}{4}(0 - 1 - 1 - 1) = -1.75$$



If we keep going iterating the Bellman operator, we obtain what we see in the last row of the image above. In the images above, for every iteration of the policy evaluation step we calculate the greedy policy. The **greedy policy** is the policy which **choose in every state the action which leads to the most promising state w.r.t. V_k** . Note that **at the first iteration, in all the states close to the absorbing state, choosing the action that brings you to the absorbing state is better since the reward in that case is 0, instead when you go to the other states is -1 . Instead in the inner state the actions are all the same since always lead to a state whose value is -1** . Continuing the iteration, the greedy policy changes according to the new V_k definition. We can notice how after three steps the greedy policy is already optimal. This is due to the fact that from every state we need to perform at most three steps to reach the goal.

The fact that **after few steps the greedy policy is already the optimal one**, means that we have already some kind of information even after 3 iteration, **that is not the absolute value of the value function, but the shape of the value function**. Indeed

the value of the **value function** are very far from the true ones, but is the shape of the value function to define the best policy that can be estimated (imagine a function over the matrix, the magnitude of the values is not correct but the shape is defined i.e. the distribution of the values is correct). Since **what is wrong is the magnitude of the values but their "quantitative order"** is defined the greedy policy won't change, and the optimal policy can be found without waiting the true value function.

Consider a deterministic policy π . For a given state s would it better to do an action $a \neq \pi(s)$? Once we have found V^{π_0} we can calculate a new policy π_1 that improves the value function. We can **improve the previous policy by acting greedily in every state**, i.e. using the greedy policy in each state. The **greedy policy is a deterministic policy that chooses the action that are the best according to the value function that you have computed**. So the improved policy is the greedy one that, having computed the action-value function is

$$\pi_1(s) = \underset{a \in A}{\operatorname{argmax}} \left\{ Q^{\pi_0}(s, a) \right\}$$

Hence the **greedy policy π' is the one that maximizes the utility looking at the value function of another policy**. Beware that π' is not said to be the optimal policy, because we know that the optimal policy is the greedy policy with respect to the optimal value function

$$\pi^* = \underset{a \in A}{\operatorname{argmax}} \left\{ Q^*(s, a) \right\}$$

So if I have the optimal value function applying the greedy operation gives the optimal policy, but if I have a certain policy π the greedy operation give us another policy π' that is not said to be optimal, **but we know that π' is never worse than π in any state**

$$\pi' \geq \pi$$

This improves the value from any state s over one step

$$Q^{\pi_0}(s, \pi^1(s)) = \underset{a \in A}{\operatorname{max}} \left\{ Q^{\pi_0}(s, a) \right\} \geq Q^{\pi_0}(s, \pi^0(s)) = V^{\pi_0}$$

i.e. is like it is unrolling the recursion one step further.

Furthermore:

Theorem 9.2 (Policy improvement theorem). Let π and π' be any **pair of deterministic policies** such that

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s), \quad \forall s \in S$$

Then the policy π' must be as good as, or better than π

$$V^{\pi'}(s) \geq V^\pi(s), \quad \forall s \in S$$

Proof.

$$\begin{aligned}
V^\pi(s) &\leq Q^\pi(s, \pi'(s)) = E_{\pi'}[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s] \\
&\leq E_{\pi'}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi'(s_{t+1}))|s_t = s] \\
&\leq E_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 Q^\pi(s_{t+2}, \pi'(s_{t+2}))|s_t = s] \\
&\leq E_{\pi'}[r_{t+1} + \gamma r_{t+2} + \dots |s_t = s] = V^{\pi'}(s)
\end{aligned}$$

□

Since **can be** $\pi' = \pi$, what happens if improvements stops ($V^{\pi'} = V^\pi$)? **If it happens it means that**

$$Q^\pi(s, \pi'(s)) = \max_{a \in A} \left\{ Q^\pi(s, a) \right\} = Q^\pi(s, \pi(s)) = V^\pi(s)$$

but this is the Bellman optimality equation. Therefore

$$V^\pi = V^{\pi'}(s) = V^*(s) \quad \forall s \in S$$

but this means that π **cannot get better since it is the optimal policy**:

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_0} \rightarrow \dots \rightarrow \pi^* \rightarrow V^* \rightarrow \pi^*$$

At this point we have an algorithm, the following iterative process:

1. Starting from the policy π , estimate its value function.
2. Apply the greedy operation, getting a new policy π' that is never worse (i.e. better or equal) w.r.t. policy π in any state.
3. There are two cases:
 - the greedy policy is an improvement \Rightarrow iterate
 - the greedy policy is not an improvement \Rightarrow we reached the optimal policy, stop.

Since at each step you need to improve, at a certain point **in the worst case you will finish all the possible policies**, so it will terminate in a finite number of steps finding the optimal value function and the optimal policy.

NOTE Supposing that, in the greedy step ($\arg\max_{a \in A} Q^\pi(s, a)$) to find π' , instead of using Q^π we use the definition ($\arg\max_{a \in A} R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s')$), **in some sense we are applying the Bellman optimality operator since in some sense we are maximizing over the action**, and this has the same effect of the Bellman optimality operator. So the only time where you cannot improve is if that is in the fixed point of the Bellman optimality operator, otherwise there must be an improvement.

NOTE Why policy iteration is better than applying recursively Bellman optimality operator to find V^* ? The latter is the **alternative** to policy iteration and is called **Value iteration**. Among the two there is no one that is better than the other one, i.e. there is not a better alternative. They **have different properties**:

- Each single iteration of Policy iteration is more expensive than a single iteration of value iteration. Since the value iteration is simply the iteration of the Bellman optimality operator, while an iteration of the policy iteration consist in a policy evaluation (using the Bellman expectation operator NOT the optimality one as in the value iteration) step plus the greedy operator, so it is more expensive.
- Each iteration of policy iteration produces larger improvements with respect to value iteration.
- While value iteration may achieve the optimal value function only in the limit of infinite application of the operator, policy iteration converges to the optimal solution in a finite time. This considering a result without approximation, i.e. zero error computation is achieved for one in finite time and the other has only an asymptotic convergence.

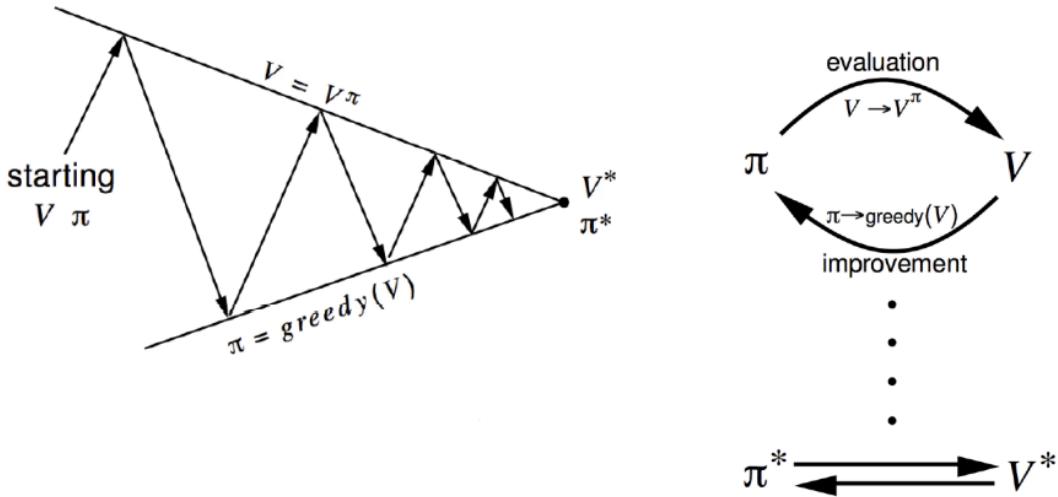
Policy iteration can't be stuck in local minima. We have a **finite number of policies**. For the theorem we have seen above, if we find a greedy policy π' with respect to V^π , we are sure that $V^\pi \leq V^{\pi'}$. This means that if we found, in two consecutive steps two identical value function, we have reached the optimal V^* , and so π^* .

Modified/Generalized policy iteration Does the policy evaluation need to converge to V^π ? Or should we introduce a stopping condition (e.g. ϵ -convergence of value function)? Or simply stop after k iterations of iterative policy evaluation? In the example above, we can notice that in the policy evaluation step, after the third iteration of the Bellman operator, the corresponding greedy policies of each iteration are the same. Stopping our policy evaluation at the third iteration would have produced the same policy improvement step. It can be showed that using this kind of procedure (**Modified policy iteration**) that consist only in few iteration of the iterative policy evaluation, computing an approximation of the value function, the convergence is still reached. of course is not guaranteed to be more efficient, i.e. to converge faster, but you are sure you are reducing the cost of each iteration of the policy iteration and in the limit will converge to the optimal solution anyway. For example is possible to do a policy update (improvement) after only one step ($k=1$) of Bellman expectation operator (evaluation) without waiting to reach perfect convergence of the policy evaluation. Of course the finite convergence (i.e. convergence in a finite number of step) is guaranteed when you compute the exact value function during the policy evaluation, instead if we stop earlier is guaranteed an asymptotic convergence. Hence **Generalized policy iteration** simplify the policy evaluation stopping the estimation step earlier, in order to reduce the amount of computation needed.

NOTE Policy evaluation converges to V^π since we are using the Bellman expectation operator, i.e. it evaluates the optimal function of a given policy, not the optimal one (policy). Instead in the value iteration procedure, where we use the Bellman optimality operator it converges to V^* .

A recap of what we said:

- Policy evaluation: Estimate V^π (e.g. iterative policy evaluation)
- Policy improvement: generate $\pi' \geq \pi$ (e.g. greedy policy improvement)



where the two converging lines in the figure on the left, represent one the space of policies and the other the space of value functions. You achieve the condition of **finding the optimal policy and value function** after a **finite number of iteration**. The algorithm **stops if there is no improvement**, i.e. both the last policy found and the greedy one belongs to the set of optimal policies (indeed they are different policies whose value function has the same utility).

9.6.2 Value iteration

Dynamic programming gave us **another methodology** to find the optimal policy of an MDP (i.e. to solve it). As we have hinted before, we can actually use the **Bellman optimality operator** to **iteratively estimate V^*** . We know that finding V^* is equivalent to solving the MDP. So we know that starting from an arbitrary value function V , we can obtain V^* by applying **iteratively infinite time** the Bellman optimality operator ($\lim_{k \rightarrow \infty} (T^*)^k V = V^*$). Using a synchronous backups at each iteration $k + 1$, for all the states $s \in S$, update $V_{k+1}(s)$ from V_s . **For practical reasons, we can't perform infinite**

iteration steps. We stop iterating when we see that our approximation is close to V^* . We can use as a **stopping condition** the bound ($\|V^* - V\|_\infty \leq \frac{2\gamma\epsilon}{1-\gamma} \rightarrow (83)$). Hence, by moving in the space of all the value functions, starting by an arbitrary value function, using the Bellman optimality operator, **being the operator a max-norm contraction w.r.t. to the fixed point V^*** it moves towards V^* .

$$V_0 \rightarrow V^1 \rightarrow \dots \rightarrow V^*$$

Once we have obtained a $V_k \approx V^*$, we can find the relative greedy policy to estimate the true optimal policy

$$\tilde{\pi}^*(s) = \underset{a \in A}{\operatorname{argmax}} \left\{ R(s, a) + \gamma \sum_{s' \in s} P(s'|s, a) V_k(s') \right\} \quad (85)$$

Must be noticed that unlike policy iteration **there is no explicit policy**: while in the policy iteration you start from a policy that you have to store in memory and then compute the value function, here in memory you have to keep only the value function that you update using the value function operator. Furthermore, intermediate value function may not correspond to any policy, indeed while in the policy iteration the value function where found using a policy (i.e. Bellman expectation operator) in a value iteration is not sure that exist a policy that is able to achieve that value function. You are **sure that exist a policy corresponding to the value function only when the optimal value function is reached**.

NOTE Remember that there is no closed form solution of the Bellman Optimality operator, since it produces a set of **non-linear equation** because in each Bellman optimality equation there is the **maximization over the action space** that introduce a **non linearity equation**.

The value iteration algorithm is based on the property of the Bellman optimality operator, so defining the maximum-norm as:

$$\|V\|_\infty = \max_s |V(s)|$$

Differently from the policy iteration where we can prove the convergence in a finite number of steps the value iteration converges asymptotically to the optimal value function and converges thanks to the max-norm contraction where the factor of contraction is the discount factor γ

Theorem 9.3. *Value Iteration converges to the optimal state-value function*

$$\lim_{k \rightarrow \infty} V_k = V^*$$

Proof.

$$\|V_{k+1} - V^*\|_\infty = \|T^*V_k - T^*V^*\|_\infty$$

$$\begin{aligned}
&\leq \gamma \|V_k - V^*\|_\infty \\
&\leq \dots \\
&\leq \gamma^{k+1} \|V_0 - V^*\|_\infty \rightarrow \infty
\end{aligned}$$

□

Furthermore

Theorem 9.4.

$$\|V_{i+1} - V_i\|_\infty < \epsilon \implies \|V_{i+1} - V^*\|_\infty < \frac{2\epsilon\gamma}{1-\gamma}$$

so if the distance between two consecutive value function in two consecutive iteration of the value algorithm is smaller than ϵ then we can bound the distance between the value function V_{i+1} and the optimal value function V^* . The bounding term depends on ϵ and the discount factor. Hence, as we have already noticed, when we are far from the optimal value function, the Bellman optimality operator makes you perform large steps in the value function, instead when you get close to V^* what happens is that you start to make smaller and smaller steps. This behaviour is encoded by the above theorem: if ϵ is large you can be far from V^* but if it is small you cannot be too far from it. Furthermore, for how the bound is defined if γ is close to 1 you need a lot of iteration before converging (as we should expect), indeed the bound is quite loose, and the value function is very close to the optimal one only if ϵ is very small. The latter consideration connects to the fact that when the discount factor is large the problem is harder than when the discount factor is small. Hence for these reasons, the latter theorem can be used for the value iteration algorithm as a **stopping condition**, since the accuracy you have achieved with respect to V^* is bounded by $\frac{2\epsilon\gamma}{1-\gamma}$, that is equivalent to set a distance that is not larger than that. In practice

$$\begin{aligned}
\tau &= \frac{2\epsilon\gamma}{1-\gamma} \\
\implies \epsilon &= \frac{1-\gamma}{2\gamma}\tau
\end{aligned}$$

and we test that $\|V_{i+1} - V_i\|_\infty < \epsilon$, i.e. V_{i+1} is close enough to the optimal one.

NOTE Looking at the Stanford demo of gridworld dynamic programming we could see how the **policy iteration** gradually propagates the best policy like a oil stain, starting from the absorbing states. Instead the **value iteration** adjust gradually from the first step the value functions and the correlated greedy policies, improving at each step.

NOTE An **hybrid method** between policy iteration and value iteration are sometimes used. For example, we could **apply value iteration**, remembering that **intermediate value function may not have a corresponding policy**, then I take the **value function** and **compute the greedy policy with respect to that value function** and then I start to do **policy iteration**. The advantage of this hybrid approach would be to get closer, somehow to the optimal value function, and then use the **policy iteration to have not just an asymptotic convergence but a finite convergence**.

Remember This two algorithms are not RL algorithm, these are dynamic programming algorithm, since they needs to know the dynamics of the problem, the model of the transition, not required instead when we use reinforcement learning.

Efficiency of DP Synchronous dynamic programming algorithms

- if based on the state-value function $V^\pi(s)$ or $V^*(s)$ have a complexity of $O(mn^2)$ per iteration, with m the number of actions and n the number of states.
- if based on the action-value function $Q^\pi(s, a)$ or $Q^*(s, a)$ have a complexity of $O(m^2n^2)$ per iteration.

Furthermore, to find the optimal policy the complexity is polynomial in the number of states, but the number of states is often very very big, e.g. (often) growing exponentially with the number of state variables (curse of dimensionality). In practice classical DP can be applied to problems with few millions states. Asynchronous DP can be applied to larger problems, and appropriate for parallel computation. It is surprisingly easy to come up with MDPs for which methods are not practical.

9.7 Linear Programming

Beyond dynamic programming we can also use linear programming, that find the solution of a MDP, **using a linear program**.

To do that, again, the **idea** is to use the **Bellman optimality equation**. It states that for each state in the problem the optimal value function in that state is equal to:

$$\forall s \in S \quad V^*(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s') \right\}$$

i.e. the **recursive decomposition** of the Bellman optimality equation (**immediate reward + the what we expect to gain as recursive state-value**). Starting from this point we can rewrite it as the following **linear program (optimization) formulation to find V^*** :

$$\min_V \quad \sum_{s \in S} \mu(s) V(s)$$

$$\text{s.t. } V(s) \geq R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V(s') \quad \forall s \in S, \forall a \in A$$

Hence we rephrased the problem defining a **minimization problem**, where the **unknowns are the values in each state (V)**, and μ is the probability of the initial state (that is known). So we have to find the values of V that minimizes that summation, that is the **expected value of the value function V with respect to the probability of the starting state μ** . The minimization is then subject to the **constraint** that encodes the **Bellman optimality equation**, indeed in each state the value of the state is never small than the same quantity contained in the maximum operator of the Bellman optimality equation. The **constraint must be valid for each combination of state and action**. Hence we are **replacing the maximum over the action with a number of constraints that is equal to $|S||A|$** (for each state, action combination) in $|S|$ the unknown (the state values).

Theorem 9.5. *V^* is the solution of the above linear program.*

Indeed, supposing you **start from the zero** value function, what you will find (assuming that the optimal value function is larger than zero in all the states) is that **none of the constraints will be satisfied**. So this will **push the value function up**. Instead, supposing you **start from a value function that is very large in all states**, since in the linear program we have the **minimization** of the weighted value function, it **will push the value function down and the constraints will prevent it to go below the true value function**. Hence there is a force that pushes the value function up (i.e. the constraints) and another that pushes the value function down (i.e. the minimization), in this way **the only solution that satisfies the linear program is the value function V^*** .

What is the **benefit** of writing the **solution of a MDP as the solution of a LP**? There are a lot of commercial or open-source **linear solver** that are **very optimized**, that allow to **solve LP in very efficient way**. Furthermore, **the worst case complexity of a linear program is much lower of the worst case complexity of dynamic programming**, so it seems that a linear program should be better than using LP. This is true theoretically, but **in practice, solving some problem you will see that dynamic programming algorithm converge much faster than the time required by a linear program for solving the MDP**. This is due to the fact that **dynamic programming has been built to solve MDP**, so it **exploits the structure of MDP a lot**, they are customized to solve MDPs, while **linear program solver is a general tool that solves a very large class of problems so is not able to properly exploit the structure of MDP**, while **solving it**. This gives to the dynamic programming method a great advantage w.r.t. LP when solving MDP. For this reason you can solve optimally a MDP with LP, but this is not the common way to do it, but instead using dynamic programming. Also 99% of RL algorithms take inspiration from dynamic programming and not from linear programming. There are for example approximated linear programming that use linear programming approach also in the RL setting, but usually RL is an adaptation of dynamic programming in a setting where the model is unknown.

NOTE In dynamic programming the probabilities of starting from a state μ are not relevant since in dynamic programming what you are going to do is to learn the value function everywhere. Instead in the linear programming case is used since you have to have a scalar objective function, so a function that put together in a single function the performance of the agent, and those can be encoded as the expected discounted reward encoded in $V(s)$ weighted by the probability of starting from that state, so this encodes the expected value of the agent gain.

NOTE Until now we have seen how to solve a MDP, when you have a perfect knowledge of the MDP model.

NOTE Dynamic programming is not always better than linear programming, since the worst case complexity is higher than the one of linear programming. But in general dynamic programming is usually faster in practice, since it uses explicitly the bellman operator is more aware of the structure of the problem, exploiting it, while the linear programming is a little bit agnostic, indeed even if in the constraints there is encoded the Bellman equation is not getting advantage of the structure of the MDP problem. This helps a lot the dynamic programming to converge faster in the average case. We saw how in the policy iteration example, even in few iteration of iterative policy evaluation, even if the approximation of the value function is bad we have already a good shape of the value function that allow us to find the greedy policy that we find once the evaluation reach convergence. This because Bellman expectation operator encodes very well the structure of the MDP problem, allowing much faster convergence.

NOTE The consideration about the complexity on the slider are interesting, but not so important to have a deep understanding.

Furthermore, LP methods become unpractical at a much smaller number of states than DP methods do.

10 RL in finite domains

So far we have seen the very basics of MDPs and some dynamic programming method to find V^* and π^* . Now, we will focus on **RL methodologies on finite MDP** which aim to estimate as well V^* and π^* , but in **unknown environment where the MDP structure is not known**, so we are only able to measure the reward function in the states that we have explored. **RL** theory provides us a vast catalogue of algorithm that can be identified by the following **characteristics** (taxonomy of RL techniques)

- **Model-free vs Model-based** Both, being RL approaches, **do not have the knowledge of the model a-priori**. Model-free approaches try to directly estimate (i.e. learn) V^* , without (explicitly) computing the MDP model i.e. never using it. On the other hand, **model-based approach** try to solve the MDP problem (i.e. finding the optimal value function or the optimal policy) by **learning the model and then use the model to compute the optimal solution**, so they estimate the MDP model from data, and then they apply classic dynamic programming (or linear programming) methods to estimate the state-value function.
- **On-policy vs Off-policy** On-policy means that we estimate the same metric, like state-value function, of the policy that we are using to collect the data, i.e. in the interaction with the environment. Off-policy methodologies **try to estimate a policy that is different from the one generating the data**.
- **Online vs Offline** An **online algorithm update the policy or value functions every time we get new data**, interleaving data collection (interaction phase) and learning phase. Offline algorithms **separate the data collection phase from the learning phase**.
- **Tabular vs Function approximation** Tabular approaches **stores directly the values of the value functions**, i.e. stores a parameter for each state or state and action or groups of state and action, in a tabular representation. This is **feasible only on small problems**, i.e. for problems with a limited number of state and action. Function approximation doesn't store directly the values of the value function, but it approximate it with some function (e.g. linear approximator, neural network ...), **i.e. a parametric value function**. Indeed by using a parametric value function we can generalize over states and action that we have not experienced. Doing so, it can even describe infinite value function.
- **Value-based vs Policy-based vs Actor-Critic** Value based approaches try to estimate the value function directly (sort of value iteration in RL). Policy-based

search in policy space to find the optimal policy, they **do not learn the value function**. Actor-critic combines these two approaches.

In the following sections we will **focus on model-free algorithms**. We will see both prediction and control case.

Important The difference of RL and dynamic programming is that in the former the model is unknown a-priory and then learns, while in dynamic programming the model is known.

NOTE - Model-free vs Model-based Historically model free approaches are more used, but in the last years model based approaches are more used. The **main problem of model free approaches is that they need a lot of data for learning, since they do not do any assumption about the model**. Instead, in **model based approaches** you are considering a **structure of the model** you want to learn and maybe you can consider an **hypothesis space of model that comprehend only a subclass of models**, introducing a bias but reducing the variance, thus helping with the amount of data required to learn the model.

NOTE - On-policy vs Off-policy Historically on-policies approaches were more used, then nowadays **off-policy is getting more importance since in many cases I am not able to decide completely the policy of interaction of the environment**. Working off-policy **is harder** than working on-policy, but it can give advantages in many applications.

NOTE - online vs offline Of course the **offline approach is necessary in some cases when you cannot have a direct control on the system, since someone else is controlling the system**, and you can suggest a new control policy by observing the behaviour of the controller that is operating on the environment.

NOTE - Tabular vs Function Approximation The tabular methods are not so good in generalizing. Instead the **function approximation method can generalize better, but are more difficult to learn**. There is indeed the **usual problem of bias and variance trade-off**. But RL problems are much harder than supervised learning problem (RL can be seen as a superclass of supervised learning problems), so the problems are even worse: while in the supervised learning you measure the input and the target and you want to find the function that relates the input and the target, instead in the **RL** case you **measure the input (state and action) but the target is the value of the state (or state,action)**, but unfortunately the value is unknown to you. Hence the **target variable needs to be estimated and this makes the problem much harder**, indeed in the supervised learning you know the target, instead in RL you have an **estimate of the target variable that changes based on the amount of data you collect**. The estimation of the target in RL introduces a **loop in the learning problem**

that introduces serious problems, so it is not the function approximation of supervised learning, but a much more complex problem.

NOTE - Value Based vs Policy based vs Actor-Critic Value based RL is similar to value iteration in dynamic programming, so it **learns in the value space**. Instead policy based RL search the optimal policy **in the policy space and they do not try to learn the value function**, so usually what is done is to define a parametric policy space, trying to estimate the performance of the policy by **simulating the policy and then you try** (e.g. with policy gradient methods) **to search how to modify the policy in order to improve its performance**. In the actor critic the actor is the policy, so what actually takes the action in the environment. The critic instead is a value based approach that evaluate the performance of the policy by observing the policy, and suggest to the actor how to change the policy in order to improve. There are advantages and disadvantages for all the three methods.

- Value based methods are really efficient if the problem is really Markovian, since they strictly rely on the Bellman equation, so they strictly exploit the Markov assumption. The problems for this kind of methods are that if the problem is not fully observable or not Markovian they have difficulties. This methods also have problem in complex problems, where you have a very large number of action, since they do not scale well with many actions.
- Policy based approaches are able to work quite well even if the problem is not Markovian, and not fully observable and when you have continuous action, so in very complex problems. The drawback of this approach they are less efficient, so in general they require more samples than value based ones to learn good policy.
- Actor-Critic approach take the best of the two approaches: try to get the efficiency of the value based approach learning but also the capability of policy based algorithm of working on continuous actions, even in problem with non-fully observable states.

As done for the phase of dynamic programming, we can distinguish two main problems:

- Model prediction
- Model control

in particular now we will focus on model free approaches:

- Model-free prediction: estimate the value function of an unknown MRP (Markov reward process, i.e. MDP + fixed policy), simply looking at the interaction of the agent and the environment.
- Model-free control: optimize the value function of an unknown MDP.

indeed in the model based approach we would try to learn the model and this is manly a supervised learning problem, that then is coupled with dynamic programming once the estimation of the model (not the true one) is found, so that the solution can be computed if dynamic programming does scale for the complexity of the model.

10.1 Model-free prediction

With Model-free prediction we want to estimate the value function of an unknown MRP³⁴. This is clearly different from what we have seen in the previous chapter, because we **don't know the model of the MDP**. We only **know which actions are available and which states makes up the MDP**.

10.1.1 Monte-Carlo reinforcement learning (TD(1))

MC is a **model-free approach** which **don't need the knowledge of the MDP transitions/rewards**. This method has no knowledge of the dynamic of the problem but it simply **learn directly from episodes of experience** (i.e. interaction of the agent and the environment). One of the **drawbacks** of this approach is that we **must learn from complete episodes**, i.e. cannot learn from a partial episode, so in order to update its estimates it **needs to wait the reaching of an absorbing state**, i.e. the episode is ended. This can be a **problem** where there are **no observing states**, i.e. continuing problem, and also in problems when the concept of episode exist this approach is **not efficient since you have to wait the end of the episode before having information for updating the estimate**. If the **episodes are very long** composed by many actions of course the **learning can be slow**. As said the Monte-Carlo approach in order to **estimate the performance of a policy** uses the **simplest idea** possible **replacing the expected value contained in the value function with a sample mean** (i.e. the infinite definition of the expected value is replaced with the discrete empirical definition). Hence it must be used in episodic MDPs³⁵ because the **learning phase is based on the entire trajectory**³⁶, from the starting state to the goal state. MC **can't be used in MDPs that don't terminate, all episodes must terminate**.

Remember The value function is the expected value of the discounted sum of the future rewards.

NOTE So Monte-Carlo technique **cannot be used in infinite time horizons, but only in finite and indefinite**, so that sooner or later the episode will end in an absorbing state.

MC **can be used for both prediction and control**. For **prediction** we have

³⁴MDP + policy

³⁵An episodic MDP means that we have an absorbing/goal state and when we reach it, the problem starts all over again from the starting state

³⁶A trajectory is a **sequential combination of states, actions, and rewards** $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

- **Input** Episodes $\{s_1, a_1, r_1, \dots, s_T\}$, generated by following policy π in given MDP, i.e. the state s_i , the action a_i and the reward r_i that have been taken in all the time steps until the end of the episode is reached.
- **Output Value function V^π , of the policy that have generated all the experienced episodes.**

instead for **control** we have:

- **Input** Episodes $\{s_1, a_1, r_1, \dots, s_T\}$, generated by following policy π in given MDP, i.e. the state s_i , the action a_i and the reward r_i that have been taken in all the time steps until the end of the episode is reached.
- **Output optimal value function V^* and/or the optimal policy π^***

First, let's recall what the Monte-Carlo approach consist of. **MC is a class of methods that rely on repeated random sampling to obtain numerical results.** Let X be a random variable with mean $\mu = E[X]$ and variance $\sigma^2 = Var[X]$. Let $x_i \sim X$, $i = 1, \dots, n$ be n i.i.d. realization of X .

The empirical mean of X is

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^N x_i$$

We also know that the **empirical mean is an unbiased estimator** since

$$E[\hat{\mu}_n] = \mu, \quad Var[\hat{\mu}_n] = \frac{Var[X]}{n}$$

i.e. the expected value of the empirical mean is the expected value of the random variable, and the variance of the empirical mean reduces as the number of samples n increases. These properties give us the property of convergence of the empirical mean to the expected value of the problem:

- Weak law of large numbers: $\hat{\mu}_n \rightarrow \mu$
- Strong law of large numbers $\hat{\mu}_n \rightarrow \mu$
- Central limit theorem $\sqrt{n}(\hat{\mu}_n - \mu) \rightarrow N(0, var[X])$

NOTE Since the value function at each state is an expected value, MC approaches simply takes the samples and average them to find the value function.

Now we have to figure out **how to build the samples we have to average**. As we have said, our goal is to compute V^π , that is under the policy π

$$s_1, a_1, r_2, \dots, s_T \sim \pi$$

Recalling that the **return** v_t is the total discounted reward,

$$v_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_{t+T} \quad (86)$$

that is using the rewards the agent received during a single trajectory observed until the end of the episode. Being the value function the expected return, based on the return v_t we can write

$$V^\pi(s) = E[v_t | s_t = s] \quad (87)$$

that is the **expected value of the return over the all the trajectory**, i.e. the expected return. Hence in the MC technique we take for each example, **so for each trajectory of our agent observed, we simply take the return**, i.e. the discounted sum of the reward our agent has taken in each state until the end of the episode. This will be one sample of the return associated to a specific state at time zero t . The expected value is then replaced with the **empirical definition since the sample trajectories are not infinite**:

$$V^\pi(s) = \frac{1}{N} \sum_i v_i$$

Monte-Carlo policy evaluation uses empirical mean return instead of expected return to estimate V^π . In practice, we **perform various episodes and we collect the returns for every state**. Then, we use the empirical mean of the returns collected in s to estimate $V^\pi(s)$. The averaging of the returns can be performed in two ways

- **First visit** Average returns only for the first time s is visited. Hence **seeing the same state multiple times inside the trajectory you will use only one samples and is the one associated with the first visit.** (unbiased estimator)
- **Every visit** Average returns for every time s is visited. Hence **you generate one sample for each state in the episode even if the state has been visited multiple times.** (biased but consistent estimator)

but which are the **pros and cons of this two type of estimation?**

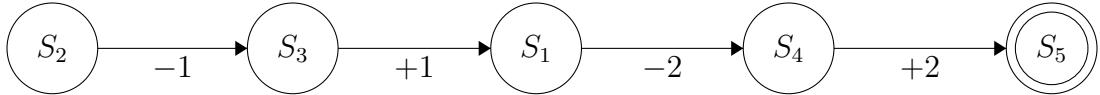
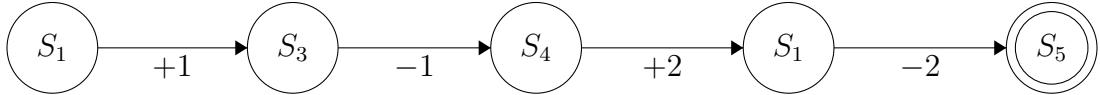
- **First visit:** it is an **unbiased estimator**
- **Every visit:** is a **biased estimator, indeed considering all the sample**, repeated states are not independent because the reward we considered in a longer trajectory are present again in the shorter ones, hence the two samples of V^π are correlated (e.g. $s_1^{(1)} \rightarrow s_2 \rightarrow s_1^{(2)} \rightarrow s_4$, the r_4 is both in the $V(s_1^{(1)})$ and $V(s_1^{(2)})$). Hence **considering dependent samples you are introducing a bias in the estimator**. Indeed the fact that the **sample average is unbiased is true only for i.i.d. samples**. But **it is still a consistent estimator, which means that the bias vanishes as the number of examples goes to infinite.**

Considering this properties we could **choose the best one depending on the problem**, in particular on the number of data you have.

- Having a lot of samples (episodes) is better to not have a biased estimator, hence is better to use first visit.
- Having few samples is better to have a biased estimator but with less variance, hence is better to use the every visit. Since the every visit uses more examples even if they are correlated actually the variance reduces, so this reduction in variance with few samples is more advantageous even if you are introducing some bias in this estimation.

NOTE Since the beginning we are highlighting one central point in machine learning and statistics: **having an unbiased estimator is not always the best thing**, is desirable having infinite samples but having few samples usually the absence of bias means having a lot of variance. Since the error is the sum of variance and bias we must find a trade-off.

Example - MC prediction Suppose to have the following episodes with $\gamma = 1$



We want to estimate the value-state function of s_1 using first visit

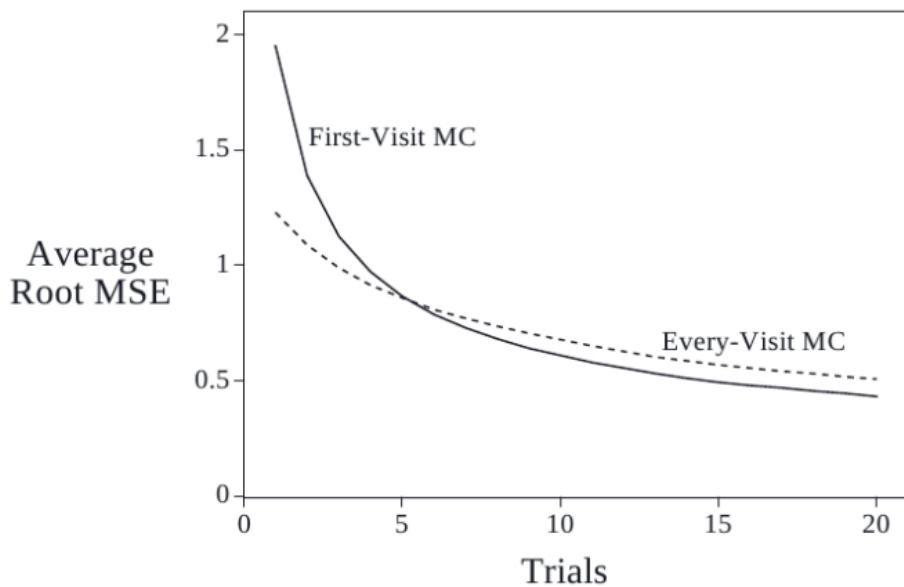
$$\begin{aligned}\hat{V}_1^\pi(s_1) &= r_1 + \gamma r_3 + \gamma^2 r_4 + \gamma^3 r_1 = 1 - 1 + 2 - 2 = 0 \\ \hat{V}_2^\pi(s_1) &= r_1 + \gamma^2 r_4 = -2 + 2 = 0 \\ \hat{V}^\pi(s_1) &= \frac{0 + 0}{2} = 0\end{aligned}$$

Using every visit, we consider all the occurrences of s_1

$$\begin{aligned}\hat{V}_{11}^\pi(s_1) &= 1 - 1 + 2 - 2 = 0 \\ \hat{V}_{12}^\pi(s_1) &= -2 \\ \hat{V}_{21}^\pi(s_1) &= -2 + 2 = 0 \\ \hat{V}^\pi(s_1) &= \frac{(0) + (-2) + (0)}{3} = -\frac{2}{3}\end{aligned}$$

This two methods have different properties. First visit is unbiased. We know for the bias-variance trade-off that, if we have low bias, we have high variance. So first visit is suited for problems with many many training samples, that can reduce the variance. First visit will produce noisy estimation of the state-value function even with a decent amount of samples (due to variance). This limits a lot the learning speed, because we need to calculate a lot of episodes. Every visit is consistent³⁷. It has more bias and less variance compared to first visit. This can be explained by the fact that every visit considers some transitions of an episode multiple times, introducing bias. But, from the same number of episodes, it can extract more training samples reducing the variance.

Looking to the **algorithm** definition the **only difference** is that for **first visit you return only one occurrence of the state s instead in the every visit you return all the occurrence of the state s** , hence the averaging for the value function is made across more samples that are correlated, and from there comes the unbiasedness of the estimator. Comparing first visit and every visit we can discover the **typical behaviour of this two methods**:



where is plotted the estimation error, so how much the approximation of the value function is good or bad, w.r.t. the number of trials, i.e. the number of episode collected. Hence every visit with a smaller number of episodes guarantee a smaller error, indeed **having few samples the main part of the error is given by the variance and not by the bias**; as you **collect more and more sample the preference of one method over the other**

³⁷An estimator is consistent if $\lim_{n \rightarrow \infty} \hat{x}_n = E[x]$. If we have infinite samples the estimator converges to the true value

change since the variance is reduced by the number of sample, hence the advantage of the unbiased estimator appears. Considering that every visit is consistent, so the bias vanishes with the number of episodes observed, in general, is still better to use the first visit approach with a large number of samples since the rate at which the bias vanishes is slower with respect to the speed at which the variance vanishes decreases in first visit. So with infinite samples they would be the same but with a finite number of samples it depends.

Example - First vs Every visit Let's analyze the following model the probability p is the combination of the dynamics of the problem and the policy of the agent. Indeed once the policy is fixed the MDP becomes a Markov chain, there are no decision, but only the probability of remaining in the state or leaving it. Since in the RL setting the model is unknown we do not know p , but we can observe the realization of this Markov chain

$$S \rightarrow S \rightarrow S \rightarrow S \rightarrow T$$

applying first and every visit we find:

$$\begin{aligned} V^{FV}(S) &= 1 + 1 + 1 + 1 = 1 \\ V^{EV}(S) &= (4 + 3 + 2 + 1)/4 = 2.5 \end{aligned}$$

Now, supposing we want to use a model based approach, trying to estimate the maximum likelihood method model, i.e. what is the value of p for the maximum likelihood model. Looking at the only episode we have a clear estimation of these probabilities since 3 times out of 4 it remained in the state S and 1 out of 4 goes to state T . So the maximum likelihood model we can build over the above episode is: that is the model that maximize the likelihood of the observed episodes events. Assuming this is the true model (we do not know since this is computed considering only one episode), i.e. fixing the policy i.e. the probability p are fixed, the value of S can be found using the bellman expectation equation

$$\begin{aligned} V(S) &= 1 + 0.75V(S) + 0.25V(T) \\ V(T) = 0 \implies V(S) &= \frac{1}{1 + 0.75} = 4 \end{aligned}$$

being $V(T) = 0$ since it is an absorbing state. Hence the first visit estimate is the same of the maximum likelihood model estimation. So the **first visit model is equivalent to assuming a maximum likelihood model with respect to the data**. We are not saying that first visit is better than every visit but only that it is equivalent to the value estimation of the maximum likelihood model. We do not know p but using first visit is like we are using the p that comes from maximum likelihood method. If we were able to say which estimation is better between the V^{FS} and V^{ES} there would be no problems. The maximum likelihood model is not correct, is just a solution, not the true one since p is estimated considering only that single episode, that in case of stochastic behaviour is not much representative. Not knowing the true value of p we cannot say which estimation is better, so we are only seeing their prediction, we are not evaluating the quality of the prediction.

Example - Black Jack speed of learning Evaluating the policy for which you stand if the sum of the cards is 20 or 21, else ask for another card. The montecarlo prediction approach after 10000 plays of the game was not able to have a good estimation of the value function for this part of the state space. This behaviour is quite typical since montecarlo estimation, especially the first visit which is unbiased, is very sample inefficient because the variance is very large, especially in cases like black jack where there is a lot of stochasticity, which means a lot of noise and many samples to have a good estimate of the performance of a given policy.

What follows is useful to introduce the temporal difference approach.

We have said that the MC approach computes a sample average of the returns samples measured during the episodes, so the estimate is simply an empirical mean over the return. What we have seen is a batch formula, that uses a batch of episodes together. From a computational point of view, it would be nice if we could calculate incrementally the empirical mean of the return such that $\hat{\mu}_k = f(\hat{\mu}_{k-1}, x_k)$. IN this way if we observe a new episode x_k , we can update the mean without recalculating it from scratch.

$$\begin{aligned}
\hat{\mu}_k &= \frac{1}{k} \sum_{j=1}^k x_j \\
&= \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right) \\
&= \frac{1}{k} \left(x_k + (k-1)\hat{\mu}_{k-1} \right) \\
&= \hat{\mu}_{k-1} + \frac{1}{k} (x_k - \hat{\mu}_{k-1})
\end{aligned} \tag{88}$$

that can be also written as

$$\hat{\mu}_k = \left(1 - \frac{1}{k}\right) \hat{\mu}_{k-1} + \frac{1}{k} x_k \tag{89}$$

so the new sample average with k samples can be written as a weighted sum where the old estimate is weighted by $1 - 1/k$, instead the new sample is weighted $1/k$. This is computationally good since you can update the sample average without storing all the samples of the past but simply storing their sample mean, so that the samples can be deleted since are no more needed for the next computations (using a batch update, each time we would need to use again all the samples). This is more important as the number of samples we observe grows, and keeping them in memory could be prohibitive in some cases. In the MC case, using (88) we have that we can update $V(s)$ incrementally after episode $s_1, a_1, r_2, \dots, s_T$. For each state s_t with return v_t

$$N(s_t) \leftarrow N(s_t) + 1 \\ V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)} (v_t - V(s_t))$$

* where $N(s_t)$ is the number of times we have calculated the return for s_t , and the values on the left are the new one (k -th estimate) and the one on the right are the old ones ($(k-1)$ -th). The advantage is again that we do not have to store all the v_t from the past episodes, but all I have to store for all the states is the estimation $V(s_t)$ and the number of visits $N(s_t)$.

This is done if you want to compute a sample average and is reasonable for stationary problems, since you want to give the same importance to the samples you have collected because they all come from the same process. But when you are in a non-stationary process, for example when the environment changes or when the policy changes or there may be some non-stationarity in how you are updating and learning the values. This updates can be extended to non-stationary problems. In this cases is useful to forget old episodes, that is controlled by a generic learning rate α

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t)) \tag{90}$$

where α is a learning rate between zero and one, that is the indicator of how much importance you want to give to new samples with respect to old ones. Usually you take:

$$\alpha \geq \frac{1}{N(s_t)}$$

giving more importance to the new information w.r.t. the old one, since in non-stationary problem the old information is surely less relevant since is related to a problem that has been changed. Increasing the learning rate result in the past to be forgotten giving more importance to newer information, because the weighting of $(v_t - V(s_t))$ doesn't converge to zero as in the case of $\frac{1}{N(s_t)}$, but remain constant.

NOTE If you want to go back to MC, you simply choose $\alpha = \frac{1}{N(s_t)}$, going back to the computation of the empirical mean giving the same weight to the episode you see, no matter if they are present or past ones.

NOTE The above expression can be written as

$$\begin{aligned} V(s_t) &\leftarrow V(s_t) + \alpha(v_t - V(s_t)) \\ \implies V(s_t) &\leftarrow (1 - \alpha)V(s_t) + \alpha v_t \end{aligned}$$

Note - MC characteristic recap

- Must work on **episodic MDPs**. To learn it use the entire episode.
- Unlike dynamic programming, MC can evaluate at each episode only one choice at each state, i.e. only one action is considered in each state (the one chosen by the agent in the episode/trajectory). Indeed in MC we do not have the model, so we cannot simulate all the actions, but can only be observed the effect of the action actually chosen by the agent.
- MC doesn't bootstrap. It means that MC approaches does not use the Bellman equation. In the Bellman equation the value is estimated considering the immediate reward and the value of the state that is reached, MC instead never uses the value of the state that are encountered during the simulation, it only looks at the immediate reward collected during the trajectory (i.e. consider the reward step by step and not the value estimation recursively). Hence is not bootstrapping since is not using the value V of other states, but simply sums of immediate (real and not estimated) reward.
- Time required to estimate one state does not depend on the total number of states, but on the variance of the return: if the variance of the return is large you will need a lot of samples with MC, instead if the variance is small the learning can be faster since it requires less episodes.

10.1.2 Temporal difference reinforcement learning (TD(0))

Before explaining this new algorithm, it's worth recalling a very important learning rate property.

Proposition 10.1. *Let X be a random variable in $[0, 1]$ with mean $\mu = E[X]$. Let $x_i \sim X$, $i = 1, \dots, n$ be n i.i.d. realizations of X . Consider the following exponential average estimator*

$$\mu_i = (1 - \alpha_i)\mu_{i-1} + \alpha x_i$$

with $\mu_0 = x_1$ and α_i 's are learning rates.

If $\sum_i \alpha_i = \infty$ **and** $\sum_i \alpha_i^2 < \infty$, **then** $\hat{\mu}_n \xrightarrow{a.s.} \mu$. **The estimator** $\hat{\mu}_n$ **is consistent.**

hence the series of learning rate needs to diverge and the series of the square of the learning rate to converge. For example $\frac{1}{i}$ satisfies the above condition, and the exponential average gives us the empirical mean $\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^n x_i$ which is a consistent estimator (according to the strong law of large numbers). But even if we do not want to use $1/i$, the above proposition says that a learning rate that respect the above conditions creates a consistent estimator. For this reason in our update rules we will choose a learning rate that satisfy this condition. The very good thing about the estimator above is that, manipulating the learning rate, we can interpolate between the previous estimator value and the new sample. With $\alpha_i = 0$ we are ignoring new sample and we keep unaltered our estimate. With $\alpha_i = 1$ we forget the previous estimate and we fully rely on the new sample. With α_i between zero and one we can decide how much we want to rely on the past or on the new sample. Keep in mind this concept, because it will be used for the temporal difference approach.

NOTE The two condition of the proposition are telling us that the series of learning rate must go to 0 not to fast ($\sum \alpha = \infty$) and not to slow ($\sum \alpha^2 < \infty$). Hence there is a boundary inside which the learning rate should decrease.

TD are a **model-free** methods which learn directly from the episodes experience. The idea is to mimic the Bellman equation, indeed this method can be seen the reinforcement learning version of the Bellman expectation equation, having the difference that the Bellman expectation equation requires the knowledge of the transition model, since it computes the expected value with respect to the policy and the transition model computing a summation over all the possible combination of state and action, but this cannot be done in RL because the model is unknown, hence the summation is done by looking only to samples of trajectory, i.e. computing the Bellman equation only in the state and action that have been shown to the agent.

- TD methods learn directly from episodes of experience, i.e. from the agent interactions with the environment, and not from the model.
- TD is a model-free approach since in the computation the model never appears, so no knowledge of the MDP transitions are needed, since the information needed are all implicitly in the update rule.

- TD can learn from incomplete episodes, i.e. does not need to wait the end of the episode to update the parameters. This is possible since it uses the Bellman equation so after each single step it can create a target by using the value estimated for the state that has been reached. So after each step I can have a value, a sample, to update the estimate, indeed I am using the values that I am estimating in the other states. This means that it does bootstrapping.
- TD updates a guess (estimate) towards a guess (estimate), indeed to estimate the value of a state I am using the estimate value of the state that I have reached in the episode. As we can imagine **updating an estimate using another estimate we are introducing a bias**, but the advantage in TD is that this bias introduction is compensated by a very large reduction in variance. Again as usual is a matter of bias variance trade-off. As we will see TD produces a consistent estimator, so with a higher number of samples the bias vanishes.

The goal of TD prediction is to **learn V^π online from experience** under policy π . Differently from MC, it **can use incomplete episodes to make guesses about the value functions**. To do this, we **have to estimate the return from a given state**. For MC we have

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t))$$

where in particular $\alpha = 1/N(s_t)$. In the latter formula v_t is the sum of the discounted reward that I have measured in the episodes performed by the agent

$$v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \dots$$

For the simplest TD learning algorithm (TD(0)) update the value $V(s_t)$ towards estimated return $r_{t+1} + \gamma V(s_{t+1})$. To do this we can simply substitute v_t with $r_{t+1} + \gamma V(s_{t+1})$

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

Doing so, we can use partial rollout of an episode, because we don't need the entire episode to estimate r_t . This practice is called **bootstrapping**. Furthermore, $r_{t+1} + \gamma V(s_{t+1})$ is called **TD target** which is the value towards which we want to move the estimate $V(s_t)$, while the speed at which we go towards the target is regulated by the learning rate α . In particular, $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called **TD error**, i.e. the difference between the new value towards which I want to update the estimation and the value of the estimate. Again, as we saw for the Bellman operator, if the TD error is large it means that the current estimate is far from the fixed value, so the update will be high. Again the TD target mimic the Bellman expectation equation, with the difference that the Bellman expectation equation performs this update for all the actions assuming all the possible transitions to the next state weighted by the probability of the transition. Here instead of doing all the actions and transitions, is simply used one action and one transition that is the one chosen by the agent in the observed trajectory. Hence we are sampling the policy and the model in one point and I am using this sample to update the estimate of the utility of the state s_t . The main difference with

MC is that while MC is using a real reward (v_t is the sum of measured rewards), TD is using only the first real reward while all the other are replaced with the value estimated in the next state, that is not a real reward but something estimated. As we go in the process and we collect many data we have that all the estimate will converge to the true value. In the beginning I have, of course, random variable since we start from an arbitrary value function. as we will see this update rule will make the estimate to converge to the fixed point of the Bellman equation and so to the solution we are looking for. Thus the main drawback of MC is solved by TD, indeed MC works only on episodes, instead in TD after each single step the value estimation is updated. MC needs to see all the rewards until the end to compute v_t , in TD only the first reward is needed and then once the new state is known the value estimation can be updated. Hence TD does a step by step update without having to wait the whole episode to be completed. Not only it has advantages in episodic problem since it can use the information as soon they are available without having to wait the end of the episode, TD can be applied also to non episodic problem, i.e. **problem with infinite horizon**. Since the first samples that I am using are random guesses, since in the beginning the value estimates are random guesses, what I want is to forget the first samples, that's why we want a learning rate that is not equal to $1/N(s_t)$, since the first samples used for the very first estimates are very bad samples influenced by the random initialization of the estimate, hence I want to forget them. In order to forget the far samples we should have $\alpha > 1/N(s_t)$ so that more importance is given to the newer samples that are closer to the correct value, forgetting the older ones that are where generated using not precise estimation.

NOTE Having a large TD error means that what we expect (i.e. estimation $V(s_t)$) is very different from what we observe (i.e. what the world returns to you $r_{t+1} + \gamma V(s_{t+1})$), we change accordingly the estimation in order to learn what is happening in the world.

Example - MC vs TD update Let's consider for example a guy driving home. The information we have are the following:

- The elapsed time can be considered as the reward (in particular is a cost, so the reward can be obtained considering the negative time). The slightly different definition is not important since we are doing prediction (how much t) and not control (i.e. optimization).
- In each state you have a predicted time to go, that represent what you have learned, i.e. the value function, that we want to minimize since is the time that takes to reach home from that state.
- The predicted total time is the sum of the two above.

Considering that the states in the image represent an episode of experience, how MC and TD will update the predicted time to go?

- The MC method would update (when the episode is terminated) using

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t))$$

that for example for the state "leaving the office" is

$$V(s_t) \leftarrow \underbrace{V(s_t)}_{30} + \alpha \underbrace{(v_t - V(s_t))}_{43-30}$$

since $v_t = 43$ is the discounted sum of the reward. To better understand the value of v_t , considering the state "reach car, raining" we obtain

$$V(s_t) \leftarrow \underbrace{V(s_t)}_{35} + \alpha \underbrace{(v_t - V(s_t))}_{(43-5)-35}$$

where $(43 - 5)$ is given by the fact that the elapsed time is a cost

- The TD method would update step by step, so the value for a state ("leaving office") is updated when the next state is reached ("reach car, raining"), with

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(r_{t+1}) - V(s_t))$$

that for example for the state "leaving the office" is

$$V(s_t) \leftarrow \underbrace{V(s_t)}_{30} + \alpha \underbrace{(r_{t+1} + \gamma V(r_{t+1}) - V(s_t))}_{5+35-30}$$

where the update is done considering the immediate reward of the new state and the reward of the next state is instead considered using the estimated value of the new state. For the state "reaching car, raining" instead we have

$$V(s_t) \leftarrow \underbrace{V(s_t)}_{35} + \alpha \underbrace{(r_{t+1} + \gamma V(r_{t+1}) - V(s_t))}_{(20-5)+15-35}$$

where the $(20 - 5)$ is given by the fact that the elapsed time is a cost.

In the following the image show how the states are updated

- Changes recommended by MC method ($\alpha = 1$) It shows that in MC you are **moving all prediction toward the actual outcome** (look at the level at which the arrows are pointing).
- Changes recommended by TD method ($\alpha = 1$) It shows that in TD you are **moving all prediction toward the prediction in the next state** (look at the level at which the arrows are pointing).

Note - MC & TD rewriting We can observe that the MC and TD value function updates can be rewritten as we have seen in proposition 1.1. For example for MC

$$\begin{aligned} V(s_t) &\leftarrow V(s_t) + \alpha(v_t - V(s_t)) \\ &\leftarrow V(s_t) + \alpha v_t - \alpha V(s_t) \\ &\leftarrow (1 - \alpha)V(s_t) + \alpha v_t \end{aligned}$$

As we have seen before, α balances the importance of new data versus old estimations.

We are sure that TD is a fairly biased, but consistent estimator. The high bias generates from bootstrapping, because at the beginning of the learning phase we rely on random, and so biased, estimates of the value-functions to estimate the return v_t .

Note - MC vs TD comparison

- **Episode termination**

- MC must wait until end of episode before return is known. For this reason MC can only learn from complete sequences, so only works for episodic (terminating) environments
- TD can learn online after every step, i.e. before knowing the final outcome. For this reason TD can learn from incomplete sequences, so it works in continuing (non-terminating) environments.
- **Bias-Variance** The return $v_t = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-1} r_{t+T}$ (i.e. the first visit method) is an unbiased estimate of $V^\pi(s_t)$, instead the **TD target** $r_{t+1} + \gamma V(s_{t+1})$ is a **biased estimate** of $V^\pi(s_t)$ (unless $V(s_{t+1}) = V^\pi(s_{t+1})$, but until it converges the estimation are biased). But the **TD target has a much lower variance** indeed

- the return depends on **many** random actions, transitions, rewards. Indeed it uses many values (rewards) coming single realization
- TD target depends on **one** random action, transition, reward. Indeed it uses an estimate value that is an average across many realizations, thus having less variance.

indeed the r_{t+1} reward is present in both, instead the following rewards in TD are replaced with an average term V , and we know that averages has less variance than single realizations. The main difference is in the bias variance trade-off is that

- MC (especially first visit) has high variance, zero bias. MC returns in non-Markovian problems. Indeed it does not exploits the Markov property of the model because it is looking at sequences of rewards without never bootstrapping, i.e. looking at what is the estimated value for the next state.

- TD has low variance, some bias (a lot in the beginning and then vanishes through time). TD is more used when the problem is Markovian, since it exploits the Markov property, i.e. it leverages the Markov assumption. Thus TD get a lot of advantages if the problem is an MDP.

So the advantages of TD comes out when the MC estimation has an high variance, for example having a lot of stochasticity in the MDP. There is not a winner, it depends on the case. For example in a deterministic MDP or with a very low stochasticity, variance is not an issue, so MC work fine, on the other hand with a lot of variance TD works better.

Bias-Variance comparison between MC and TD

- MC has high variance and zero bias.
 - Good convergence properties
 - Works well with function approximation. Assuming you cannot store a value for each function you have to use some kind of function approximation (e.g. gaussian process, linear approximator, neural network). This is not a problem for MC approach since it has to wait for the episode and once the end of the episode is reached it has a target value, that is the return associated to each state that was visited during the trajectory. So in that case you can use traditional supervised learning.
 - Not very sensitive to initial value, since with MC we can simply average the samples that we have, so we can even forget immediately the initial value and then average the new samples as we collect them.
 - Very simple to understand and use.
- TD has low variance and some bias
 - Usually more efficient than MC, especially with a small number of examples. Having a small number of example is the case in many RL problems, because collecting samples means interacting with the environment, hence the speed to collect example is very slow, so usually you do not have time to collect such a large number of examples.
 - TD(0) converges to the true value function $V^\pi(s)$ (as for the MC approach), which means that the introduced bias disappear in the limit of infinite explorations, so the estimator is consistent.
 - Has problems with function approximation, indeed differently from MC that uses samples that are unbiased examples, TD produces samples that are biased, and using them with function approximation make things go wrong. This happens since when you want to update your function approximation in a state (point) that you have visited, you use the value of the estimator in another state (point),

so not having a tabular approximation (a value for each state) but a function approximation (e.g. linear model), the value that you are updating is influenced by the value you have estimated in another point introducing a loop of errors that can make your approximator oscillate or diverge (especially with neural networks).

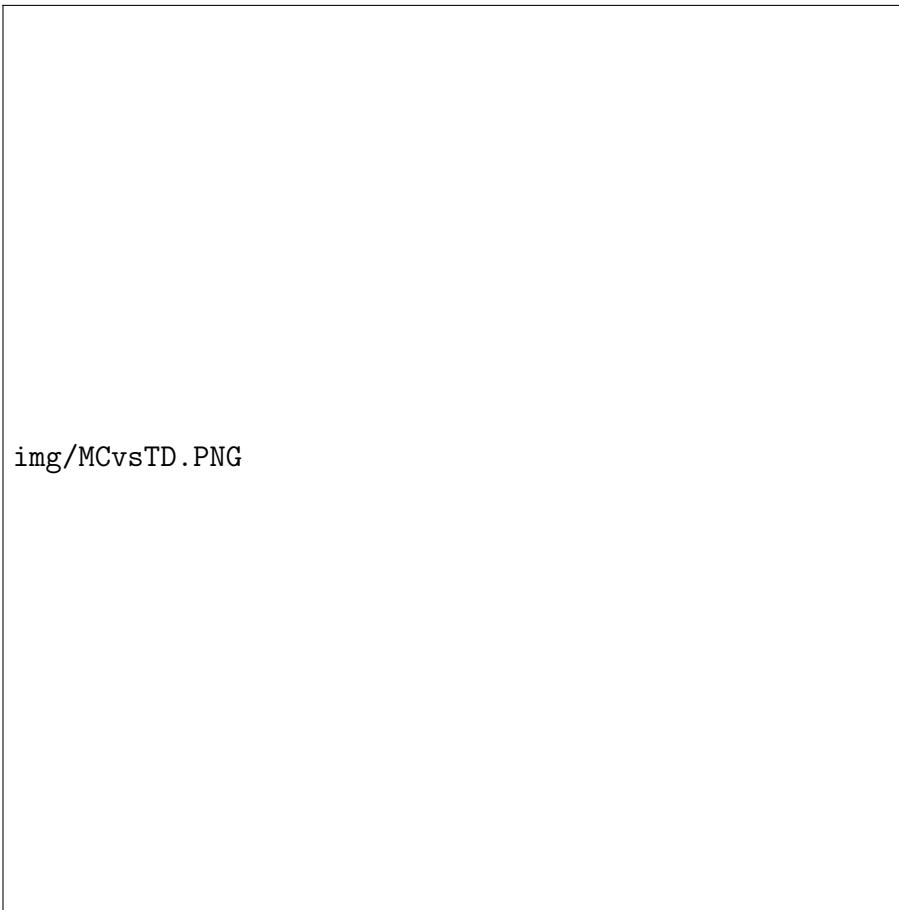
- More sensitive to initial values since they are used to build the target and so at the beginning our estimate are a lot conditioned by our initialization of the value function. This is the reason for which using TD requires a higher learning rate w.r.t. MC, because we want to forget the initial value faster.

NOTE In MC $V(s_t) \leftarrow v(s_t) + \alpha(v_t - v(s_t))$ is done for each trajectory, resulting at the end in a average between them. Building a tree where $i - th$ level correspond to the $i - th$ state in the trajectory, a trajectory would connect the root to one leaf. In MC each update uses a trajectory, i.e. visit the three until the leaf, instead TD each update use only the connection with the $(i + 1) - th$ level and the estimate of the next state. So in the MC case, if a state is visited at the same step by more trajectories is not considered, because you are simply taking different trajectories whose return is different, which means introduction of variance. While TD averages the value of the trajectory passing from the same state at the same step, considering the estimated value of the state, that already averages what may happen in the next state, i.e. the trajectory departing from that state, which has the effect of reducing the variance of the following reward, because you are using an average to update an average, and an average has less variance than a single example.

Example - Random Walk Suppose we have the following chain problem where the squares are the absorbing states. Notice that the reward is zero everywhere except for $E \rightarrow abs..$. We want to evaluate the performance (value function) of the random policy, i.e. the policy that randomly choose the action left or right. Using temporal difference we have where the value of all the state was initialized to 0.5, and the number 1, 10, 100 indicate the number of updates. Hence TD by repeatedly visiting the random chain approximate better and better the value function of the random policy. The following image shows how the error (RMS over the states) in the value function reduces , as the number of examples increases. In particular it compares MC (in black) and TD (in grey) approaches. The fact that the error does not goes to 0 is due to the fact that the learning rate considered is constant, instead it should decrease as the number of sample increase. We can notice that MC is reducing the error much slower w.r.t. TD. This is not surprising because there is a lot of variance in the problems and MC struggles with it. Furthermore the increase of the learning rate let you to have a faster reduction of the error but at a certain point the error becomes worse, lowering the performance. This phenomenon is caused by the fact that the estimator has the variance, so if the learning rate is too large it means that the examples are not averaged so well since we are giving more importance to newer episodes. This problem particularly affects MC since this method has a high variance, that a higher learning rate is not able to reduce. We can also notice that TD can afford much higher learning rate than MC, due to the fact that TD has a small variance and so is able to learn even with few example and

high learning rate. Hence the above picture shows the behaviour of an high variance-low bias estimator and the one of a low variance-high bias one. It's clear how with few examples TD (i.e. having low variance and high bias) is much more effective than MC (i.e. having high variance and low bias); and by reducing the learning rate through time is possible to choose the best of the two approaches.

Markov property TD exploits the Markov property, indeed the TD update rule is simply the Bellman expectation equation that is sampled in one action and one next state. Hence is exploiting the Markov assumption, that says that being in a state and choosing an action, the probability of the next state depends only on the current state and the current action. For this reason TD is usually more efficient in Markov environments. Instead, MC is not using the Markov property anyway since it works on trajectories without looking at the states visited but only considering the reward of the trajectory. For this reason is not aware to be in a Markov decision process, and so usually is preferred and more efficient in non-Markov environments, so that it is not sensible to the lack of the Markov property exploitation.



RMS error averaged over states

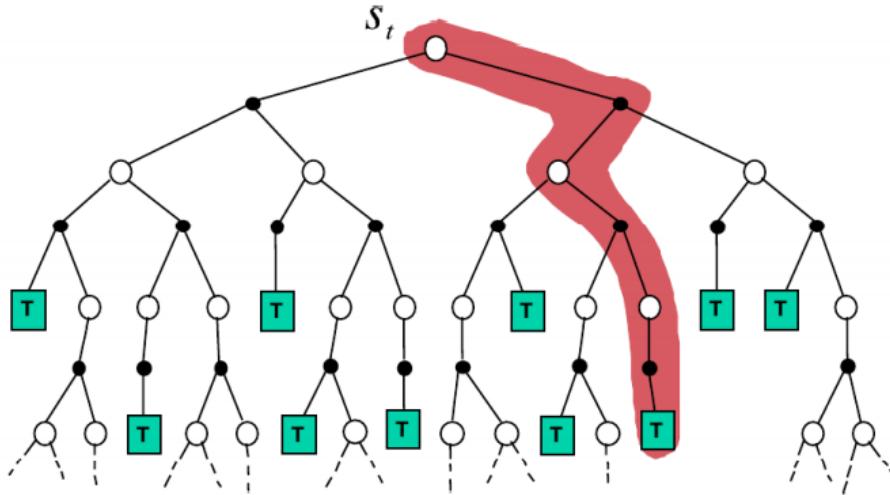
From the figure above we can observe some interesting properties. TD converges faster and has better results. The learning rate influences the convergence speed. Higher values will

make the estimator converge faster, but the error will be higher. In TD this is due to the introduction of bias when we rely too much on new samples. When α is too large we are estimating the value function of a state, with another biased estimate of another state. This can clearly introduce some problems. MC suffers as well as TD with too high learning rates. The largest problem arises because of the large variance. We can see how with $\alpha = 0.04$, MC becomes very noisy.

MC vs TD vs Dynamic Programming Starting from a certain state we can have a tree of trajectories, where the white dots are states, the black ones are actions and the green squares are absorbing states. For MC we have

$$V(s_t) \leftarrow V(s_t) + \alpha[v_t - V(s_t)]$$

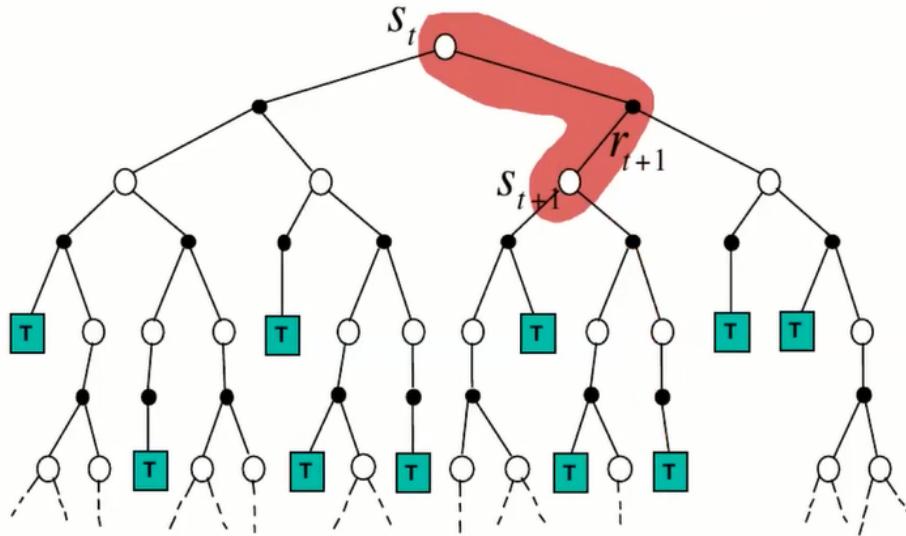
where R_t is the actual return following state s_t



What MC does during an update is colored in red, in particular takes the return as the sum of the discounted reward in the red trajectory. MC visit the tree in depth choosing one branch of the tree at time. Instead for TD we have

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

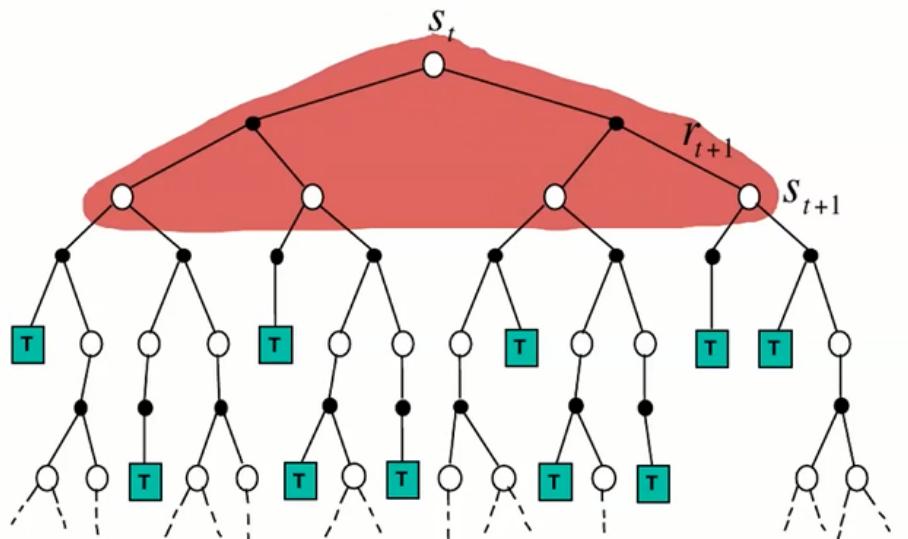
where R_t is the actual return following state s_t



What TD does during an update is colored in red, in particular it performs the update after a single step, choosing only one branch. Starting from one state reaches the next state and uses the value of the next state to update the value of the starting state. So it does not wait to reach the end, but it perform a single action and update the value after this single action. Finally, for DP we have

$$V(s_t) \leftarrow \mathbb{E}_\pi[r_{t+1} + \gamma V(s_{t+1})]$$

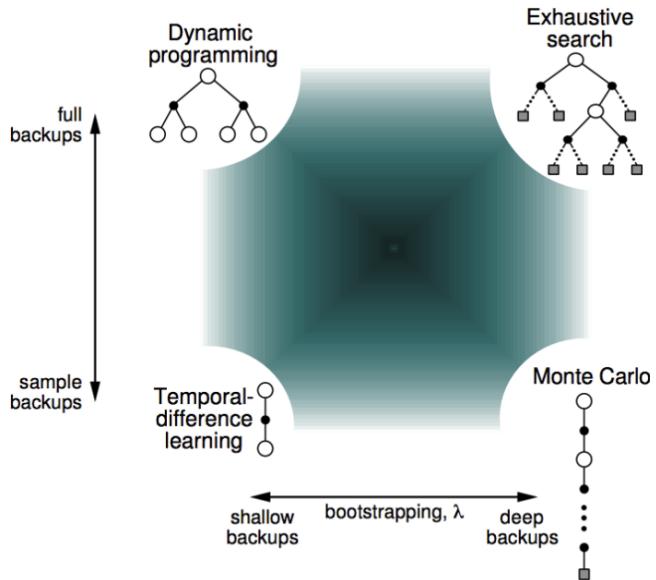
where R_t is the actual return following state s_t



What TD does during an update is colored in red, in particular starting from a state consider all the possible action and all possible next states (all possible branches) and then uses the values of the state that are reached in one step and average them to update the value of the starting state. Hence MC and TD, are approaches that use a single sample (one branch), perform a single realization of the stochastic process, while the DP having the full model updates using the full knowledge (all branches). But again as TD, DP performs one step without reaching the leaf of the tree (absorbing state).

| | Bootstrapping | Sampling |
|----|---------------|----------|
| MC | No | Yes |
| TD | Yes | Yes |
| DP | Yes | No |

TD and DP are said to bootstrap since the value of the reached state is used to update the value of the current state. Instead sampling refer to the fact that the update rule use only one sample (i.e. branch). So DP does not use sampling since the model is known and the value can be computed for all the action and next states (the probability of this events are known). A clearer distinction between the various method

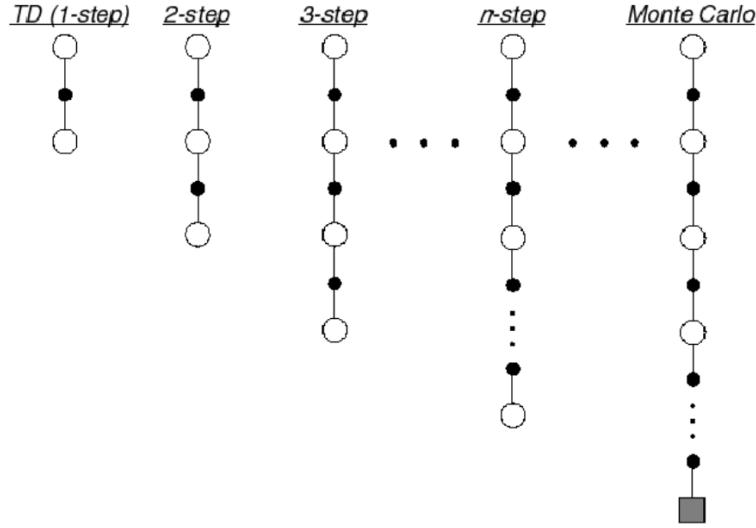


where the exhaustive search is done having the full model and you build the full tree that is fully used to compute the value (but actually is never used). Furthermore sample backups means that the model is not known so they need to sample the model, conversely to full backups where the model is known. Hence on the leftmost side (full bootstrapping) we have the Markovian learning, and on the downmost side (sampling) we have the learning approaches, where the model is unknown.

NOTE DP does not use sampling since the model is known. Indeed it uses the Bellman equation that uses the probability of transition and the probability of taking the action, i.e. you are computing the exact expected value, you are not sampling the model (you do not need to sample the model since you know it).

10.1.3 TD(λ)

We have seen how MC and TD have opposite characteristics in terms of bias-variance. It would be nice if we could choose which properties our algorithm should have. To do so, we can use TD(λ). With the hyperparameter $\lambda \in [0, 1]$, we can swing between TD with $\lambda = 0$ and MC with $\lambda = 1$, regulating the bias-variance trade-off. To understand the TD(λ) approach let's consider this



White circles are states and black dots are actions

Before introducing formally the TD(λ) equation we have to define the $n - step$ returns that we see in the figure above. For $n = 1, 2, \dots, \infty$ the n -step return is

$$v_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$$

where

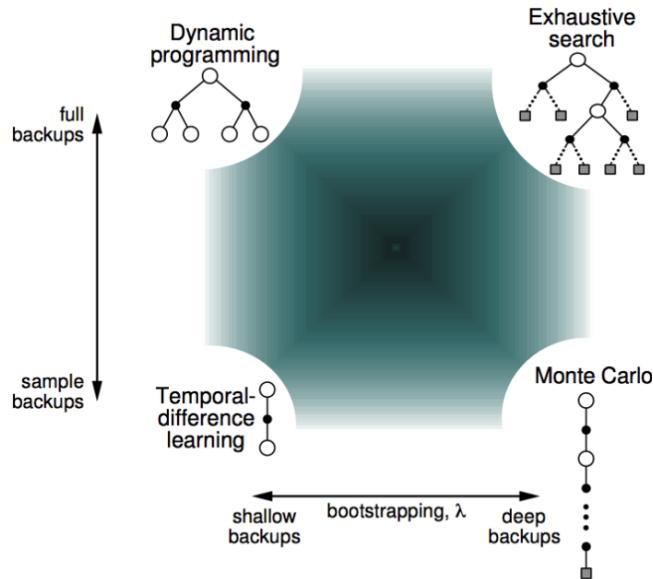
$$\begin{aligned} n = 1 \text{ (TD)} \quad v_t^{(1)} &= r_{t+1} + \gamma V(s_{t+1}) \\ n = 2 \quad \quad \quad v_t^{(2)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) \\ &\vdots \\ &\vdots \\ n = \infty \text{ (MC)} \quad v_t^{(\infty)} &= r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-1} r_T \end{aligned}$$

The $n - step$ return is composed by n discounted immediate rewards, plus the estimate of the state-value function in the upcoming state. So we are using more and more real

rewards and you are using the value function later, multiplied for a discount factor that is $0 < \gamma < 1$, meaning that for greater n the importance of the value estimation decreases since γ^n decreases. The idea is that increasing the number of step before using bootstrap is that you are reducing the bias but you are increasing the variance until the MC case is reached, where the bias is null and the variance is maximum. If we replace this estimator in the TD learning we get the $n - step$ temporal difference learning

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t^{(n)} - V(s_t))$$

Example - Large random walk The following image shows the error in the random walk w.r.t. different learning rates, for different n in the $n - step$ TD



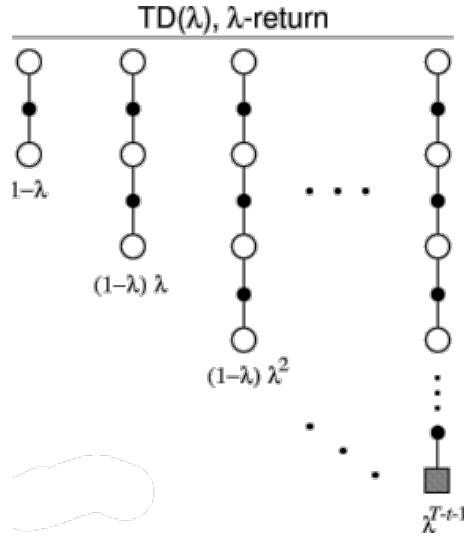
For example, for TD ($n = 1$) perform the best with a learning rate of ~ 0.8 . Increasing the number of step before having the bootstrap, the best performances becomes better and are achieved with smaller learning rates. Continuing the increase of n , the learning rate of the best performances continues to decrease but the performances themselves start to become worse. What is happening here again is the **usual bias-variance trade-off**: when I use TD we have small variance and high bias so with small variance we could use high learning rate. Once we increase the steps before bootstrap we are reducing the bias, so the performance improve because the bias is smaller, but since the variance is higher the learning rate must be reduced. At a certain point we are introducing more bias than the variance that we are reducing so the performance start to get worse. Notice that the plot is made considering only 10 episodes, so we already said that for such small number of episodes an approach more similar to TD is better. Increasing the number of episode we expect that the best method would be closer to MC, since the availability of the episode reduce the variance

of the estimate.

This isn't the TD(λ) formula, but it gives us an intuition about the procedure. The idea of TD(λ) approach is to have an average of the $n - step$ returns over different n , for example averaging the 2-step and the 4-step returns

$$\frac{1}{2}v^{(2)} + \frac{1}{2}v^{(4)}$$

in particular the meaning od the hyperparameter λ is how to combine the different n-step returns together to have a new averaged return. This combines the information from two different time-steps. Can we efficiently combine information from all-time steps? The idea is to weight the different $n - step$ return using the following scheme



where T is the number of step needed to reach the absorbing state. In practice, we perform a weighted average called λ -return v_t^λ , that combines all $n - step$ returns $v_t^{(n)}$ using weight $(1 - \lambda)\lambda^{n-1}$,

$$v_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} v_t^{(n)} \quad (91)$$

where $(1 - \lambda)$ is the normalization constant that all term have in common. We can use this new estimator in the TD learning formula

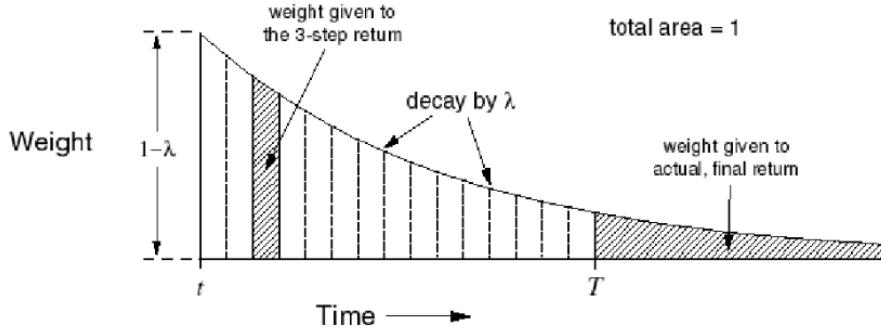
$$V(s_t) \leftarrow V(s_t) + \alpha(v_t^\lambda - V(s_t)) \quad (92)$$

This approach is called **Forward-view** of TD(λ). We can see that for $\lambda = 0$, only the $1 - step$ return have a weight different from zero. On the other hand, for $\lambda = 1$, only the complete episode estimate have a weight different from zero. For values in between, we can give more importance to returns with fewer steps or give more weight to $n - step$ returns with more steps. We can **manage the bias-variance trade-off by changing λ** .

Note - Weights As one can imagine, the weight formulation is very important. One of the most important property is that the **weights must sum to 1**. In our case, the sum of the weight is

$$\begin{aligned} \text{WeightSum} &= (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \\ &= \underbrace{\sum_{n=1}^{\infty} (1 - \lambda) \lambda^n}_{\text{geometric series}} = 1 \end{aligned}$$

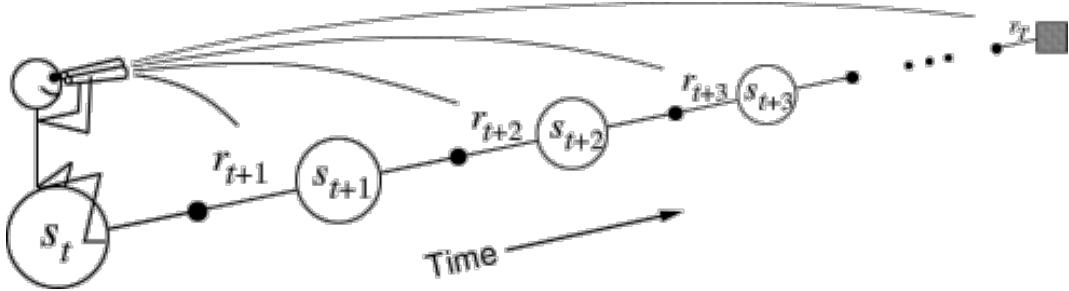
The weighting is exponential and can be represented as follows



By changing λ the weight can be moved all to time t ($\lambda = 0$) or all to time T ($\lambda = 1$) or distributed in between ($0 < \lambda < 1$). Again, by moving the weights we are moving the bias-variance trade-off:

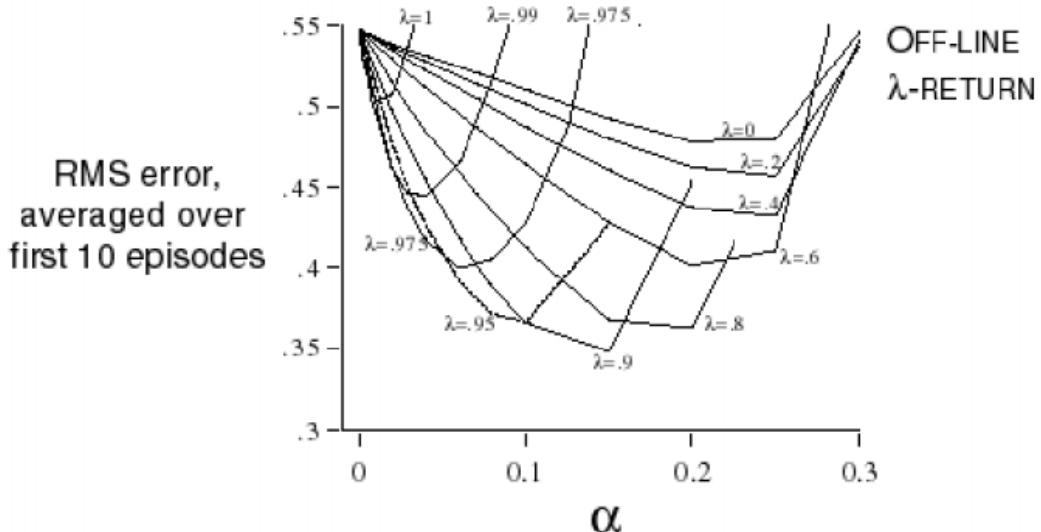
- Small λ : high bias and small variance (closer to TD $\iff \lambda = 0$)
- High λ : small bias and high variance (closer to MC $\iff \lambda = 1$)

To recap, we want to estimate the state-value function of a state. Starting from the given state, at each step, we perform an action, stretching the trajectory length. Every step, we calculate the n-step return. To estimate the starting state value function, we average over the n-step returns obtained at each step, weighted with the hyperparameter λ . The approach is called Forward-view TD(λ) since in order to update, we need to know all the n-step return, which means that like MC we still need to perform every step until we reach the end of the episode, because v_t^λ still need the n-step return associated to the complete episode. This means that we have the same limitation of MC method, it is an episodic method that needs to wait the end of the episode before updating the value.



The above image clearly show how the update of the value function, being done towards the λ -return, needs the complete episode (like MC) since it have to look into the future to compute v_t^λ .

Example - Forward-view TD(λ) on large random walk The following figure shows how the error of the estimator changes with λ and w.r.t. the learning rate α



Increasing λ we are reducing the bias so we have an improvement, but at a certain point increasing the λ too much we are introducing too much variance so the performance become worse. As the variance increase with the reduction of λ also the best learning rate decreases. As before the error is evaluated considering only 10 episodes, too few to use MC approach.

To solve the problem of episodic updates, to be able to update the value at each step and not after each episode, we can use the so called **Backward-view TD(λ)**.

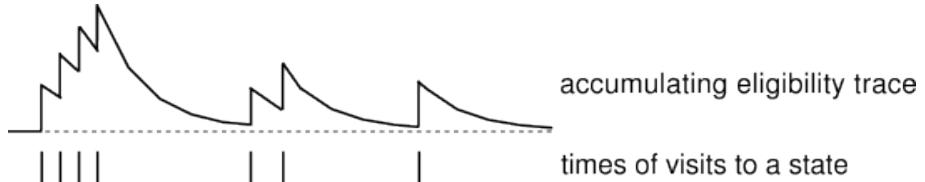
The forward-view provide us some the theory necessary to approach the problem. Backward-view gives us the mechanism. Our **objective** is to generate an algorithm capable of **updating our estimate of the value function at each step**. Our first step is to define the **eligibility traces**. The eligibility traces are used to solve the **credit assignment problem**, which consist in associating the merit of a given reward to a state. Given a trajectory,

which states have influenced the most the reward? The eligibility traces are one of the many heuristic for solving the problem,

- **Frequency heuristics** We assign credits to the most frequent state, i.e. visited more frequently.
- **Recency heuristics** We assign credit to the most recent states
- **Eligibility traces** We combine the frequency and recency heuristic

$$e_{t+1}(s) = \underbrace{\gamma \lambda e_t(s)}_{\text{recency term}} + \underbrace{1(s = s_t)}_{\text{frequency term}} \quad (93)$$

where $+1(s = s_t)$ means that we add one only if the current state s is equal to s_t and $e_{t+1}(s)$ is the new value of the eligibility trace in the state s while $e_t(s)$ is the old value. The eligibility trace is **calculated in every state, and is update at each step** with the above computation. The recency term makes the eligibility trace decay (decreases) over time because γ (discount factor) and λ (TD hyperparameter) are less than one, thus is the product. The frequency term adds one to the trace every time the state is visited, i.e. if in the step t I am in the considered state. Hence for all the states that are not visited at the time t the eligibility traces reduces of $\gamma\lambda$ factor, while for the state that was visited the eligibility trace is reduced by the same factor but then is incremented by 1 unit.



So the eligibility trace at the beginning is 0 since the state was never visited and then it increases as soon as the state is visited and decreases continuously. Hence, uses both the heuristic to take account how much important is a state for the reward that I am observing. Notice that if $\lambda = 0$ (i.e. TD) the eligibility trace coincide with the frequency heuristic, so you update only the state you visited, while if $\lambda = 1$ the eligibility will decrease slowly at a rate defined by only the discount factor, which means that you give a lot of importance to state visited a lot of time steps before, which is similar to what MC does. Indeed MC propagates much more information since updates all the state of the trajectory, not only the previous state like TD(0). Hence **by regulating λ you change how much you want to propagate in the past the information you have collected by now**.

We have seen how the eligibility trace $e_t(s)$ measure the contribution of state s to the return. We can rewrite the state-value function including this term. Starting from TD we update the value $V(s)$ for every state s in proportion to TD-error δ_t and the eligibility trace $e_t(s)$

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad (\text{TD})$$

$$\begin{aligned}
&\leftarrow V(s_t) + \alpha(\delta_t) \\
V(s_t) &\leftarrow V(s_t) + \alpha(\delta_t e_t(s)) \quad TD(\lambda) \text{ Backward-view}
\end{aligned}$$

$TD(\lambda)$ Backward-view uses the eligibility traces to give more importance to the most recently or more frequently visited states. Hence it updates all the state that have some kind of eligibility of the update. The **states with high eligibility trace will rely more on newly seen data, by giving a boost to the learning rate.**

Here we have the algorithm for backward-view $TD(\lambda)$

Algorithm 4: backward-view $TD(\lambda)$

Output : V^π

Input : $S, A, \pi, \gamma, \lambda$

Initialize: $V(s)$ arbitrarily

for all episodes **do**

$e(s) \leftarrow 0, \forall s \in S$ (eligibility null in every state)

$s \leftarrow$ starting state (initialization)

repeat

$a \leftarrow$ action given by π for s

Take action a , observe the reward r , and the next state s'

$\delta \leftarrow r + \gamma V(s') - V(s)$ (TD-error)

$e(s) \leftarrow e(s) + 1$ (update eligibility only for the visiting state)

for all $s \in S$ **do**

$V(s) \leftarrow V(s) + \alpha \delta e(s)$

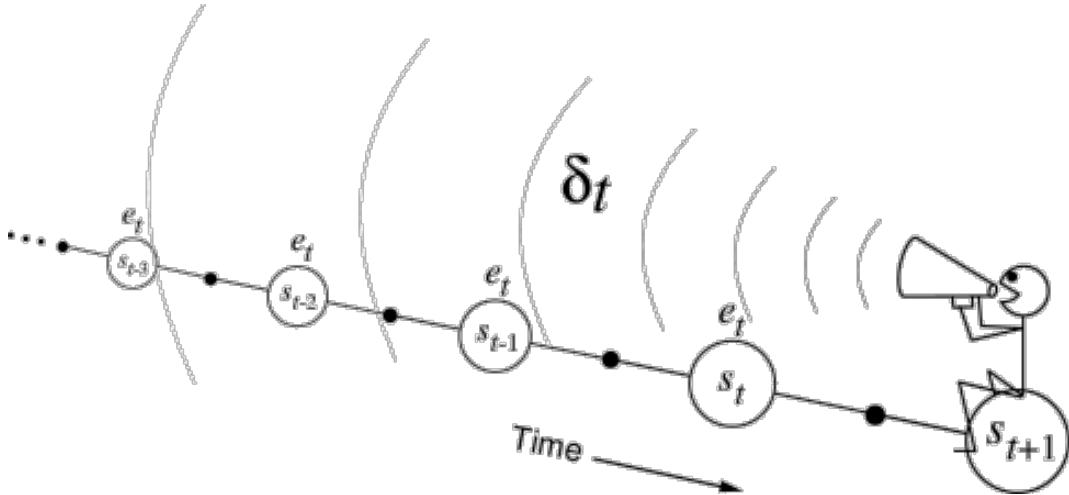
$e(s) \leftarrow \gamma \lambda e(s)$ (update eligibility)

end

$s \leftarrow s'$

until s is terminal;

end



Hence the reward is propagating back in all the states that have a positive eligibility. Notice that $e(s)$ is decreased ($\gamma\lambda e(s)$) only after being used for the update of the value function in the corresponding state. Again when $\lambda = 0$, only the current state is updated

$$e_t(s) = 1(s = s_t)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t e_t(s)$$

and this is exactly equivalent to TD(0) update

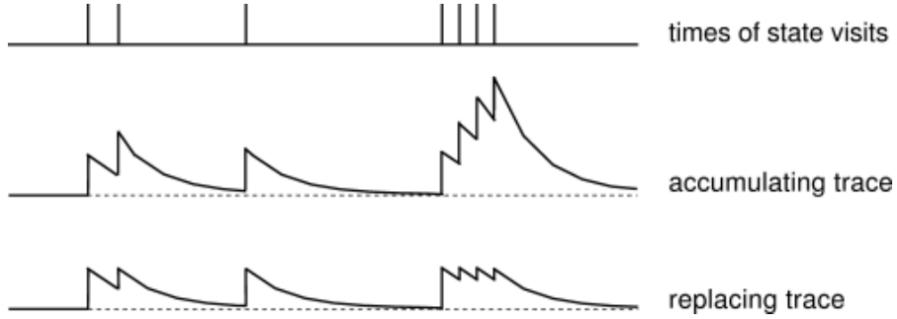
$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t \quad (94)$$

While when $\lambda = 1$ the approach is comparable to MC (there is a small difference between the value computed by TD(1) and MC), so TD(1) can be considered a good approximation of MC approach.

Using accumulating traces, frequently visited states can have eligibilities much greater than one. This very large values of elibility introduce a lot of variance, making the learning not so stable and can be a problem for convergence. **Replacing traces**, are eligibility traces in which you bound the maximum value for the eligibility to 1. Instead of adding 1 when you visit a state, set that trace to 1.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), & \text{if } s \neq s_t \\ 1, & \text{if } s = s_t \end{cases} \quad (95)$$

i.e. it simply do not allow eligibility higher than 1. So it bounds the elegibility of the state as it is clear in the following figure



10.2 Model-free control

So far we have seen how to estimate the value function of a given policy. Now our objective is to find the policy π^* , that maximize the value function, i.e. the optimal policy. We already done this in the previous chapter, under the assumption that we know the transition model of our problem. This is no longer true, so we need to modify our algorithms to accommodate this new situation.

Some examples of problems that can be modeled as MDPs are:

- Elevator
- Parallel parking
- Protein folding
- Robot walking
- Game of go

For most of this problems, either:

- MDP model is unknown , but experience can be sampled.
- MDP model is known, but is too big to use (too many state and action variables) and applying exact methods like dynamic or linear programming or control theory methods is not feasible, since this methods do not scale to very difficult problems. So the model is used to generate samples used to learn an approximate solution using RL (an approximate solution is better than not being able to compute any solution due to complexity).

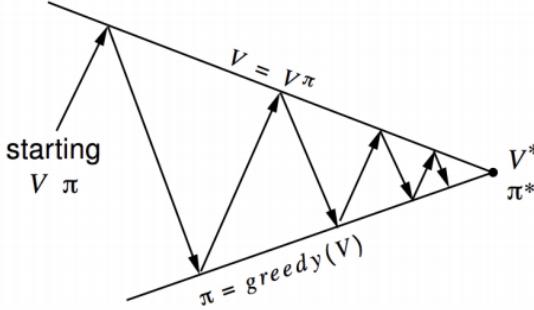
There are two types of learning:

- **On-policy** methods use the data coming from the interaction between the agent and the environment to estimate the value function of the policy used by the agent. So it "learns on the job" i.e. learn about the policy π from experience sampled from π . This is similar to policy iteration.
- **Off-policy** methods learn the policy π from the experience sampled from $\hat{\pi}$, sort of "learning over someone's shoulder"

10.2.1 On-Policy-Monte-Carlo control

This approach is based on policy iteration. As a reminder, policy iteration is divided in two steps, which are iterated until convergence. The first is policy evaluation. In this step we estimate the value function of our policy V^π (e.g. iterative policy evaluation approach, that has a max-norm contraction towards V^π). Once we have an estimate, we can perform the second step, policy improvement (e.g. greedy policy improvement). From the value function

estimate, we can calculate the relative greedy policy, which we are sure is better than the previous policy ($\pi' \geq \pi$). We iterate this two step until two consecutive policy evaluation are the same. The relative policy is the optimal one.



In particular the generalized policy iteration approach, that is a dynamic programming approach must be transformed into an RL approach.

The policy evaluation step uses the Bellman expectation operator that need the transition model of the MDP. To solve this problem, we can use one of the model-free prediction algorithm we have seen in the previous section. For example we can use MC, that is known to converge to V^π . We are done right? Not so fast. Now we can evaluate a policy model-free, but the policy improvement still needs the transition model, because it need to calculate the greedy policy. Having the value function the equation of the greedy policy is the following

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}} \left\{ R(s, a) + \gamma \sum_{s' \in S} \mathbf{P}(s'|s, a) V(s') \right\}$$

that contains the transition model and the reward function that are assumed to be not available. The greedy policy improvement becomes model-free by using the action-value function $Q(s, a)$

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}} \left\{ Q(s, a) \right\} \quad (96)$$

In this way, we don't need anymore the transition model, because the greedy policy can be calculated by choosing the action that maximizes the action-value function in a state. In the policy evaluation step, we are no longer estimating V , but we have to calculate the action-value function $Q = Q^\pi$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \underbrace{\frac{1}{N(s_t, a_t)}(v_t - Q(s_t, a_t))}_{\alpha} \quad (97)$$

where $N(s_t, a_t)$ is the number of times in the state s_t was taken the action a_t and v_t is the usual MC discounted return. Now the greedy policy improvement can be done, but there is a problem. Indeed if we estimate the state-value function Q and you take the greedy

policy as the action that maximize the Q function, i.e. $\pi'(s) = \underset{argmax}{a} Q(s, a)$, the policy that you get (π') is Markovian, stationary but deterministic because the probability of the action that maximizes the Q will be equal to 1. So since π' is deterministic, using the approach of computing the policy and then estimate the value function, is possible to estimate the Q function of the deterministic policy? The problem of estimating the Q function of a deterministic policy using the MC method is that in any state the action performed is always the same so you will never be able to update the Q value of the other actions, but in this way you will never know the performance of the other actions using this approach, since the policy never executes the other actions due to its deterministic behaviour. But not being able to measure the performance of the other action you will never be able to improve the policy, because you do not know what happens if you do something else. Hence in the model-free using deterministic policy you are not able to learn. To learn you need to explore and to explore you need to try all the other actions. For the latter reason we introduce the concept of on-policy exploration.

Example - Exploration Imagine you have to choose between two doors. You open one of them and get a random reward, according to some unknown distribution. You want to find the door that gives you the largest reward.

- You open the left door and get reward 0.
- You open the right door and get reward +1.
- You open the right door and you get reward +3.
- You open the right door and you get reward +2.

Are you sure you've chose the best door? No, indeed the left door was opened only once, and maybe opening it now will give you a huge reward. The question is: when I have to explore, i.e. randomize the action, and when I have to select the action that is performing better? As we can imagine we need to have a trade-off indeed we want to perform the best action, but to know which is the best action I have also to try the action that is not the best one, i.e. take sub-optimal actions, but obviously this sub-optimal action should be taken the minimum time possible. Multi-Armed-Bandit algorithms focus on this problem of **exploitation vs exploration**.

NOTE MAB are useful in case of web advertisement banners, where you do not know which is the best banner to show, so you try them and then see.

NOTE For what concerns which actions are available in which state there are two main approaches:

- The agent can choose among a set of action that depends on the states, so it explicitly knows which actions are available. (most common solution)

- If the information of available actions are not provided explicitly, they are encoded in the reward, so taking an action not available in a certain state will give a very negative reward, so that it can learn to avoid to take that action.

Now we will use a practical solution for the exploration problem, that is not the optimal one, but is simple and practical, showing that we can build model free control algorithm that converge to the optimal value function.

We estimate the value of each state-action pair. Obviously the number of state-action pair are way more than the number of states, so we need to estimate more values than before. To compensate, we need to get more samples. Are we good? Not really unfortunately. Suppose to start our policy iteration with the random policy. We perform the policy evaluation step, and then we calculate the new greedy policy. Usually the greedy policy is deterministic. It means that in every state, we will always choose only one action. This is a problem, because to estimate Q we need to explore all the state-action pair, but being deterministic we take always the same action in a given state. It happens that some state-action pairs will be never explored, and so we don't have a clue on the utility of the pair. The major consequence of this is that we can't use the greedy policy improvement as we have seen before, and furthermore we can't use deterministic policies using policy iteration to estimate Q . Here we can start to see one of the most important concept in RL. The **exploration vs exploitation dilemma**. This concept will be explained better in the next chapter, for now we say that when learning the value function of an MDP, we need to find the right balance between exploring the state-action pairs and exploiting the knowledge collected so far.

One of the easiest way to produce non-deterministic policies during the policy improvement step, is to use ϵ **greedy exploration**. This approach ensures continual exploration, since all m actions are tried with non-zero probability. Indeed having a policy that has 0 probability to an action the risk is that you put 0 probability to the optimal action, so in any state you need to give a positive probability to any possible action. The idea of ϵ -greedy is to give probability $1 - \epsilon$ to the greedy action, i.e. to the action that is associated to the largest estimated Q value, and the remaining ϵ probability is distributed uniformly among all the action that you have. Hence the policy will be calculated as follow

$$\pi(s, a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a^* = \underset{a \in A}{\operatorname{argmax}}\{Q(s, a)\} \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases} \quad (98)$$

The greedy action have a large probability of being chosen, so the value of ϵ regulates the amount of exploration of the agent, i.e. manages the exploration-exploitation dilemma:

- having small value of ϵ the policy will be almost the greedy one
- Larger values of ϵ will generate policies which explore more, because it gives more probability to the non-greedy actions.

- using ϵ equal to one it means that you will use the random policy.

The very good thing about ϵ -greedy policy is its theoretical guarantees, indeed the following theorem states the policy improvement for ϵ -greedy policy:

Theorem 10.1 (ϵ -greedy policy improvement). *For any ϵ -greedy policy π , the ϵ -greedy policy π' with respect to Q^π is an improvement*

$$\begin{aligned}
Q^\pi(s, \pi'(s)) &= \sum_{a \in A} \pi'(a|s) Q^\pi(s, a) \\
&= \frac{\epsilon}{m} \sum_{a \in A} (Q^\pi(s, a)) + (1 - \epsilon) \max_{a \in A} Q^\pi(s, a) \\
&\geq \frac{\epsilon}{m} \sum_{a \in A} (Q^\pi(s, a)) + (1 - \epsilon) \sum_{a \in A} \frac{\pi(a|s) - \frac{\epsilon}{m}}{1 - \epsilon} Q^\pi(s, a) \\
&= \sum_{a \in A} \pi(a|s) Q^\pi(s, a) = V^\pi(s)
\end{aligned}$$

Therefore from policy improvement theorem, $V^{\pi'}(s) \geq V^\pi(s)$

That is the same result we had for greedy policy. This is great news, the ϵ -greedy policy with respect to a policy is for sure better. This is important since we cannot use simple greedy policy in model free control, since it put zero probability to all the action but one, and since for learning we need to try all the action we need to use randomization. With ϵ -greedy policy we can use the randomization but we have still the guarantee, to have a policy improvement. To recap in model-free control we have,

- **Policy evaluation** use MC policy evaluation, evaluating the action-value function $Q = Q^\pi$ (and not the value function V)
- **Policy improvement** we ϵ -greedy policy improvement, (not greedy policy).

As we have seen for DP policy iteration we don't need to calculate the true value function in the policy evaluation step, but we can stop when our approximation is close enough. In this case we have generalized model-free policy iteration

- **Policy evaluation** MC policy evaluation, $Q \approx Q^\pi$

So now all this elements are model free. Now the question is if combining the latter policy evaluation and policy improvement, where I converge? To completely define on-policy MC control, we still need some definitions.

Definition 10.1 (Greedy in the Limit of Infinite Exploration - GLIE). *We say that an exploration strategy is Greedy in the Limit of Infinite Exploration (GLIE) if*

- All state-action pair are explored infinitely many times

$$\lim_{k \rightarrow \infty} N_k(s, a) = \infty, \quad \forall (s, a)$$

- The policy converges on a greedy policy

$$\lim_{k \rightarrow \infty} \pi_k(a|s) = \mathbf{1}(a = \operatorname{argmax}_{a' \in a} \{Q_k(s', a')\})$$

where this are the properties we expect in the limit of infinite explorations, which can be rephrased saying that the exploration of each state is done infinite many times converging to the greedy policy making the exploration disappear. Since we are using an on-policy algorithm, i.e. estimating the value function of the policy that is collecting the samples, you will learn the optimal value function only when you will play the optimal value function, and since the optimal value function may be induced by the optimal policy which may be only deterministic, you need that the policy becomes deterministic in the limit of infinite explorations.

In GLIE MC control we assume to have an episodic problem since we are using MC for evaluation. Given sample k^{th} episode using $\pi : \{s_1, a_1, r_1, \dots, s_T\} \sim \pi$, for each state s_t and action a_t in the episode we have

$$\begin{aligned} N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\ Q(s_t, a_T) &\leftarrow Q(s_t, a_t) + \frac{1}{N(s_T, a_t)}(v_t - Q(s_t, a_t)) \end{aligned}$$

The policy improvement step based on the new action-value function will use

$$\begin{aligned} \epsilon &\leftarrow \frac{1}{k} \\ \pi &\leftarrow \epsilon - \text{greedy}(Q) \end{aligned}$$

where when the number of episodes goes to infinite the value of ϵ goes to 0, so that the exploration goes to 0 and the policy becomes greedy. Following the latter scheme:

Theorem 10.2 (GLIE Monte-Carlo control). *GLIE Monte-Carlo control **converges to the optimal** action-value function, $Q(s, a) \rightarrow Q^*(s, a)$*

but by learning the optimal value function we also learned the optimal policy.

Now we have a way to find the optimal policy with a model-free approach, i.e. a model free control algorithm that guarantee to reach the optimal solution. Remember that the only information given comes from the interaction with the environment.

This method have many hyperparameters which are related to some time scales. We can recap their role and their relationship in order to make things clearer. There are three time scales,

- **Behavioural** related to the discount factor γ . Gamma regulates the horizon, where $\frac{1}{1-\gamma}$ can be considered as (an approximation of) the effective horizon.
- **Sampling** related to the learning rate α for the estimation of Q . Which is a measure of how fast you want to forget the old estimate to incorporate the new knowledge gathered from the environment.

- **Exploration** related to ϵ , for the ϵ -greedy strategy. Regulates the exploration of the ϵ -greedy algorithm.

To have good result during the learning phase, these three hyperparameters (usually) have to respect this relationship

$$1 - \gamma \ll \alpha \ll \epsilon$$

As a starting point we can use $1 - \gamma \approx \alpha \approx \epsilon$. Then we decrease ϵ faster than α . We do so because if the learning rate is too small, while ϵ is still large, the exploration will not be counted, so is easier for the algorithm to learn. When updating our value function, α will weight more the old samples. This in practice, would ignore the newly explored states and actions. Practically, given M trials we can set $\alpha \sim 1 - \frac{m}{M}$ and $\epsilon \sim (1 - \frac{m}{M})^2$ where $m = 1, \dots, M$ is the current number of episodes, and we can notice how ϵ decrease faster. γ should remains constant, because it is a property of the problem we are trying to solve, and not of the learning algorithm we are using. But in practice, γ is initialized to low values, because the problem becomes easier. Then it is gradually moved towards its correct value. This approach is called curriculum learning: changing γ to make the problem easier means introducing a bias, since we are changing the problem, but learning a different easier problem at the beginning will allow you to have a better performance (in terms of learning speed) to learn the original problem, and by gradually switching to the original problem the performance obtained are not so bad. Again this last technique is related to the bias variance trade-off: reducing gamma we are increasing the bias since we are changing the problem and reducing the variance since we are simplifying the problem, i.e. minimizing over a shorter horizon. Increasing gamma as the number of samples increase will reduce the bias and increase the variance, but the latter is reduced thanks to the high number of samples.

NOTE The on-policy Monte Carlo approach is never used, because MC has to much variance for real problems, so since here the problem is sample efficiency, usually Monte Carlo is not used. MC is used only in the policy search case, but in all the value based approaches TD is used.

10.2.2 On-policy Temporal-Difference control

As we did before, we can use temporal difference to solve the policy evaluation step. This approach have, as in the prediction case, a lot of advantages compared to MC.

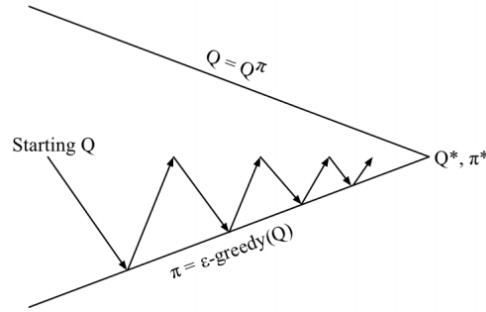
- Lower variance
- it is on-line, i.e. update the estimate after each step.
- works with Incomplete sequences, i.e. you do not need episode to terminate.

The most simple way to adapt the control problem using TD is by applying temporal difference to $Q(s, a)$ in the policy evaluation step. Then, we can use ϵ -greedy policy improvement as before. These steps are done every time we take a new action, i.e. at every time step.

This on-policy control is called **SARSA**³⁸. The idea is to use the following simple update rule, that is the TD update rule for the Q function. As in the MC case we need to evaluate the action-value function

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

where s' and a' are the next state and action, of the policy π . From this the name of the algorithm indeed the update can be done when we have the state, the action, the reward and the new state and the new action.



As we can see from the figure, at each time-step we perform

- **Policy evaluation** SARSA, $Q \approx Q^\pi$
- **Policy improvement** ϵ -greedy improvement

indeed after each time-step, updating the Q function you are also updating the policy because since the policy is ϵ -greedy with respect to the Q function and the Q function is updated at

³⁸SARSA stands for: State Action Reward State Action

every step you are continuously changing the policy.

Algorithm 5: SARSA On-Policy control

Output : Q^*

Input : $S, A, \pi_0, \gamma, \epsilon$

Initialize: $Q(s, a)$ arbitrarily

$$\pi \leftarrow \pi_0$$

do

$$s \leftarrow \text{starting state}$$

$a \leftarrow$ action given by π for s , i.e. using the policy derived from Q (e.g. ϵ -greedy)

repeat

 Take action a , observe r, s'

$a' \leftarrow$ action given by π for s' i.e. using the policy derived from Q (e.g. ϵ -greedy)

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

$$s \leftarrow s'$$

$$a \leftarrow a'$$

$$\pi \leftarrow \epsilon\text{-greedy}(\pi)$$

until s is terminal;

while convergence;

NOTE The initialization of Q can be any, indeed we are applying the Bellman expectation equation, plus the policy improvement. So the algorithm converges starting from any Q function.

The question now is: does the algorithm converge to the optimal Q function? To ensure that the SARSA algorithm converges to the optimal solution, we have to check if some conditions are respected

Theorem 10.3. SARSA converges to the optimal action-value function, $Q(s, a) \rightarrow Q^*(s, a)$, under the following conditions

- GLIE exploration strategy(sequence of policies $\pi_t(s, a)$)
- Robbins-Monro sequence of step-sizes α_t

$$\sum_{t=1}^{\infty} \frac{1}{\alpha_t} = \infty$$

$$\sum_{t=1}^{\infty} \frac{1}{\alpha_t^2} \leq \infty$$

NOTE With MC we did not mention the Robbins-Monro condition since in MC the learning rate is $1/N(s_t, a_t)$ that automatically satisfy the Robbins-Monro condition. While in TD we need to forget the initial values since they are taken at random, so the learning rate should be larger than the one of MC and it has to satisfy the Robbins-Monro condition.

Can we have something in between SARSA and MC?

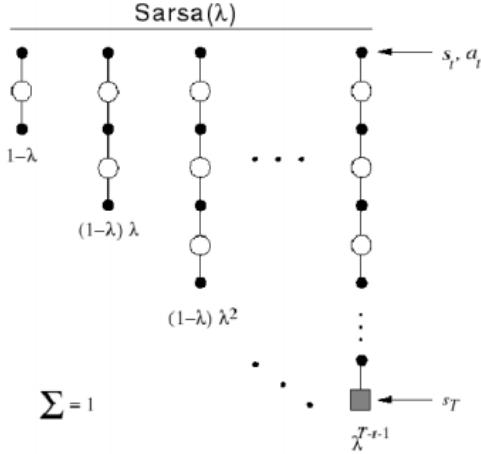
- SARSA has low variance but introduces variance because it uses bootstrap, indeed it uses the value of the next state and next action.
- MC does not perform bootstrap but has a lot of variance.

We can see that for SARSA we are using TD(0), because we are bootstrapping after a single step. As we did before for the prediction problem, we can define a procedure which can swing across MC and TD with a parameter λ . This procedure is called SARSA(λ)

- **Forward view** update action-value $Q(s, a)$ to λ -return v_t^λ
- **Backward view** use eligibility traces for each state-action pairs.

$$e_t(s, a) = \gamma \lambda e_{t-1}(s, a) + \mathbf{1}((s_t, a_t) = (s, a))$$

i.e. the eligibility traces are no more only function only of the state but both of the state and the action. The $+1$ is done only when the same state is visited and the same action is taken.



Algorithm 6: Backward-view SARSA(λ)

```

Output :  $Q^*$ 
Input   : S, A,  $\pi_0$ ,  $\gamma$ ,  $\lambda$ ,  $\epsilon$ 
Initialize:  $Q(s,a)$  arbitrarily
 $\pi \leftarrow \pi_0$ 
do
     $e(s,a) \leftarrow 0, \forall s,a$ 
     $s \leftarrow$  starting state
     $a \leftarrow$  action given by  $\pi$  for  $s$ 
    repeat
        Take action  $a$ , observe  $r$ , and next state  $s'$ 
         $a' \leftarrow$  action given by  $\pi$  for  $s'$ 
         $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ 
         $e(s, a) \leftarrow e(s, a) + 1$ 
        for all  $s, a$  do
             $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ 
             $e(s, a) \leftarrow \gamma \lambda e(s, a)$ 
        end
         $s \leftarrow s'$ 
         $a \leftarrow a'$ 
         $\pi \leftarrow \epsilon\text{-greedy}(\pi)$ 
    until  $s$  is terminal;
while convergence;

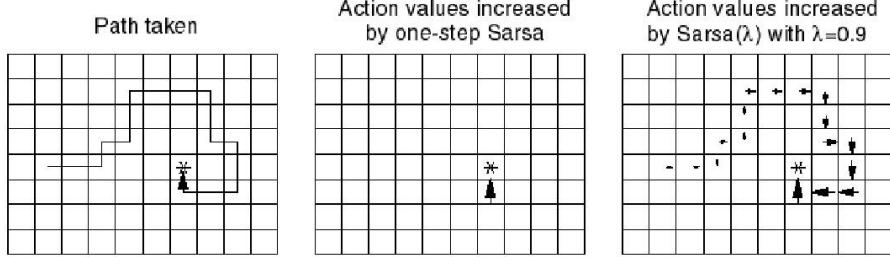
```

The algorithm is the same as SARSA(0), the only difference is that now the eligibility trace is defined over state and actions and all the state action pair must be updated with an eligibility value that is larger than 0. So at each step we will go back into the state action space updating all the Q value in the state action pairs that have been visited in the past using the value of the eligibility. Using eligibility trace you can propagate the information

that you collect step by step much faster around the problem, and this propagation reduces the bias but increases the variance of our estimate.

NOTE The preferred method for what we have seen before is the backward view.

Example - SARSA(λ) Imagine to have the gridworld in the figure below



From left to right: 1.Path taken 2.SARSA(0) 3.SARSA(0.9)

Suppose to perform a complete episode of our MDP and we have estimated the action-value function with SARSA(0) and SARSA(0.9). The first corresponds to the TD approach. In fact, we update only the state-action pair right before the goal state, which is the only one returning a non-zero reward. All the other pairs are not updated because the goal return echoes only to the previous pair on the trajectory. So to propagate the action to the other state you need to visit again the same state, otherwise the propagation does not happen. On the other hand, with $\lambda = 0.9$, the goal return is echoed through all the trajectory. We can also observe that the update decay over the trajectory, as we would expect, due to recency of the action state pair. Indeed:

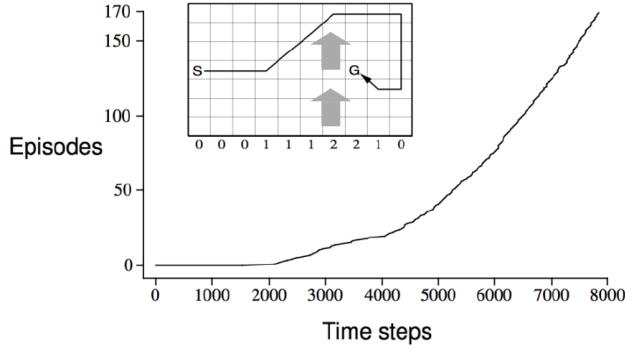
- SARSA(0):

$$Q(s, a) \leftarrow Q(s, a) + \alpha(Q(s, a) + \gamma Q(s', a') - Q(s, a))$$

- SARSA(0.9):

$$Q(s, a) \leftarrow Q(s, a) + \alpha(Q(s, a) + \gamma Q(s', a') - Q(s, a))e(s, a)$$

where the update is done for all state for which the eligibility trace is non-null, that is only the previous for SARSA(0) and all the states that have visited in the past for SARSA(0.9). Instead in the following example:



The slope represent the difference of the length that becomes shorter and shorter, the higher the slopes, the shorter is the number of steps that compose the episodes, At the end the agent converges to the solution of the problem.

10.2.3 Off-Policy learning

Off-policy methodologies try to estimate a policy that is different from the one generating the data. The estimated policy is called **target policy** $\pi(a|s)$, while the data generating one is called **behavior policy** $\bar{\pi}(a|s)$. This could seem counter intuitive, but it can bring a lot of advantages to our learning phase.

- in many cases the environment cannot be directly controlled, but only be observed. Splitting the two policy can give us an advantage when learning from observing humans or other agents. The best policy learned can suggest the human or the other agent the best behaviour.
- reuse experience generated from old policies $(\pi_1, \dots, \pi_{t-1})$.
- learn about optimal policy while following explanatory policy
- simultaneously learn about multiple policies while following one policy. This can be used in multi objective problems when there are several objective function that are in trade-off among them, and you want to learn the performance of different trade offs by using a single exploratory policy.

How can we estimate a policy from another one? We can use the concept of **importance sampling**. This (statistic) method is used in general to estimate the expectation of a different distribution w.r.t. the distribution used to draw samples. Imagine to have a generic function $f(x)$. We sample $f(x)$ using a distribution p ³⁹. To denote that the samples x are drawn from p we use $x \sim p$. The expected value of $f(x)$ can be written as $E_{x \sim p}[f(x)]$. We do not know p , otherwise I can take the samples from p , average them and get the expected

³⁹Note that $f(x)$ is simply a generic function, for which we want to find the expected value. We want to estimate the expected value through sampling. To sample from the input space (x), we must use a probability distribution, in our case p

value of the function f . Our objective is to find $E_{x \sim p}[f(x)]$ by sampling x with another distribution q . We can manipulate the expected value to achieve this.

$$\begin{aligned} E_{x \sim p}[f(x)] &= \int p(x)f(x)dx \\ &= \int q(x)\left(\frac{p(x)}{q(x)}f(x)\right)dx \\ &= E_{x \sim q}\left[\frac{p(x)}{q(x)}f(x)\right] \end{aligned}$$

We can see in the second line of the procedure, that we can consider as a new generic function $\frac{p(x)}{q(x)}f(x)$. Seeing that, we have an integral of a function, multiplied by a distribution q . This is by definition an expected value, with the samples drawn from q . In this way we can estimate the expected value of $f(x)$, by sampling from another distribution. The ratio $\frac{p(x)}{q(x)}$ is also called **importance weight** $W(x)$. We can guess that this **ratio weights the importance of the given sample**. Considering that the samples are taken from q , When p gives a big importance to a region, and q doesn't, we must amplify the sample. In p we sample frequently a given region. In q we sample rarely, but when applying importance sampling every sample from that region have a large weight to compensate. In fact when $p(x)$ is large and $q(x)$ is small $W(x)$ becomes bigger. Importance sampling is like doing a weighted average, where the samples are weighted according to their importance. In this way we can have an unbiased estimate of what I want. So by simply computing this ratio the estimate of the expected value of a distribution, sampling values from another distribution, is unbiased.

In our RL problem, we can use it at our advantage. If we consider as our generic function the return v_t , and as sampling distributions the target policy π and behavior policy $\bar{\pi}$, we can apply importance sampling⁴⁰. Assume to use a MC approach, so we estimate the return with the whole trajectory. Using $\bar{\pi}$, we obtain at the end of an episode the return v_t , Then, we weight using importance sampling the return obtained obtaining v_t^μ .

$$v_t^\mu = \frac{\pi(a_t|s_t)\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_t|s_t)\bar{\pi}(a_{t+1}|s_{t+1})} \dots \frac{\pi(a_T|s_T)}{\bar{\pi}(a_T|s_T)} v_t \quad (99)$$

Once we obtained the re-weighted return, we can update the action-value function using v_t^μ

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(v_t^\mu - Q(s_t, a_t)) \quad (100)$$

The behavior policy must be chosen very wisely. It has to be very close to the target policy, otherwise the estimation variance grows dramatically. In particular we must use a $\bar{\pi}$ that is zero where π is zero.

We can also use importance sampling with SARSA. In this case we get much better results compared to MC. We use TD targets generated from $\bar{\pi}$ to evaluate π . In practice with importance sampling, we weight the TD target $r + \gamma Q(s', a')$ according to the similarity

⁴⁰ $f(x) \rightarrow v_t, p(x) \rightarrow \pi, q(x) \rightarrow \bar{\pi}$

between the two policies. As we have seen before, with SARSA we perform a step and then we evaluate the action-value function. Doing so, we only need a single importance sampling correction at each step

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \frac{\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_{t+1}|s_{t+1})} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \quad (101)$$

With SARSA we have much lower variance than Monte–Carlo importance sampling and policies only need to be similar over a single step.

Off-Policy Control with Q-learning Q-learning is the most used RL algorithm. It's very similar to SARSA,

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in A} \{Q(s', a')\} - Q(s, a)) \quad (102)$$

We can see how the only difference is the usage of the max operator in the TD target. This little difference makes a lot of difference. First of all, the algorithm becomes off-policy, because the next action a' in the TD target is decided using the greedy policy in the current state, and not using the policy π . Most importantly, we can demonstrate that Q-learning converges to the optimal Q^* even when the GLIE condition is not respected. This means that we don't have to stop exploring in order to converge to the optimal solution. The algorithm is so solid that, even if we use always the random policy the algorithm will converge to the optimal solution. As SARSA can be seen as the model-free version of policy iteration, Q-learning can be seen as the model-free version of value iteration. In fact, we can see how Q-learning uses the optimal Bellman operator over the single step. Note that, to choose the actual action a in a given state we use the policy π derived from the ϵ -greedy improvement. To estimate the next action a' we use the greedy policy of π .

Algorithm 7: Q-learning

```

Output :  $Q^*$ 
Input   :  $S, A, \pi_0, \gamma, \epsilon$ 
Initialize:  $Q(s,a)$  arbitrarily
               $\pi \leftarrow \pi_0$ 
do
     $s \leftarrow$  starting state
    repeat
         $a \leftarrow$  action given by  $\pi$  for  $s$ 
        Take action  $a$ , observe  $r$ , and next state  $s'$ 
         $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in A} \{Q(s', a')\} - Q(s, a))$ 
         $s \leftarrow s'$ 
         $\pi \leftarrow \epsilon\text{-greedy}(\pi)$ 
    until  $s$  is terminal;
    while convergence;

```

11 Multi Armed Bandit

Multi Armed Bandit(MAB) is a classic reinforcement learning problem that exemplifies the exploration-exploitation tradeoff dilemma. The name comes from imagining a gambler at a row of slot machines (sometimes known as "one-armed bandits"), who has to decide which machines to play, how many times to play each machine and in which order to play them, and whether to continue with the current machine or try a different machine. The multi-armed bandit problem also falls into the broad category of stochastic scheduling. Let's make an example to introduce the topic.

Example - Beer selection problem Suppose to enter in a newly opened brewery. We are allowed to choose among a set of available beers. The brewery have a rule. You can order a new beer only if you finished the previous one. After each beer, you assign a mark from 1 to 10 according to how much you liked it. It might happen that the value you assign to a beer varies. For example, one beer was expired or it had a production defect. We have two goals

- Find the beer you like the most
- Don't get wasted tasting beers

We can transpose this problem into an MDP. We can describe the value of each action(drinking a beer) in every state with a action-value function $Q(s, a)$. When we enter the brewery we don't know the Q function. We have to learn it **online**⁴¹. Every online decision problem make us face a fundamental choice

- **Exploration** gather more information from unexplored/less explored options. Choose a random beer to try something new. The new beer can be a revelation or be disgusting
- **Exploitation** select the option we consider to be the best one so far. Choose a type of beer we are sure we like

This balance is influenced by many factor. One of these is the time-horizon of our problem. Depending on how much we are far-sighted we might make some sacrifice in the short-term to gain more in the future. With **infinite time horizon** we have infinite steps(we are going several time to the brewery). We want to gather enough information to find the best overall decision. With **finite time horizon** we have a limited number of steps(We only go one time to the brewery). We want to minimize the short-term loss due to uncertainty. Usually MAB works with finite time horizon problems.

⁴¹Online means that we estimate the Q function interleaved with data gathering. In our case after every beer we try to estimate the action-value function

Example - Dilemma on real application To better visualize the exploration-exploitation dilemma, we can make some more examples.

- Clinical Trial
 - Exploration: Try new treatments
 - Exploitation: Choose the treatment that provides the best results
- Slot machine (a.k.a. one-armed bandit) selection
 - Exploration: Try all the available slot machines
 - Exploitation: Pull the one which provided you the highest payoff so far
- Game Playing
 - Exploration: Play an unexpected move
 - Exploitation: Play the move you think is the best
- Oil Drilling
 - Exploration: Drill at an unexplored location
 - Exploitation: Drill at the best known location

11.1 MAB problem

There are different types of MAB. They are fully characterized by the type of **reward**

- **Deterministic** we have a single value for the reward for each arm(We get the same reward from the same type of beer)
- **Stochastic** the reward of an arm is drawn from a distribution which is stationary over time(We get different rewards from the same type of beer)
- **Adversarial** an adversary chooses the reward we get from an arm at a specific round, knowing the algorithm we are using to solve the problem

We have seen in the previous chapters, how we can use MDPs to model a problem. It would be nice if we could map a MAB problem to a MDP. It turns out that we can see the Multi-Armed Bandit setting as a specific case of an MDP where

- S set of states. We have a single state, $S = \{s\}$
- A set of actions. Also called arms, $A = \{a_1, \dots, a_N\}$
- P state transition probability matrix. $P(s|a_i, s) = 1, \forall a_i$
- R reward function. $R(s, a_i) = R(a_i)$

- γ discount factor. We have a finite time horizon, $\gamma = 1$
- μ^0 set of initial probabilities. $\mu^0(s) = 1$

In our case we have all the ingredients to build a MDP representation, except the reward function R . We have already seen how we can still solve the problem using RL using model-free control algorithms. Now let's focus on finding a policy(policy improvement) for our problem. We have seen several methods, for example ϵ -greedy.

$$\pi(a_i|s) = \begin{cases} 1 - \epsilon & \text{if } \hat{Q}(a_i|s) = \max_{a \in A} \hat{Q}(s|s) \\ \frac{\epsilon}{|A|-1} & \text{otherwise} \end{cases} \quad (103)$$

We perform the greedy action with probability $1 - \epsilon$, and a random action between the others with probability $\frac{\epsilon}{|A|-1}$. This approach converges to the optimal policy, only if ϵ converges to zero over time.

Another approach is the **softmax**

$$\pi(a_i|s) = \frac{e^{\frac{\hat{Q}(a_i|s)}{\tau}}}{\sum_{a \in A} e^{\frac{\hat{Q}(a|s)}{\tau}}} \quad (104)$$

Weights the actions according to its estimated value $\hat{Q}(a|s)$. τ is a parameter that decreases over time. Even if these algorithms converge to the optimal choice, we do not know how much we lose during the learning process.

11.1.1 Stochastic MAB

The definition of MAB problems with MDP is useful but is not the only one. A Multi-armed Bandit problem can be seen as a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$

- \mathcal{A} is a set of N possible arms(choices)
- \mathcal{R} is an set of unknown random variable $\mathcal{R}(a_i)$, where $E[\mathcal{R}(a_i)] = R(a_i)$

The process we consider is the following. At each round t the agent selects a single arm a_{i_t} . Then the environment generates a reward r_{i_t} drawn from $\mathcal{R}(a_{i_t})$. Finally the agent updates its information by means of a history h_t (pulled arm and received reward).

The final objective of the agent is to maximize the cumulative reward over a given time horizon T

$$\sum_{t=1}^T r_{i_t, t}$$

where $r_{i_t, t}$ is the realization of the reward for the arm a_{i_t} we choose for the turn. Possibly we also want to converge to the option with largest expected reward if one considers $T \rightarrow \infty$. We can redefine our objective as to minimize the reward we lost through acting non-optimally.

The objective function can be reformulated in the following way. Define the expected reward of the optimal arm a^* as

$$R^* = R(a^*) = \max_{a \in A} E[\mathcal{R}(a)]$$

At a given time step t , we select the action a_{i_t} , and we incur in a loss of $\mathcal{R}(a^*) - \mathcal{R}(a_{i_t})$. The reward is stochastic, so it can be useful to calculate the average loss of the algorithm

$$E[\mathcal{R}(a^*) - \mathcal{R}(a_{i_t})] = R^* - R(a_{i_t})$$

This difference can be called **regret**. We want to minimize the expected regret suffered over a finite time horizon of T rounds

Definition 11.1 (Expected pseudo regret).

$$L_T = TR^* - E\left[\sum_{t=1}^T R(a_{i_t})\right]$$

The expected value is taken w.r.t. the stochasticity of the reward function and the randomness of the used algorithm. Note that the maximization of the cumulative reward is equivalent to the minimization of the cumulative regret.

We can reformulate the cumulative regret in another way. We define the average difference in reward between a generic arm a_i and the optimal one a^* as $\Delta_i := R^* - R(a_i)$. Then we define the number of times an arm a_i has been pulled after a total of t time steps as $N_t(a_i)$.

$$\begin{aligned} L_T &= TR^* - E\left[\sum_{t=1}^T R(a_{i_t})\right] \\ &= E\left[\sum_{t=1}^T R^* - R(a_{i_t})\right] \\ &= \sum_{a \in A} E[N_t(a_i)](R^* - R(a_{i_t})) \\ &= \sum_{a \in A} E[N_t(a_i)]\Delta_i \end{aligned}$$

Note how we rewrote the summation over time to a summation over arms. At each time step we choose an arm. If we choose the arm a_i $N_t(a_i)$ times we have that $T = N_t(a_1) + \dots + N_t(a_N)$. Looking at formula we can see that to minimize the regret we want to minimize the number of times we select a sub-optimal arm.

We can find a lower bound of the regret

Theorem 11.1. *Given a MAB stochastic problem, any algorithm satisfies*

$$\lim_{T \rightarrow \infty} L_T \geq \log(T) \sum_{a_i | \Delta_i > 0} \frac{\Delta_i}{KL(R(a_i), R(a^*))} \quad (105)$$

where $KL(R(a_i), R(a^*))$ is the Kullback-Leibler divergence between the two Bernoulli distributions $\mathcal{R}(a_i)$ and $\mathcal{R}(a^*)$. The important thing to notice is that the cumulative regret depends on the time horizon T . So we can't have a constant regret over all time steps. We can show that the Kullback-Leibler divergence is proportional to Δ^2 . This means that the argument of the summation on the right hand side is proportional to $\frac{1}{\Delta_i}$. If we have a small Δ_i we have a higher lower bound and vice versa. This can explained by the fact that, if all the arms are very close to the optimal action, it will be difficult to choose which of the arm is the best.

Now let's discuss some algorithm which can came close to this lower bound. The simplest algorithm we can think of always select the action such that $a_{i_t} = \underset{a}{\operatorname{argmax}} \hat{R}_t(a)$ where the expected reward for an arm is

$$\hat{R}_t(a_i) = \frac{1}{N_t(a_i)} \sum_{j=1}^t r_{i,j} \mathbf{1}(a_i = a_{i_j})$$

This algorithm is called **pure exploitation algorithm**. It might not converge to the optimal action. We can construct a counter example. Suppose to have Bernoulli returns. At the beginning, we choose the optimal arm, and it returns zero(note that the rewards are stochastic). If we get other samples, and we never choose the optimal action again, and one of the non-optimal function gets a reward of one, the optimal arm will never be chosen. To adjust the algorithm we need to consider the uncertainty corresponding to the $\hat{R}_t(a)$ estimate. One way to do that is by providing an explicit bonus for exploration.

There are two different formulations

- **Frequentist formulation** $R(a_1), \dots, R(a_N)$ are considered as unknown parameters. A policy selects at each time step an arm based on the observation history
- **Bayesian formulation** $R(a_1), \dots, R(a_N)$ are considered as random variables with prior distributions f_1, \dots, f_N . A policy selects at each time step an arm based on the observation history and on the provided priors

Frequentist algorithms Let's start with the frequentist approach. The main take away of this approach is that the more we are uncertain on a specific choice, the more we want the algorithm to explore that option. Doing so, we might lose some value in the current round, but it might turn out that the explored action is the best one. An example is the upper confidence bound approach. Instead of using the empiric estimate we consider an upper bound $U(a_i)$ over the expected value $R(a_i)$. More formally, we need to compute an upper bound

$$U(a_i) := \hat{R}_t(a_i) + B_t(a_i) \geq R(a_i)$$

with high probability. The bound length $B_t(a_i)$ depends on how much information we have on an arm, for example the number of times we pulled that arm so far $N_t(a_i)$. Small $N_t(a_i) \rightarrow$ large $U(a_i)$ (the estimated value $\hat{R}_t(a_i)$ is uncertain). Large $N_t(a_i) \rightarrow$ small $U(a_i)$

(the estimated value $\hat{R}_t(a_i)$ is accurate). In order to set the upper bound we resort to a classical concentration inequality

Definition 11.2 (Hoeffding Bound). *Let X_1, \dots, X_t be i.i.d. random variables with support in $[0, 1]$ and identical mean $E[X_i] =: X$ and let $\bar{X}_t = \frac{\sum_{i=1}^t X_i}{t}$ be the sample mean. Then*

$$P(X > \bar{X}_t + u) \leq e^{-2tu^2}$$

We will apply this inequality to the upper bounds corresponding to each arm

$$P(R(a_i) > \hat{R}_t(a_i) + B_t(a_i)) \leq e^{-2N_t(a_i)B_t(a_i)^2} \quad (106)$$

We want to find B_t . To do so, we fix the probability of the bound

$$e^{-2N_t(a_i)B_t(a_i)^2} = p$$

We solve to find $B_t(a_i)$

$$B_t(a_i) = \sqrt{\frac{-\log(p)}{2N_t(a_i)}}$$

the probability p represents the probability of the expect value overcome the bound. If we perform many steps these probabilities sums up and almost surely the bound will be broken. To solve this problem, we can shrink p over time. For example the probability can be equal to $p = t^{-4}$

$$B_t(a_i) = \sqrt{\frac{2\log(t)}{N_t(a_i)}} \quad (107)$$

This ensures to select the optimal action as the number of samples increases. $\lim_{t \rightarrow \infty} B_t(a_i) = 0 \Rightarrow \lim_{t \rightarrow \infty} U_t(a_i) = R(a_i)$

This algorithm is called UCB1. For each time step t we do three steps.

1. Compute

$$\hat{R}_t(a_i) = \frac{\sum_{i=1}^t r_{i,t} \mathbf{1}(a_i = a_{i_t})}{N_t}, \quad \forall a_i$$

2. Compute

$$B_t(a_i) = \sqrt{\frac{2\log(t)}{N_t(a_i)}}, \quad \forall a_i$$

3. Play arm

$$a_{i_t} = \operatorname{argmax}_{a_i \in A} \left(\hat{R}_t(a_i) + B_t(a_i) \right)$$

From that we can derive the specific upper bound of UCB1

Theorem 11.2 (UCB1 Upper Bound, Auer & Cesa-Bianchi 2002). *At finite time T , the expected total regret of the UCB1 algorithm applied to a stochastic MAB problem is*

$$L_T \leq 8\log(T) \sum_{i|\Delta_i>0} \frac{1}{\Delta_i} + \left(1 + \frac{\pi^2}{3}\right) \sum_{i|\Delta_i>0} \Delta_i$$

Remember that the lower bound we have found before is general for all the algorithms. Instead, for the upper bound, every algorithm has a different one. For UCB we have that the two bounds depends on the same order, $\log(T)$. This is very nice, the upper bound is good.

Bayesian algorithms The first Bayesian algorithm we see is **Thompson sampling**. It is a general Bayesian methodology for online learning. It is structured as follow. Consider a Bayesian prior for each arm f_1, \dots, f_N as a starting point. At each round t sample from each one of the distributions $\hat{r}_1, \dots, \hat{r}_N$. Pull the arm a_{i_t} with the highest sampled value $i_t = \underset{i}{\operatorname{argmax}}\{\hat{r}_i\}$. Update the prior of the pulled arm, incorporating the new information.

Let's see in more detail this algorithm. The prior conjugate distributions are $Beta(\alpha, \beta)$. In particular we take $f_i(0) = Beta(1, 1)$ for each arm a_i . After we pulled the arm a_i , we get a Bernoulli reward. Then, we update the prior incorporating information gathered. After the update we still have a $Beta$ distribution. In detail we have,

- In the case of a success occurs $f_i(t+1) = Beta(\alpha_t + 1; \beta_t)$
- In the case of a failure occurs $f_i(t+1) = Beta(\alpha_t; \beta_t + 1)$

The upper bound of the algorithm is

Theorem 11.3 (Thompson Sampling Upper Bound, Kaufmann & Munos 2012). *At time T , the expected total regret of Thompson Sampling algorithm applied to a stochastic MAB problem is*

$$L_T \leq O\left(\sum_{i|\Delta_i>0} \frac{\Delta_i}{KL(\mathcal{R}(a_i), \mathcal{R}(a^*))} (\log(T) + \log(\log(T)))\right)$$

Note that this upper bound have the same order and constant of the lower bound. This means that we can't build a better bound.

11.1.2 Adversarial MAB

With adversarial MAB the reward is no longer given by the environment, but is returned by an adversary player. A Multi-Armed Bandit Adversary setting is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$

- \mathcal{A} is a set of N possible arms (choices)
- \mathcal{R} is a reward vector for which the realization $r_{i,t}$ is decided by an adversary player at each turn

The process we consider is the following

- At each time step t the agent selects a single arm a_{i_t}
- At the same time the adversary chooses rewards $r_{i,t}$
- The agent gets reward $r_{i,t}$
- The final objective of the agent is to maximize the cumulative reward over a time horizon T

$$\sum_{t=1}^T r_{i_t, t}$$

We cannot compare the accumulated regret we gained with the optimal one. Moreover, the fact that there is an adversary choosing the regret does not allow to use deterministic algorithms (e.g., UCB). This is due to the fact that, the return is no longer characterized by a random variable, and so it's not stochastic. Instead, we can use the **weak regret**.

Definition 11.3 (Weak regret).

$$L_T = \max_i \left(\sum_{t=1}^T r_{i,t} \right) - \sum_{t=1}^T r_{i_t, t}$$

We are comparing the policy with the best constant action. As we did before, we can construct a lower bound for all the algorithms

Theorem 11.4 (Minimax Lower Bound). *Let \sup be the supremum over all distribution of rewards such that, for $i \in \{1, \dots, N\}$ the rewards $r_{i,1}, \dots, r_{i,T}$ and $r_{i,j}$ are i.i.d. and let \inf be the infimum over all forecasters. Then*

$$\inf \sup E[L_T] \geq \frac{1}{20} \sqrt{TN}$$

where the expectation is taken with respect to both the random generation rewards and the internal randomization of the forecaster

The first non-deterministic algorithm we see is **EXP3**. This algorithm derives from the softmax algorithm. The probability of choosing an arm (policy) is

$$\pi_t(a_i) = (1 - \beta) \frac{w_t(a_i)}{\sum_j w_t(a_j)} + \frac{\beta}{N} \quad (108)$$

where w is a weight calculated as

$$w_{t+1}(a_i) = \begin{cases} w_t(a_i) e^{\eta \frac{r_{i,t}}{\pi_t(a_i)}} & \text{if } a_i \text{ has been pulled} \\ w_t(a_i) & \text{otherwise} \end{cases}$$

and β is a constant term.

For this algorithm, the upper bound is

Theorem 11.5 (EXP3 Upper Bound). *At time T , the expected weak regret of EXP3 algorithm applied to an adversarial MAB problem with $\beta = \eta = \sqrt{\frac{N \log(N)}{(e-1)T}}$ is*

$$E(L_T) \leq O(\sqrt{TN \log(N)})$$

where the expectation is taken with respect to both the random generation rewards and the internal randomization of the forecaster

11.2 Generalized MAB problem

In the beer selection problem you now have a set of breweries. Each night your friend decides which one to pick. Once you are in a specific brewery you are free to pick a beer of your choice. Is this a Bandit problem? If we fix the brewery, it is a stochastic MAB. Since we do not control the transition from one brewery to another, we can use independent MAB techniques over each one of the breweries. This change in environment(different breweries) needs a new type of MAB called **contextual MAB**. There exist other type of MAB like

- **Budget-MAB** we are allowed to pull arms until a fixed budget elapses, where the pulling action incurs in a reward and a cost
- **Continuous Armed Bandit** we have a set of arms A which is not finite

We can also have other sequential decision settings

- **Arm identification problem** we just want to identify the optimal arm with a given confidence, without caring about the regret
- **Expert setting** we are also allowed to see the rewards of the not pulled arms each turn (online learning problem)