Politecnico di Milano

Machine Learning

AA 2019/2020

Summary - Machine Learning

Author:
Valerio Colombo

June 25, 2024

Contents

1 Linear models for regression

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 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 nonlinear 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 nonlinear 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 nonlinear models in input/output space.

1.1 Linear Basis Function Models

1.1.1 Linear regression

The simplest linear model for regression is one that involves a linear combination of the input variables. Given a set comprising M-1 observations x_m, where $m = 1, ..., M-1$ we have,

$$y(x, w) = w_0 + w_1 x_1 + ... + w_{M-1} x_{M-1} = w_0 + \sum_{n=1}^{M-1} (w_n x_n) = w^T x,$$

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

This is often simply known as linear regression. The key property of this model is that it is a linear function of the parameters $w_0, ..., w_{M-1}$. It is also, however, a linear function of the input variables x_i, and this imposes significant limitations on the model.

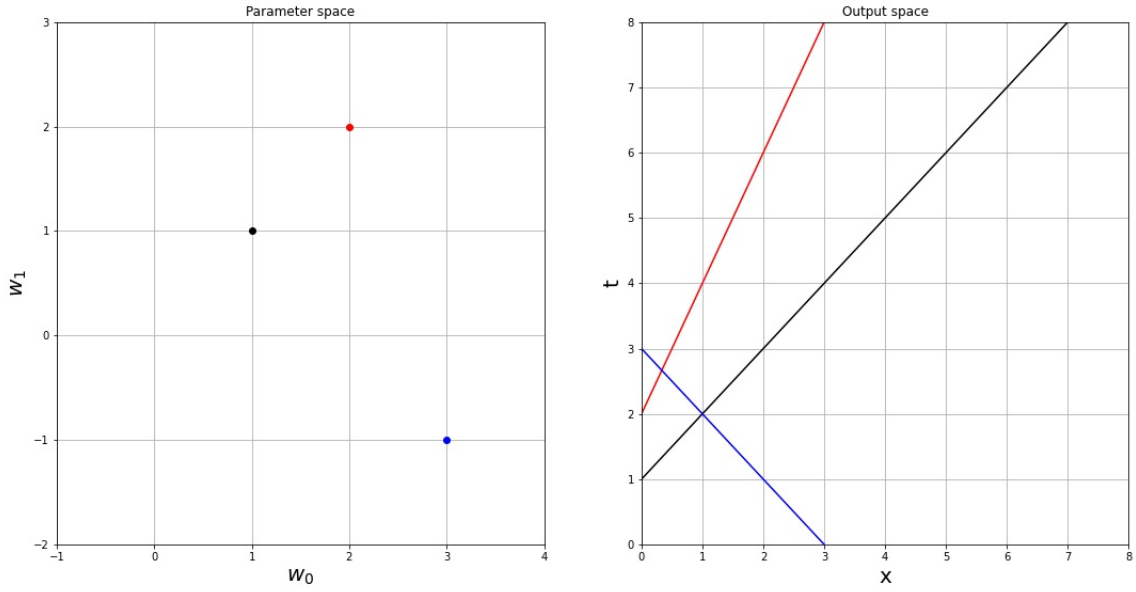
Example We want to estimate the weight of a person based on his age, height and gender So our inputs variables would be

$$\begin{aligned} &age, && x_a \in \mathbb{N} \\ &height, && x_h \in \mathbb{N} \\ &gender, && x_g \in \mathbb{N} \\ &bias, && 1 \end{aligned}$$

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

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

Note In linear regression we can distinguish two spaces. Hypothesis space(Parameters space) and Output space(Input space)



1.1.2 Linear regression with non-linear basis functions

We can extend the class of models by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(x, w) = w_0 + \sum_{j=1}^{M-1} (w_j \Phi_j(x)) = w^T \Phi(x),$$
$$where \quad \Phi(x) = \begin{bmatrix} 1 \\ \Phi_1(x) \\ \vdots \\ \Phi_{M-1}(x) \end{bmatrix} \tag{2}$$

By using nonlinear basis functions[2], we allow the function $y(x, w)$ to be a nonlinear 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.

[2]Non-linear functions like: $e^x, sin(x), x^2$,...

1.1.3 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. To do so we use loss functions.

$L(t, y(x))$	Loss function
$E[L] = \int \int L(t, y(x))p(x, t)dxdt$	Average loss function

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

$$E[L] = \int \int (t - y(x))^q p(t, x)dtdx \tag{3}$$

Based on q the model that minimize $E[L]$ is

- $q = 2 : \quad y(x) = E[t|x]$ Conditional mean. Note $E[t|x] = \int tp(t|x)dt$
- $q = 1 : \quad y(x) = median(t|x)$ Conditional median
- $q \to 0 : \quad y(x) = mode(t|x)$ Conditional mode

Note For $q = 2$ we have the squared loss function. This is the most used one.

1.2 Least square minimization

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. Given N samples, we consider the following loss(error) function

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

This loss function is called SSE(Squared Sum of Errors) or half RSS(Residual Sum of Squares). It can be also written as

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

Note Given $x = \begin{bmatrix} x_1 & x_2 & \dots & x_N \end{bmatrix}$, the $l_2 norm$ of x corresponds to

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

1.2.1 Ordinary Least Squares (Closed Form)

Given N samples and M parameters, we construct $\Phi = \begin{bmatrix}\Phi(x_1) & \dots & \Phi(x_N)\end{bmatrix}^T$ and $t = \begin{bmatrix}t_1 & \dots & t_N\end{bmatrix}^T$.
We can rewrite the SSE in matrix notation

$$L(w) = \frac{1}{2}RSS(w) = \frac{1}{2}(t - \Phi w)^T(t - \Phi w) \quad [1 \times 1]^3 \tag{7}$$

Note $\Phi[N \times M]$, $w[M \times 1]$, $t[N \times 1]$
To minimize L(w) we have to calculate the first and the second derivative. First, we find the first derivative of $L(w)$:

$$\frac{\partial L(w)}{\partial w} = \frac{\partial \left(\frac{1}{2}(t - \Phi w)^T(t - \Phi w)\right)}{\partial w} \tag{8}$$

Expanding the quadratic term:

$$L(w) = \frac{1}{2}\left(t^T t - t^T \Phi w - w^T \Phi^T t + w^T \Phi^T \Phi w\right) \tag{9}$$

Since $t^T \Phi w$ and $w^T \Phi^T t$ are scalars and equal, we can simplify:

$$L(w) = \frac{1}{2}\left(t^T t - 2t^T \Phi w + w^T \Phi^T \Phi w\right) \tag{10}$$

Taking the derivative with respect to w:

$$\frac{\partial L(w)}{\partial w} = \frac{\partial}{\partial w}\left(\frac{1}{2}t^T t - t^T \Phi w + \frac{1}{2}w^T \Phi^T \Phi w\right) \tag{11}$$

- The derivative of $\frac{1}{2}t^T t$ with respect to w is zero since it does not depend on w.
- The derivative of $-t^T \Phi w$ with respect to w is $-\Phi^T t$.
- The derivative of $\frac{1}{2}w^T \Phi^T \Phi w$ with respect to w is $\Phi^T \Phi w$.

Putting these together:

$$\frac{\partial L(w)}{\partial w} = -\Phi^T t + \Phi^T \Phi w \tag{12}$$

Simplified expression:

$$\frac{\partial L(w)}{\partial w} = -\Phi^T (t - \Phi w) \quad [M \times 1] \tag{13}$$

[3]The [ ] next to the equations indicates result dimensions

Note $\frac{\partial L(w)}{\partial w}$ are the directions of every w_i to minimize $L(w)$.

Next, we find the second derivative of $L(w)$:

$$\frac{\partial^2 L(w)}{\partial w \partial w^T} = \frac{\partial}{\partial w}\left(\frac{\partial L(w)}{\partial w}\right) \tag{14}$$

Since:

$$\frac{\partial L(w)}{\partial w} = -\Phi^T(t - \Phi w) \tag{15}$$

Taking the derivative with respect to w:

$$\frac{\partial}{\partial w}\left(-\Phi^T(t - \Phi w)\right) = \Phi^T\Phi \tag{16}$$

Thus, the second derivative (Hessian) is:

$$\frac{\partial^2 L(w)}{\partial w \partial w^T} = \Phi^T\Phi \quad [M \times M] \tag{17}$$

Note $\frac{\partial L(w)}{\partial w \partial w^T}$ is a semi-definite positive matrix[4], so all eigen values≥ 0. It also means that for $\frac{\partial L(w)}{\partial w} = 0$ we find the minimum of $L(w)$.

Now we have to find w imposing the first derivative to zero:

$$\begin{aligned}
\frac{\partial L(w)}{\partial w} &= 0 \\
-\Phi^T(t - \Phi w) &= 0 \\
-\Phi^T t + \Phi^T\Phi w &= 0 \\
\Phi^T\Phi w &= \Phi^T t \\
\hat{w} &= (\Phi^T\Phi)^{-1}\Phi^T t \quad [M \times 1]
\end{aligned} \tag{18}$$

Second derivative can give us some infos about features(basis functions) importance. If we have some eigen values close to zero we could have the following situations

- $N < M$, less samples than dimensions(parameters)
- The feature with null eigen value is linearly dependents from other features

From a computational point of view the matrix invertion is a very complex operation. In fact, the temporal complexity of OLS is

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

[4] $x^T\Phi^T\Phi x \geq 0, \quad \forall x \in \mathbb{R} \setminus \emptyset$

Geometric interpretation We can give a geometric interpretation of the problem. Given $\hat{t} = \begin{bmatrix} y(x_1, w) & \dots & y(x_N, w) \end{bmatrix}^T$ 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 and since $\hat{t}$ minimize $L(w)$ with respect to t, it represents the projection of t in S

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

1.2.2 Maximum Likelihood ML (Closed Form)

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

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

To estimate t we can make the following assumptions

- Approximate $f(x)$ with $y(x, w)$
- Assume $\epsilon \sim \mathcal{N}(0, \sigma^2)$

Given N i.i.d.[5]samples, with input $X = \{x_1, ..., x_N\}$ and outputs $t = \begin{bmatrix} t_1 & \dots & t_N \end{bmatrix}^T$ we can say that

$$t_i \sim \mathcal{N}(t_i | w^T\Phi(x_i), \sigma^2) \tag{19}$$

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

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

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

$$\Downarrow \quad \text{Transition to ln to simplify calculus.}$$
$$\text{Min \& Max remain the same}$$

$$ln(p(t|X, w, \sigma^2)) = ln \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}$$

$$= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}\right)$$

$$= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + ln\left(e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}\right)$$

[5]Independent and Identically Distributed random variables

$$
\begin{aligned}
&= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + \sum_{n=1}^{N} ln\left(e^{-\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2}}\right) \\
&= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + \sum_{n=1}^{N} -\frac{1}{2}\frac{(t_n - w^T\Phi(x_n))^2}{\sigma^2} \\
&= \sum_{n=1}^{N} ln\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2 \\
&= \sum_{n=1}^{N} ln\left((2\pi\sigma^2)^{-\frac{1}{2}}\right) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2 \\
&= -\frac{N}{2}ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\underbrace{\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2}_{RSS} \qquad (20)
\end{aligned}
$$

To find the maximum likelihood, we equal the gradient of $ln(p(t|X, w, \sigma^2)) = 0$

$$
\begin{aligned}
\overset{w}{\nabla} ln(p(t|X, w, \sigma^2)) &= \overset{w}{\nabla}\left(-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - w^T\Phi(x_n))^2\right) \\
&= -\frac{1}{2\sigma^2}\sum_{n=1}^{N} 2(t_n - w^T\Phi(x_n))(-\Phi^T(x_n)) \\
&= -\frac{1}{\sigma^2}\sum_{n=1}^{N} t_n\Phi^T(x_n) - w^T\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n) \\
0 &= \sum_{n=1}^{N} t_n\Phi^T(x_n) - w^T\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n) \\
w^T &= \sum_{n=1}^{N} t_n\Phi^T(x_n)\left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1} \\
w &= \left(\sum_{n=1}^{N} t_n\Phi^T(x_n)\left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1}\right)^T \\
&= \left(\left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1}\right)^T\left(\sum_{n=1}^{N} t_n\Phi^T(x_n)\right)^T \\
&= \left(\sum_{n=1}^{N}\Phi(x_n)\Phi^T(x_n)\right)^{-1}\sum_{n=1}^{N}\Phi(x_n)t_n \qquad having \quad \Phi = \begin{bmatrix} [\Phi^T(x_1)] \\ \vdots \\ [\Phi^T(x_N)] \end{bmatrix} \\
&= (\Phi^T\Phi)^{-1}\Phi^T t \quad [M \times 1] \qquad (21)
\end{aligned}
$$

As we can see both OLS and ML give the same result (21).

W variance estimation As we said before for ML calculation we made some assumptions. The most important for calculating the variance are:

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

Given that the variance-covariance matrix of least-squares estimates is

$$Var(\hat{w}_{ML}) = (\Phi^T\Phi)^{-1}\sigma^2 \tag{22}$$

As we have seen before $\Phi^T\Phi$ matrix can give us some insights on features importance. If a features is relevant its eigen value 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 eigen values reduce variance. We can achieve variance reduction by gathering more samples for our estimation.

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

$$\hat{W}_{ML} = \underbrace{(\Phi^T\Phi)^{-1}\Phi^T}_{\Phi^\dagger \text{Pseudo-inverse}} T \quad [M \times K], \quad where \quad T \quad [N \times K] \tag{23}$$

1.2.3 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 stochastic[6] gradient 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.

$$w^{\overset{\text{iteration}}{(k+1)}} = w^{(k)} - \underset{\text{learning rate}}{\alpha^{(k)}} \nabla L(x_n) \tag{24}$$

The learning rate α is an important hyperparameter[7]of 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

[6]Stochastic 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.

[7]In 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.

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

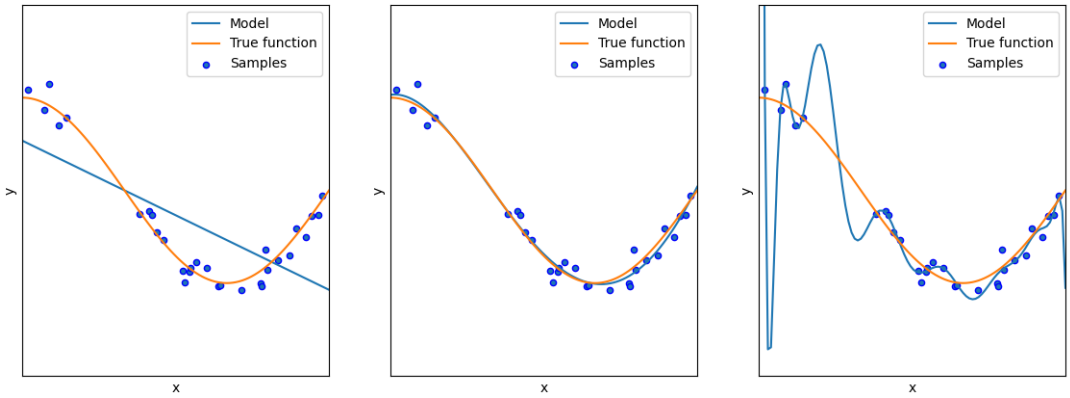
$$\sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)}} = \infty \quad \wedge \quad \sum_{k=0}^{\infty} \frac{1}{\alpha^{(k)^2}} = M, \quad M \in \mathbb{R} \tag{25}$$

1.2.4 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.

- **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.

A naive way to measure the performance of our model is to look at the 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. In fact, 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.



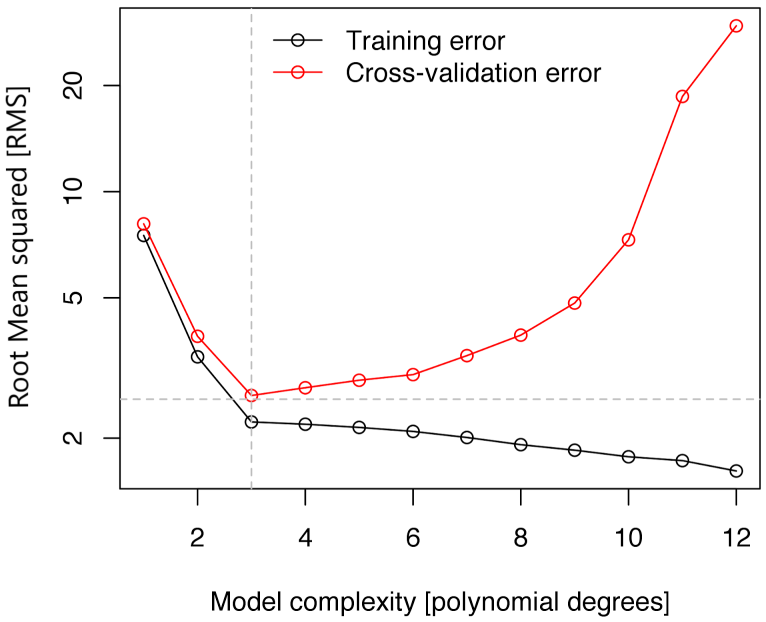
To overcome this problem we can split the dataset in two subset,

- **Training set** This set is used to learn the model's parameters.
- **Test set** This 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

$$E_{RMS} = \sqrt{\frac{2RSS(\hat{w})}{N}} \tag{26}$$

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.



Another indicator for poor generalization is parameters absolute value. If we have very large parameters it means that the model oscillate very rapidly. This is an evidence of overfitting.

1.3 Regularization

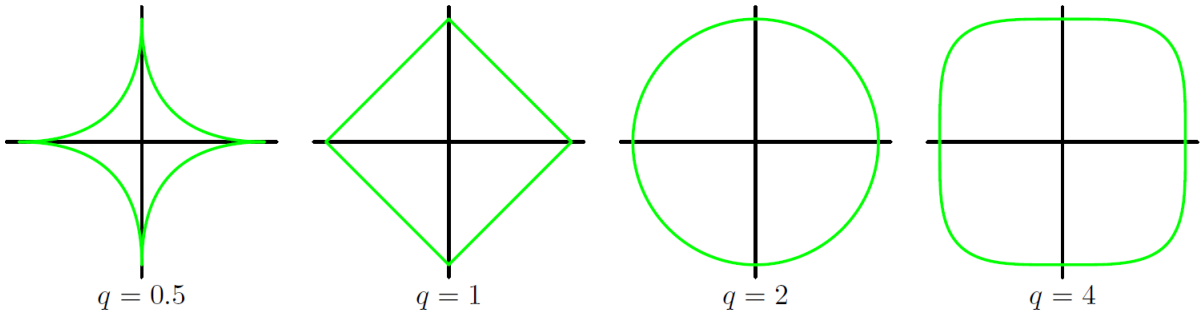
One of the major aspects of training your machine learning model is avoiding overfitting. The model will have a low accuracy if it is overfitting. This happens because your model is trying too hard to capture the noise in your training dataset. One way to avoid overfitting is using cross validation.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

$$L(w) = \underbrace{L_D(w)}_{\text{error on data}} + \underbrace{\lambda L_W(w)}_{\text{error on complexity}} \tag{27}$$

For tractability we take

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

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.

1.3.1 Ridge regression

For $q = 2$ we have the so called Ridge regression.

$$\begin{aligned} L_W(w) &= \frac{1}{2} w^T w = \frac{1}{2} \|w\|_2^2 \\ L(w) &= \frac{1}{2} \sum_{j=1}^{N} (t_i - w^T \Phi(x_i))^2 + \frac{\lambda}{2} w^T w \\ &= \frac{1}{2} (t - \Phi w)^T (t - \Phi w) + \frac{\lambda}{2} w^T w \end{aligned} \tag{28}$$

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.

$$\hat{w}_{ridge} = (\lambda I + \Phi^T \Phi)^{-1} \Phi^T t \tag{29}$$

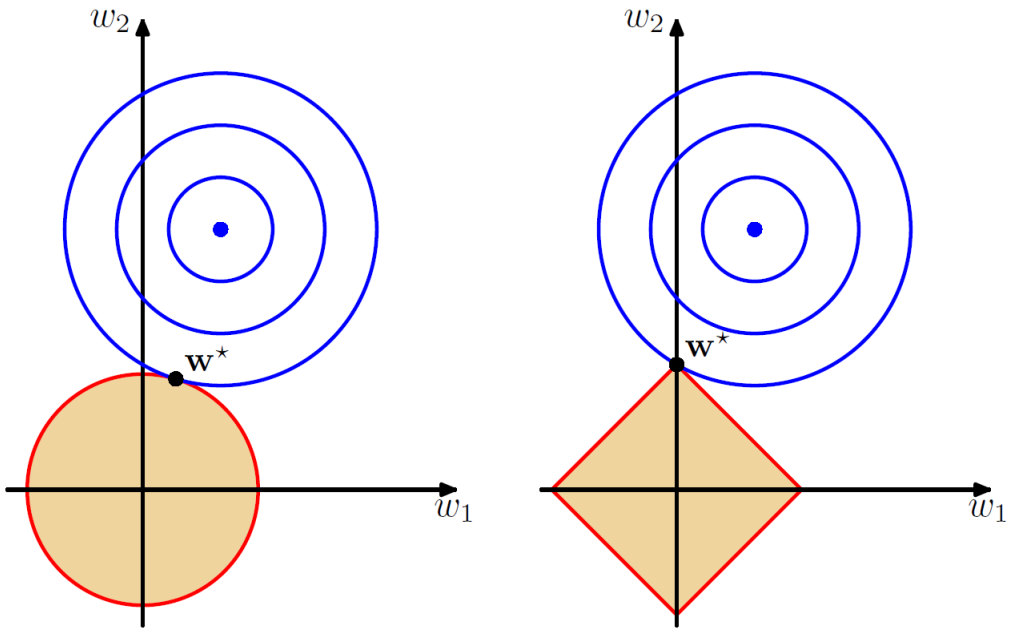
Note The eigen values of $(\lambda I + \Phi^T \Phi)$ are still greater or equal to zero because $\Phi^T \Phi$ is semi-definite positive and λI simply imposes a positive lower bound to the eigen values. $(\lambda I + \Phi^T \Phi)$ is still semi-definite positive and so invertible.

1.3.2 Lasso

For $q = 1$ we have the so called Lasso.

$$
\begin{aligned}
L_W(w) &= \frac{1}{2}\sum_{j=1}^{M}|w_j| = \frac{1}{2}\|w\|_1 \\
L(w) &= \frac{1}{2}\sum_{j=1}^{N}(t_i - w^T\Phi(x_i))^2 + \frac{\lambda}{2}\sum_{j=1}^{M}|w_j| \\
&= \frac{1}{2}\sum_{j=1}^{N}(t_i - w^T\Phi(x_i))^2 + \frac{\lambda}{2}\|w\|_1
\end{aligned} \tag{30}
$$

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[8].



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.

[8] A model is sparse if some parameters tend to zero, eliminating some features from the model

1.4 Bayesian Linear regression

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 hearth of this type of regression. As a reminder, Bayes theorem states

$$\underbrace{P(A|B)}_{Posterior} = \frac{\overbrace{P(B|A)}^{Likelihood}\overbrace{P(A)}^{Prior}}{\underbrace{P(B)}_{Marginalization}} \tag{31}$$

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 $P(r) = p = \frac{1}{2}$ ($P(Head) = \frac{1}{2}$ and $P(Tail) = \frac{1}{2}$)
- **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.

$$P(w) \sim \mathcal{N}(w|w_0, S_0) \tag{32}$$

- w_0 $[M \times 1]$ Mean of the distribution. We guess that w is equal to w_0 in the parameter space.
- S_0 $[M \times M]$ Covariance matrix of the distribution. The matrix is diagonal because we assume i.i.d. parameters.

So P(w) is a multivariate[9]Gaussian. As we have said before, the posterior will be Gaussian

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

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

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

[9]In 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,

$$\begin{aligned} S_N^{-1} &= S_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2} = \frac{\Phi^T\Phi}{\sigma^2} \\ S_N &= \sigma^2(\Phi^T\Phi)^{-1} \\ w_N &= S_N\left(S_0^{-1}w_0 + \frac{\Phi^T t}{\sigma^2}\right) \\ &= \sigma^2(\Phi^T\Phi)^{-1}\frac{\Phi^T t}{\sigma^2} \\ &= (\Phi^T\Phi)^{-1}\Phi^T t \end{aligned} \tag{33}$$

  As we can see (33) is equal to the ML estimator. So Bayesian regression reduces to the maximum likelihood case.

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

$$\begin{aligned} S_N^{-1} &= S_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2} \\ &= \frac{1}{\tau^2}I + \frac{\Phi^T\Phi}{\sigma^2} \\ w_N &= S_N\left(S_0^{-1}w_0 + \frac{\Phi^T t}{\sigma^2}\right) \\ &= S_N\frac{\Phi^T t}{\sigma^2} \\ &= \left(\frac{1}{\tau^2}I + \frac{\Phi^T\Phi}{\sigma^2}\right)^{-1}\frac{\Phi^T t}{\sigma^2} \\ &= \left(\frac{\sigma^2}{\tau^2}I + \Phi^T\Phi\right)^{-1}\Phi^T t \end{aligned} \tag{34}$$

  We can notice that (34) 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.

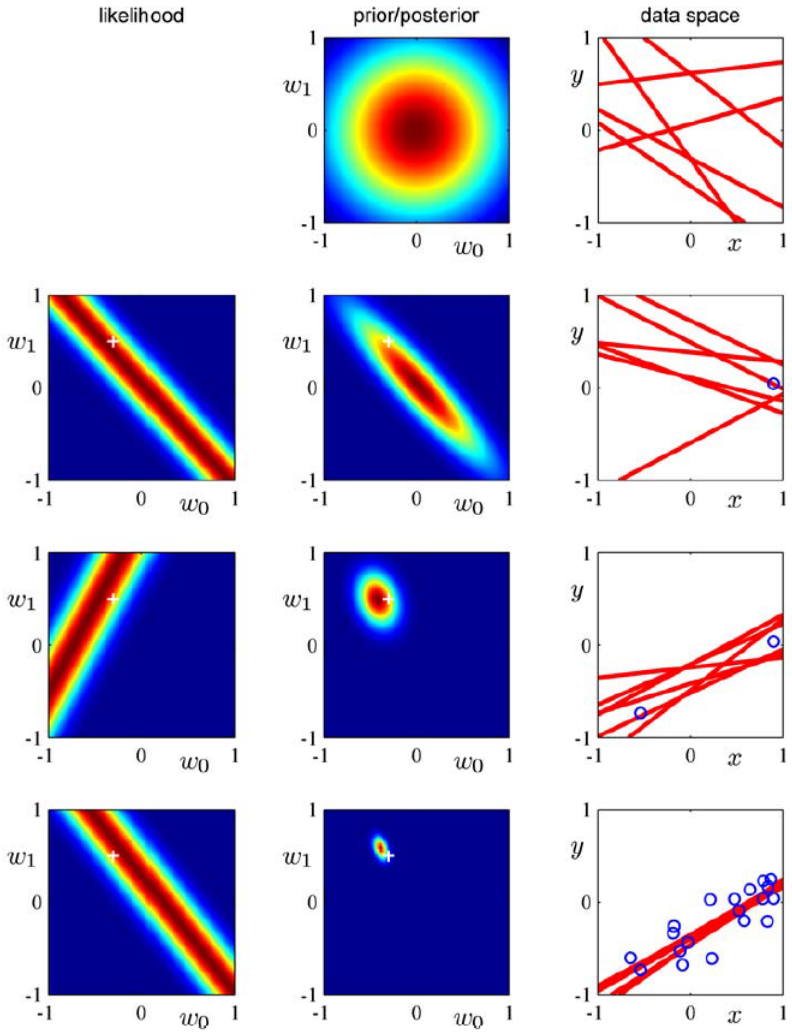
Example - Bayesian sequential learning We generate some data from

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

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.



1.4.1 Predictive distribution

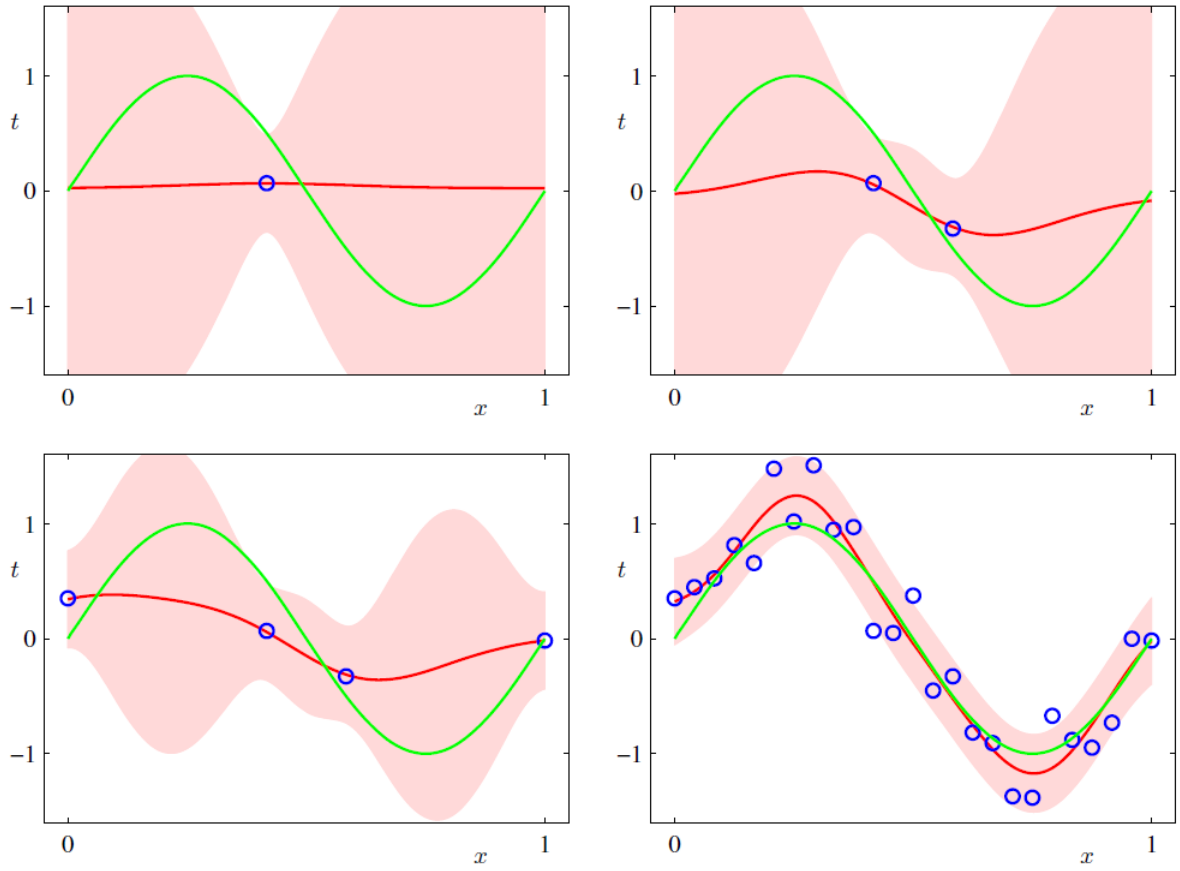
In practice, we are not usually interested in the value of w itself but rather in making predictions of t for new values of x. This requires that we evaluate the posterior predictive distribution defined by,

$$p(t|x, D, \sigma^2) = \int \mathcal{N}(t|w^T\Phi(x), \sigma^2)\mathcal{N}(w|w_N, S_N)dw = \mathcal{N}(t|w_N^T\Phi(x), \sigma_N^2(x)), \tag{35}$$

$$\sigma_N^2(x) = \underbrace{\sigma^2}_{\substack{\text{noise in the} \\ \text{target values}}} + \underbrace{\Phi(x)^T S_N \Phi(x)}_{\substack{\text{Uncertainty associated} \\ \text{with parameters values}}} \tag{36}$$

σ^2 is also called irreducible noise, in fact for $N \to \infty$, 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.



2 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 $k = 1, ..., K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. The input space is thereby divided into decision regions whose boundaries are called decision boundaries or decision surfaces. In this chapter, we 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. Data sets whose classes can be separated exactly by linear decision surfaces are said to be linearly separable.

2.1 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

$$= w_0 + \sum_{j=1}^{D-1} w_j x_j = w_0 + x^T w \tag{37}$$

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

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

A naive way to perform classification with two output classes is to choose an arbitrary activation function and for output less than 0.5 we have class 0 and viceversa class 1. We could be tricked into thinking that this classifier can predict nonlinear boundaries, but it's not the case. In fact, taken the boundary

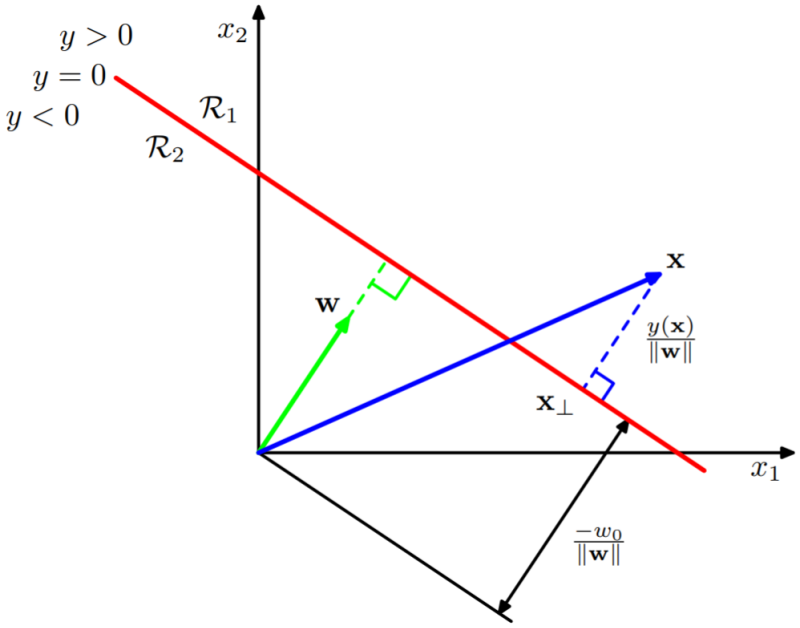
$$\begin{aligned} f(w_0 + x^T w) &= 0.5 \\ f^{-1}(f(w_0 + x^T w)) &= f^{-1}(0.5) \\ w_0 + x^T w &= f^{-1}(0.5) \\ w_0^* + x^T w &= 0, \\ \text{with} \quad w_0^* &= w_0 - f^{-1}(0.5) \end{aligned} \tag{38}$$

As we can see (38) represents an hyperplane, hence the decision surfaces are linear functions of x, even if the activation function is nonlinear. As in regression we can use basis function to make the input space linearly separable. The bad thing about this approach is that the model is no longer linear in the parameters, so no closed form solution exists.

2.1.1 Geometric interpretation

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 surface[10], 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 r we can project[11]a point x on the boundary on w

$$\begin{aligned} w^T x + w_0 &= 0 \\ w^T x &= -w_o \\ d = \frac{w^T x}{\|w\|_2} &= -\frac{w_0}{\|w\|_2} \end{aligned} \tag{39}$$



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

$$x = x_\perp + r\frac{w}{\|w\|_2}$$

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

[11]We know that the projection of a vector a on another vector b is $proj_b(a) = \frac{ab}{\|b\|}$.

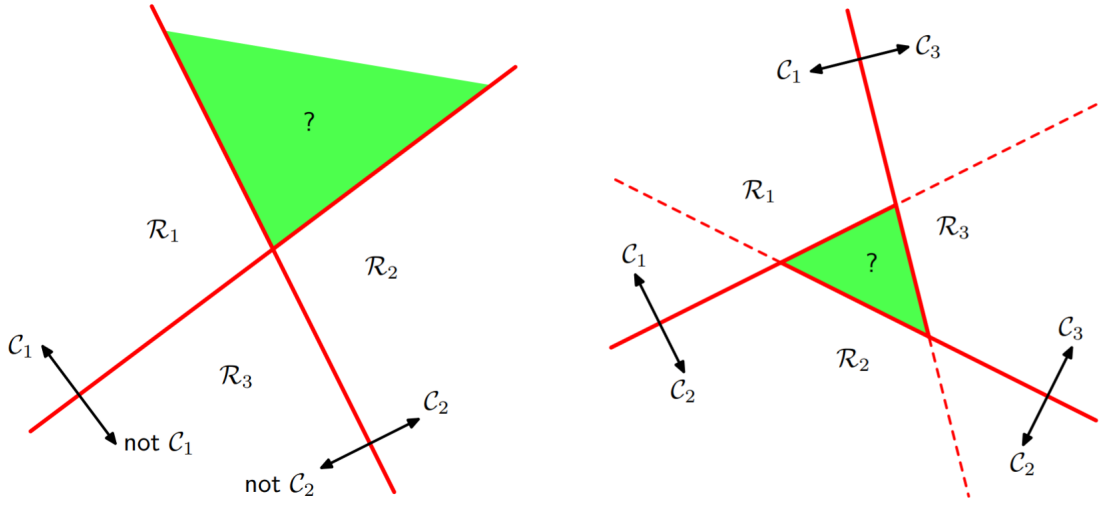
$$
\begin{aligned}
w^T x &= w^T x_\perp + w^T r \frac{w}{\|w\|_2} \\
w^T x + w_0 &= \underbrace{w^T x_\perp + w_0}_{=0} + w^T r \frac{w}{\|w\|_2} \\
y(x) &= w^T r \frac{w}{\|w\|_2} \\
y(x) &= r \frac{\|w\|_2^2}{\|w\|_2} \\
r &= \frac{y(x)}{\|w\|_2}
\end{aligned} \tag{40}
$$

2.1.2 Multiple outputs

Now consider the extension of linear discriminants to $K > 2$ classes. We might be tempted be 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. There are regions in input space that are ambiguously classified, as we can see in the left-hand diagram.

One-versus-one An alternative is to introduce K(K - 1)/2 binary discriminant functions, one for every possible pair 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, as illustrated in the right-hand diagram.



Linear discriminant functions We can avoid these difficulties by considering a single K-class discriminant comprising K linear functions of the form

$$y_k(x) = w_k^T x + w_{ko}, \quad where \quad k = 1, \ldots, K \tag{41}$$

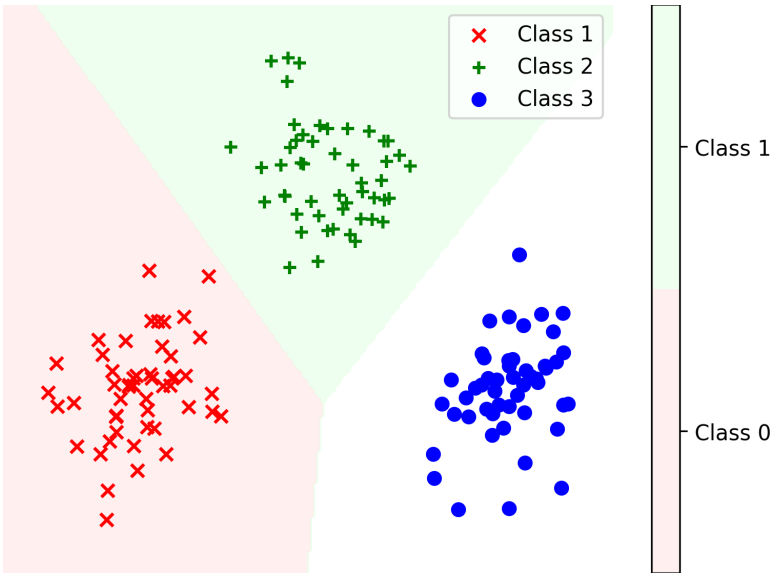
and then assigning a point x to class C_k if $y_k(x) > y_j(x), \quad \forall j \neq k$. The decision boundary between class C_k and class C_j is therefore given by $y_k(x) = y_j(x)$ and hence corresponds to a (D - 1)-dimensional hyperplane. The resulting decision boundaries are singly connected[12]and convex. 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

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

A simple example with three classes in a 2-dimensional space:



2.2 Least square for classification

Consider a general classification problem with K classes, with a one-hot 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

$$y_k(x) = w_k^T x + w_{ko}, \quad where \quad k = 1, \ldots, K \tag{42}$$

[12]It means that all the linear functions are ray originating from a common point

Using vector notation,

$$y(x) = \tilde{W}^T \tilde{x}, \quad where$$
$$\tilde{W} = \left[\underbrace{\begin{bmatrix} w_{10} \\ \vdots \\ w_{1D} \end{bmatrix}}_{w_1^T} \quad \cdots \quad \underbrace{\begin{bmatrix} w_{K0} \\ \vdots \\ w_{KD} \end{bmatrix}}_{w_K^T} \right], \quad [D+1xK]$$
$$\tilde{x} = (1, x^T)^T, \quad [D+1x1]$$

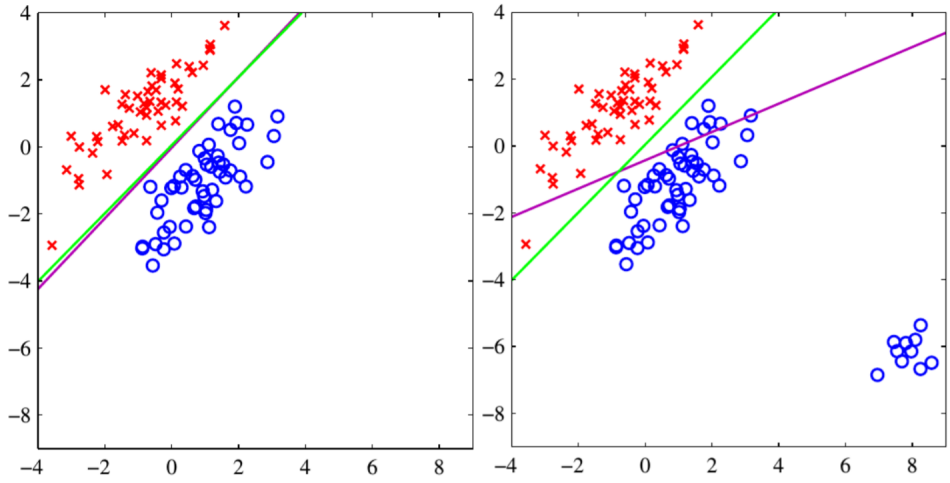
We classify the input x into class C_k if $y_k(x) > y_j(x), \quad \forall j \in [1, K] \setminus k$. $y_k(x)$ corresponds to the k^{th} element of $y(x)$. To estimate the parameter we can follow what we did for the regression problem. To minimize least square,

$$\tilde{W} = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T T, \quad [(D+1)xK] \tag{43}$$
$$\tilde{X} \quad [Nx(D+1)]$$
$$\tilde{T} \quad [NxK]$$

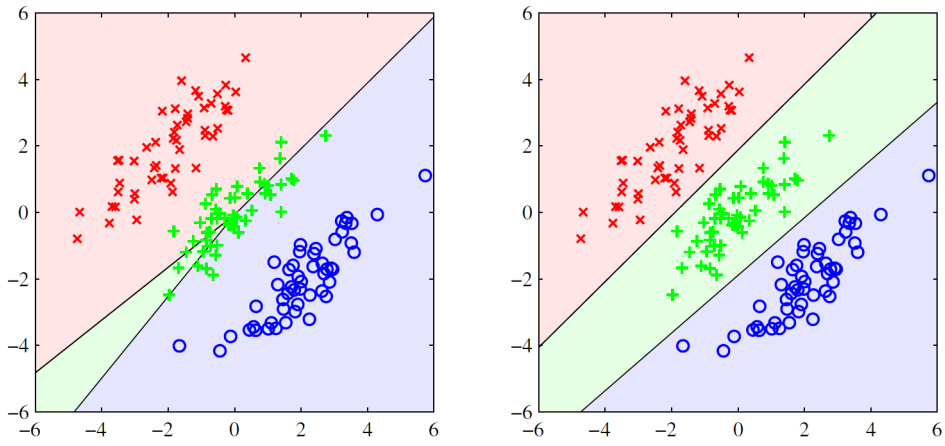
2.2.1 Least squares problems

The least square approach is problematic in some cases.

Outliers Least square is very sensitive to outliers. Least square tries to find a line which is the most close to all points. It does so evaluating 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.



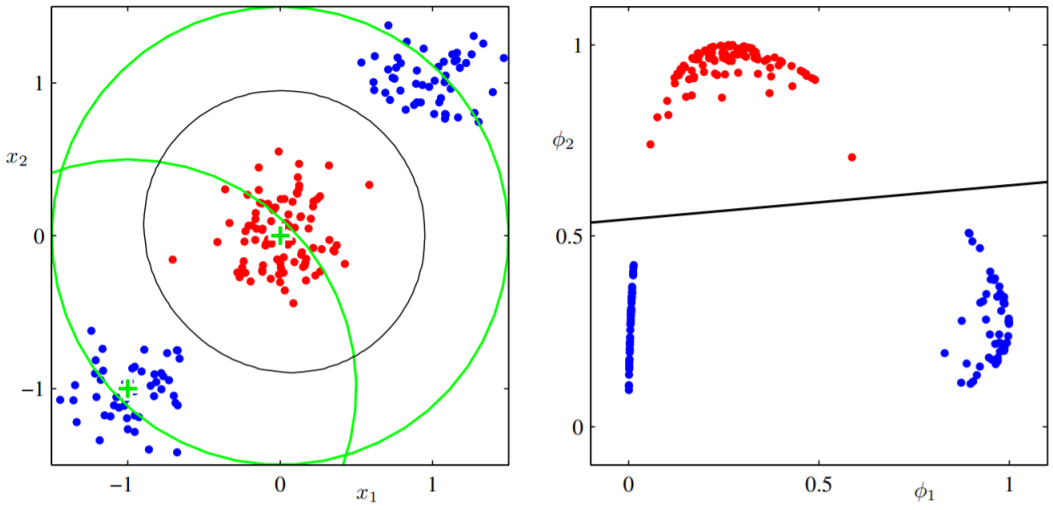
Non-Gaussian distributions We recall that least squares corresponds to the maximum likelihood under the assumption of a Gaussian conditional distribution, whereas binary target vectors clearly have a distribution that is far from Gaussian. As we can see in the figure below, even though the three classes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



2.2.2 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 nonlinear 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 nonlinear boundaries in the original input space.

Example - Nonlinear basis Illustration of the role of nonlinear 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. 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. The right-hand plot shows the corresponding feature space $(\Phi_1(x), \Phi_2(x))$ together with the linear decision boundary. This corresponds to a nonlinear decision boundary in the original input space, shown by the black curve in the left-hand plot.



2.3 Perceptron

The perceptron is an algorithm for online[13]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

$$y(x) = f(w^T \Phi(x)), \quad where$$
$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

Target values are +1 for C_1 and -1 for C_2. The algorithm finds the separating hyperplane by minimizing the distance of misclassified points to the decision boundary. Our objective is

[13]Online means that it is an iterative approach which calculate the solutions with multiple steps

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 $x_n \in C_2$. We define the error as follow:

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

$w^T\Phi(x_n)$ being our prediction, we can consider it as the non normalized distance from the decision boundary, and t_n the correct target value.

Now we define an error function for the parameter optimization,

$$L_P(w) = -\sum_{n \in M} w^T\Phi(x_n)t_n \tag{44}$$

To perform minimization we use stochastic gradient descent(online):

$$w^{(k+1)} = w^{(k)} - \alpha\nabla L_P(w) = w^{(k)} + \alpha\Phi(x_n)t_n \tag{45}$$

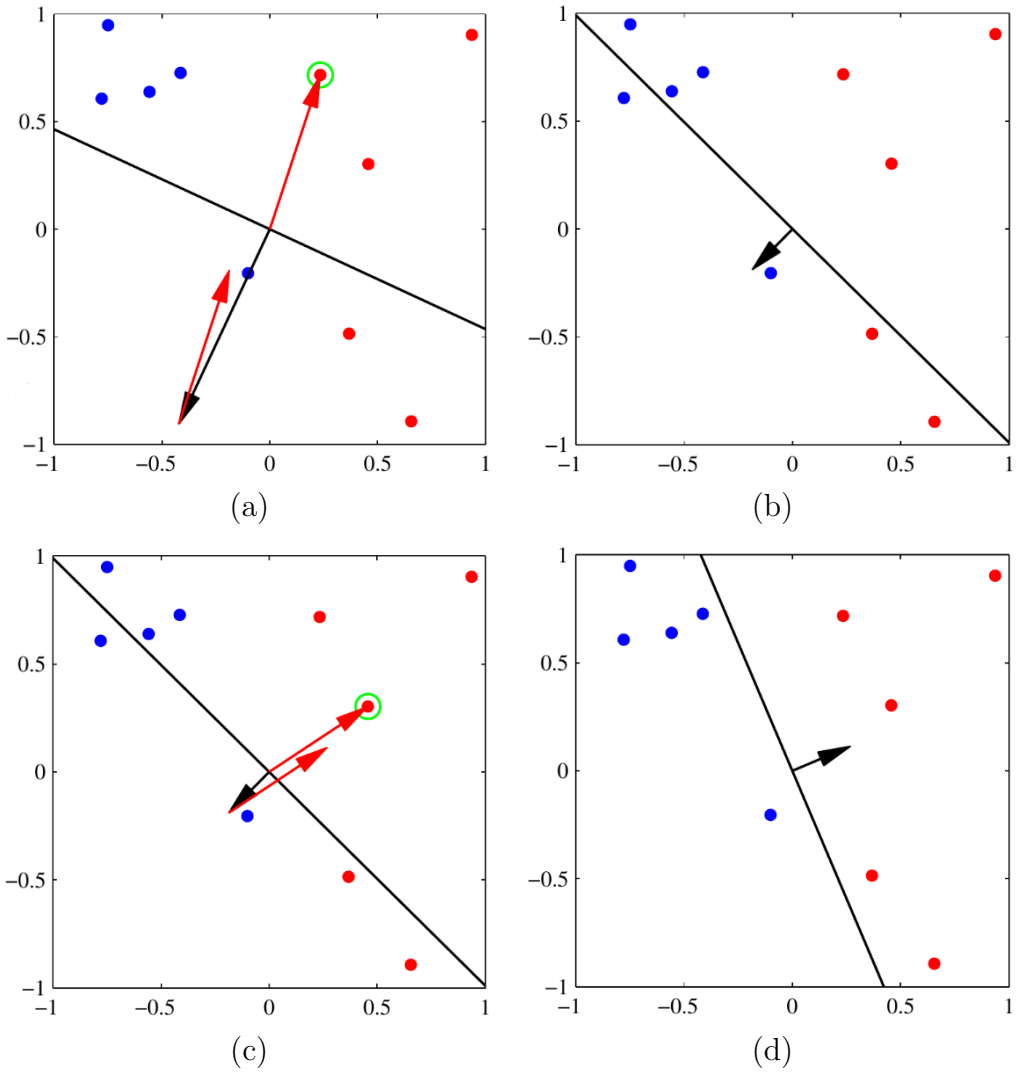
Note - Loss sign We have a minus sign in the loss function because $w^T\Phi(x_n)t_n$ will always be negative. This is due to the fact that if $w^T\Phi(x_n)$ is misclassified, then it will have an opposite sign compared to t_n.

Note - Learning rate The learning rate α can be set to 1 because it doesn't change the direction of w. We assume that w starts from the origin, so a scaling of the vector doesn't affect the boundary definition.

2.3.1 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
$k \leftarrow 0$;
while *!converged* **do**
  $k \leftarrow k + 1$;
  $n \leftarrow k\%N$;
  if $\hat{t_n} \neq t_n$ **then**
    $w^{(k+1)} \leftarrow w^{(k)} + \Phi(x_n)t_n$;
  end
end



Theorem 2.1 (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*

The number of steps before convergence may be substantial. We are not able to distinguish between non-separable problems and slowly converging ones. If multiple solutions exist, the one found depends by the initialization of the parameters and the order of presentation of the data points. Another fact about algorithm steps is that the effect of a single update is to reduce the error due to the misclassified pattern, but this does not imply that the loss is reduced at each stage. This means that we reduce the error of the misclassified point we are considering, but we have no guarantee that the error of the other points gets better.

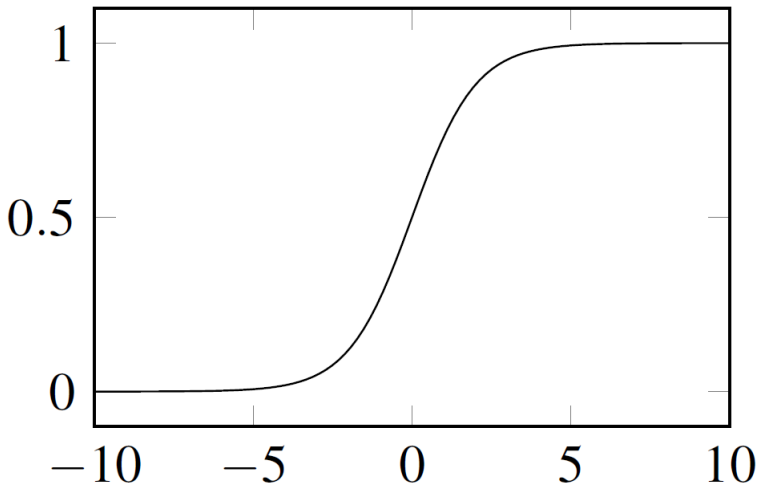
2.4 Logistic regression

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 discriminative model so we model directly the posterior probability $P(C_k|\Phi)$. In detail we use a logistic sigmoid function[14].

$$P(C_1|\Phi) = \sigma(w^T\Phi) = \frac{1}{1+e^{-w^T\Phi}} \tag{46}$$

$$P(C_2|\Phi) = 1 - P(C_1|\Phi) \tag{47}$$

Note Logistic regression is a classification method not a regression one.



2.4.1 Maximum Likelihood for logistic regression

We now use maximum likelihood to determine the parameters of the logistic regression model. Now we have to define a suitable loss function to use. To do so, we first analyze our inputs and outputs. We have a dataset $D = \{x_n, t_n\}$, $n = 1, \ldots, N$

$$y_n(\Phi(x_n)|w) = P(t_n|\Phi(x_n)), \quad t_n \in [0,1] \Rightarrow t_n \sim Be(y_n(\Phi(x_n)|w))$$

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

Knowing that t_n follows a Bernoulli distribution we have,

$$P(t_n) = y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n} \tag{48}$$

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

The equation (48) 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 $P(t_n)$.

$$
\begin{aligned}
l(w) &= \prod_{n=1}^{N} P(t_n) \\
&= \prod_{n=1}^{N} y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n} \\
\Downarrow &\quad \text{Transition to ln to simplify calculus.} \\
&\quad \text{Min \& Max remain the same} \\
l(w) &= ln(\prod_{n=1}^{N} y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\
&= \sum_{n=1}^{N} ln(y_n(\Phi(x_n)|w)^{t_n}(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\
&= \sum_{n=1}^{N} ln(y_n(\Phi(x_n)|w)^{t_n} + \sum_{n=1}^{N} ln(1 - y_n(\Phi(x_n)|w))^{1-t_n}) \\
&= \sum_{n=1}^{N} t_n ln(y_n(\Phi(x_n)|w) + \sum_{n=1}^{N} (1 - t_n) ln(1 - y_n(\Phi(x_n)|w))) \\
&= \sum_{n=1}^{N} t_n ln(y_n(\Phi(x_n)|w) + (1 - t_n) ln(1 - y_n(\Phi(x_n)|w))) \\
\Downarrow &\quad \text{For simplicity we remove } y_n \text{ parameters} \\
&= \sum_{n=1}^{N} t_n ln(y_n) + (1 - t_n) ln(1 - y_n)
\end{aligned}
$$

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 from of our loss function $l(w)$.

$$
L(w) = -\sum_{n=1}^{N} t_n ln(y_n) + (1 - t_n) ln(1 - y_n) \quad \text{(Binary cross-entropy)} \tag{49}
$$

Now we have to minimize the loss function. Unfortunately L(w) is no longer linear, so we have to use iterative methods. Nonetheless, we need to find the gradient of $L(w)$.

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

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

There is no closed form solution, due to non-linearity of the logistic sigmoid function, but the error function is convex[15]and can be optimized by standard gradient-based optimization techniques.

Note - Activation function If we replace the sigmoid with a step function we obtain the perceptron algorithm. Both algorithm use the same updating rule $w \leftarrow w - \alpha(y(x_n, w) - t_n)\Phi(x_n)$

2.4.2 Multiclass logistic regression

Logistic regression can be expanded to multiclass 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. 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 multiclass case, we have to find a new way to ensure that the second property 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,

$$P(C_k|\Phi) = y_k(\Phi|w) = \frac{exp(w_k^T\Phi)}{\sum_{j=1}^{K} exp(w_j^T\Phi)} \tag{50}$$

Given T, a $[N \times K]$ matrix containing all output vector t_n $[K \times 1]$

$$P(T|\Phi) = \prod_{n=1}^{N} \underbrace{\left(\prod_{k=1}^{K} P(C_k|\Phi(x_n))^{t_{nk}} \right)}_{\substack{\text{Only one term corresponding} \\ \text{to correct class}}}$$
$$= \prod_{n=1}^{N} \left(\prod_{k=1}^{K} y_{nk}^{t_{nk}} \right)$$

So the loss function can be expressed as

$$L(w_1, \ldots, w_K) = -ln(P(T|\Phi)) = -\sum_{n=1}^{N} \left(\sum_{k=1}^{K} t_{nk} ln(y_{nk}) \right) \tag{51}$$

Last but not least the gradient will be

$$\nabla L_{w_k} = \sum_{n=1}^{N} (y_{nk} - t_{nk})\Phi(x_n) \tag{52}$$

[15]Convex in this case implies that $L(w)$ has only one minimum

3 Bias-Variance and Model Selection

3.1 "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 data and $\mathcal{F}$ as the set of all possible concept $y = f(x)$ (all possible problems).

Theorem 3.1 (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.*

Corollary 3.1.1. *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.* $ACC_G(L_1) < ACC_G(L_2)$

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.

3.2 Bias-Variance trade-off

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

3.2.1 Bias-Variance decomposition

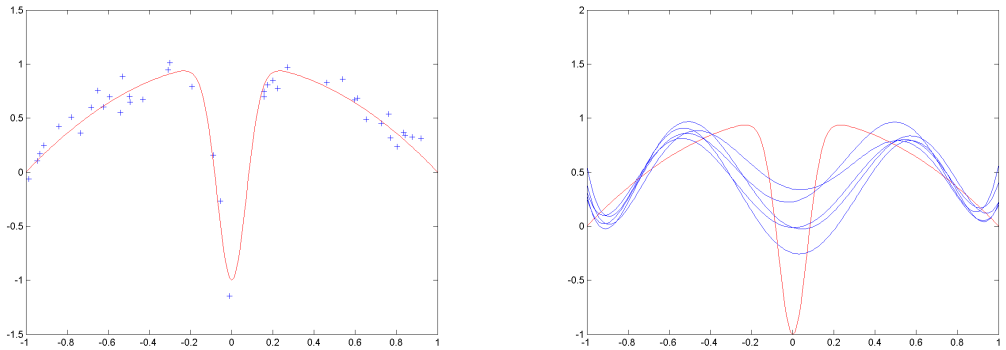
The bias–variance decomposition is a way of analyzing a learning algorithm's expected generalization error with respect to a particular problem as a sum of three terms, the bias, variance, and a quantity called the irreducible error, resulting from noise in the problem itself. Assume to have a dataset $\mathcal{D}$ with N samples taken from $t_i = f(x_i) + \epsilon$, where $E[\epsilon] = 0$ and $Var[\epsilon] = \sigma^2$ Our objective is to find a model $y(x)$ that approximate f as well as possible on unseen data. Let's consider the expected square error on an unseen sample x

$$
\begin{aligned}
E[(t - y(x))^2] &= E[t^2 + y(x)^2 - 2ty(x)] \\
&= E[t^2] + E[y(x)^2] - E[2ty(x)] \\
&\text{We can substitute t with its true function } f(x) \\
&= E[t^2] \pm E[t]^2 + E[y(x)^2] \pm E[y(x)]^2 - f(x)E[2y(x)] \\
&= Var[t] + E[t]^2 + Var[y(x)] + E[y(x)]^2 - 2f(x)E[y(x)] \\
&= Var[t] + Var[y(x)] + E[t]^2 + E[y(x)]^2 - 2f(x)E[y(x)] \\
&= Var[t] + Var[y(x)] + (f(x) - E[y(x)])^2 \\
&= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(x)]}_{\text{Variance}} + \underbrace{E[f(x) - y(x)]^2}_{\text{Bias}^2}
\end{aligned} \tag{53}
$$

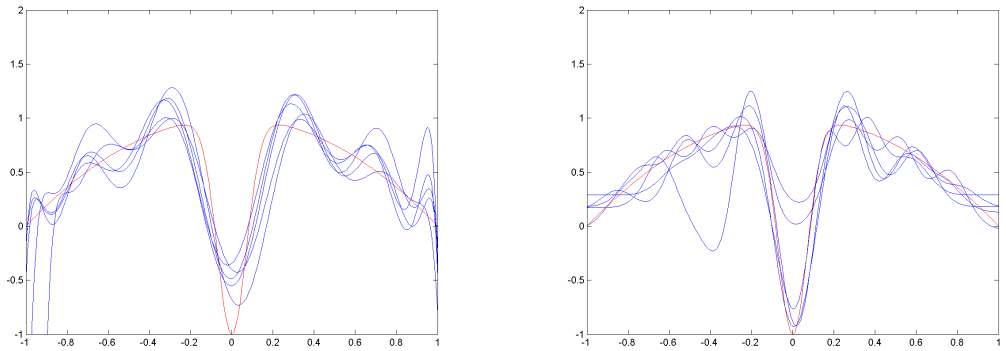
As we can see we have three sources of error

- σ^2: This is the irreducible error. It generates directly from the problem.
- Variance: This is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).
- Bias: 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).

It's worth mentioning that the expectation is performed over different realization of the training set D. Let's see an example



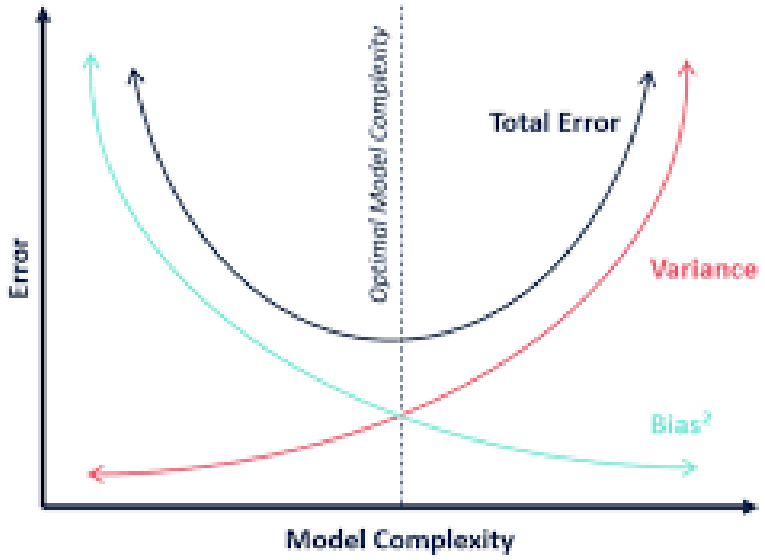
(a) (b)



(c) (d)

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 don't fit on data. 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.

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.



Example (K-NN) In the case of K-Nearest Neighbor we can derive an explicit analytical expression of the expected squared prediction error

$$E[(t - y(x))^2] = \sigma^2 + \frac{\sigma^2}{K} + \left(f(x) - \frac{1}{K} \sum_{i=1}^{K} f(x_i) \right)^2 \tag{54}$$

- 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

3.2.2 Training-test error

Given a data set $\mathcal{D}$ with N samples, we can split it in a training set and a test set. To calculate the training error we have to choose a loss function(e.g. RSS). We can distinguish between

- Regression: $L_{train} = \frac{1}{N}\sum_{n=1}^{N}(t_n - y(x_n))^2$
- Classification: $L_train = \frac{1}{N}\sum_{n=1}^{N}(I(t_n \neq y(x_n)))$

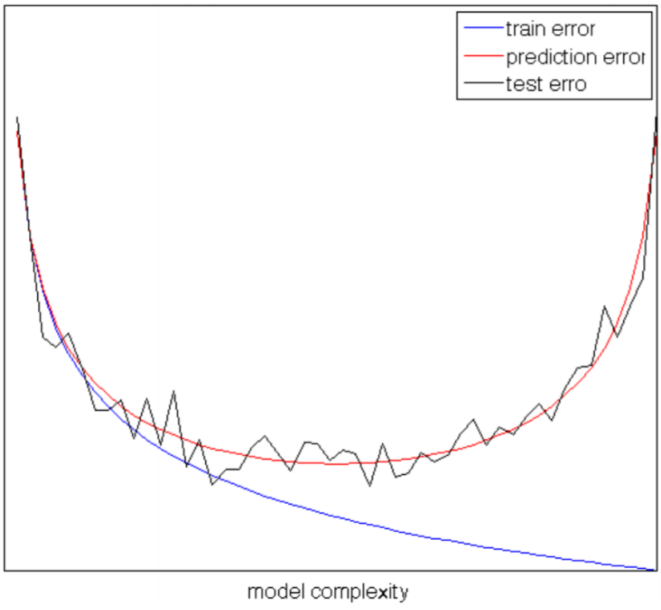
The training error measures how close our model is to training data. As we can imagine, increasing model complexity decreases the training error. But we also know that passed a certain complexity the generalization capability of our model will be decrease. So training error is not a good estimator of our model performance because is monotonically decreasing with model complexity, so it is an optimistically biased estimate of prediction error. Our objective is to estimate the true prediction error,

- Regression: $L_{true} = \int (f(x) - y(x))^2 p(x)dx$
- Classification: $L_{true} = \int I(f(x) \neq y(x))p(x)dx$

This 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.

$$L_{test} = \frac{1}{N_{test}}\sum_{n=1}^{N_{test}}(t_n - y(x_n))^2 \tag{55}$$

The test error is calculated with the test set. It is very important to not mix the training set and the test set, in order to get an unbiased estimation of the prediction error. We use the training set to estimate the model's parameters and the test set to estimate the prediction error.



3.3 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.

3.3.1 Curse of dimensionality

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. Working with high-dimensional data is difficult because the variance of the model becomes larger, we need more samples and computational power to counteract this phenomenon. So a common pitfall when we can't solve a learning problem is to add features to the input space. However the number of samples remain the same and so the importance of the old features decreases leading to worse performance. 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 hypercube with unit volume. Suppose that we have n points uniformally distributed inside the hypercube. 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.

3.3.2 Feature selection

Identifies a subset of input features that are most related to the output. 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
2. For $k = 1, \ldots, M$ (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 $\mathcal{M}_0, \ldots, \mathcal{M}_M$ using some criterion

We can start from understanding how we can evaluate and select a single best model. Obviously we can't use training error, because the most complex model will perform "better", even though is overfitting. We would like to use the test error to evaluate which is the best model among a collection of models with different numbers of features. There are two approaches to estimate the test error

- Direct estimation with a validation approach
- Making an adjustment to the training error to account for model complexity

Direct estimation So far we have divided the dataset into two subsets: training 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**. To have a fair evaluation we can introduce a new subset of data: **Validation set**.

- **Training dataset**: The sample of data used to fit the model
- **Validation dataset**: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset, while tuning model hyperparameters[16]
- **Test dataset**: The samples of data used to provide an unbiased evaluation of the final model fit

In practice, the training set is used to learn the parameters, the validation set to tune the hyperparameter and the test set to evaluate the performance of our fit. Validation set can't be use directly to estimate the error, because is used indirectly in the training phase. The solution is to use **cross validation**. 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[17]. We can randomly divide the training data into k equal subsets: $\mathcal{D}_1, \ldots, \mathcal{D}_k$. 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.

[16]The hyperparameters 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

[17]Every sample is used as training data and validation data in different phases of the learning process

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 $k \sim 10$

Algorithm 2: k-fold cross validation

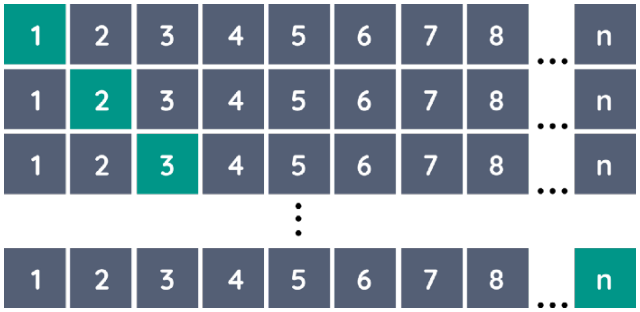
Output: Validation error L_{k-fold}
Input : Data set $\mathcal{D}$ splitted in $\mathcal{D}_1, \ldots, \mathcal{D}_k$
for $i \leftarrow 0$ **to** k **do**
  Train model $y_{\mathcal{D}\setminus\mathcal{D}_i}$ on $\mathcal{D} \setminus \mathcal{D}_i$
  $L_{\mathcal{D}_i} \leftarrow \frac{k}{N} \sum_{(x_n,t_n)\in\mathcal{D}_i} (t_n - y_{\mathcal{D}\setminus\mathcal{D}_i}(x_n))^2$
end
$L_{k-fold} \leftarrow \frac{1}{k} \sum_{i=1}^{k} L_{\mathcal{D}_i}$

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.

LOO (Leave One Out) In this case we consider at each iteration a validation set with only one sample.

Algorithm 3: LOO cross validation

Output: Validation error L_{LOO}
Input : Data set $\mathcal{D}$ with N samples
for $i \leftarrow 0$ **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
$L_{LOO} \leftarrow \frac{1}{N} \sum_{n=1}^{N} L_{\{n\}}$



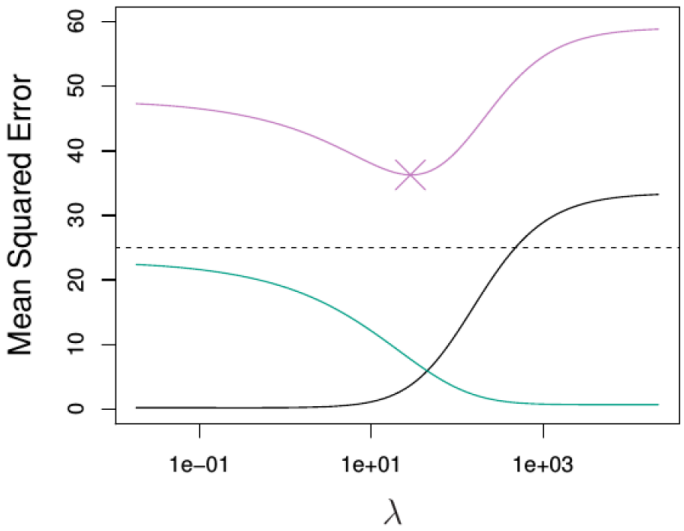
To evaluate different input features set, we apply the cross validation for every model having those input features. LOO is almost unbiased(pessimistically) as opposed to k-fold, but from a computational point of view k-fold is far better. For example, if we have 100.000 samples and each iteration of 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.

Adjustment techniques This approach tries to account for model complexity when evaluating the training error. There are several way to do that

- $C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma}^2)$
  Where d is the number of parameters, $\tilde{\sigma}^2$ is an estimate of the variance of the noise ϵ
- $BIC = \frac{1}{N}(RSS + log(N)d\tilde{\sigma}^2)$
  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 models.
- $AIC = -2log(L) + 2d$
  Where L is the maximized value of the likelihood function for the estimated model
- $AdjustedR^2 = 1 - \frac{RSS/(N-d-1)}{TSS/(N-1)}$
  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.

3.3.3 Regularization

We have already seen regularization approaches applied to linear models like 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. From the figure below we can see in action the bias-variance trade-off. When λ gets bigger we obtain simpler models, so the variance is reduced and the bias gets bigger. We need to find the optimal λ which minimizes the MSE. 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.



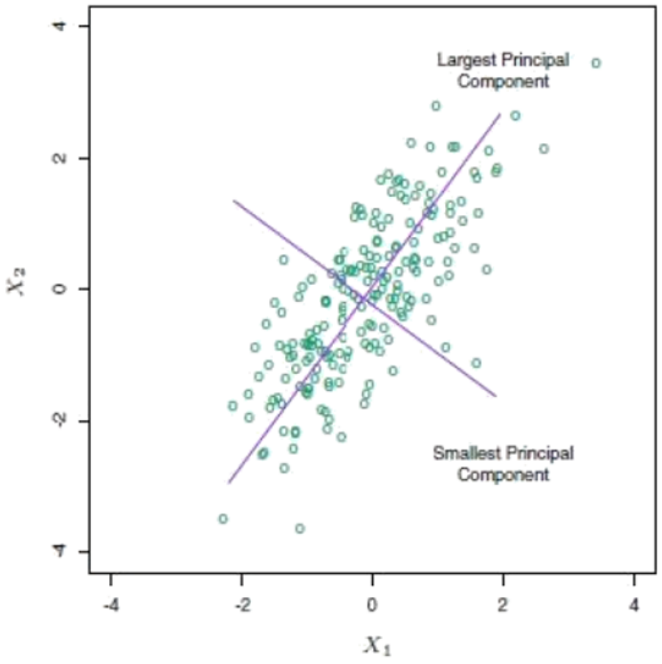
Black: squared bias, Green: variance, Purple: MSE, Dashed: minimum possible MSE, Purple cross: ridge regression model with minimum MSE

3.3.4 Dimension reduction

The approach we are going to present differs from the previous one because it doesn't operate directly on the original features. Dimension reduction methods transform the original features and then the model is learned on the transformed variables. It does so with an unsupervised learning approach. There are many techniques to perform dimensionality reduction,

- PCA (Principal Component Analysis)
- ICA (Independent Component Analysis)
- Self-organizing Maps
- Autoencoders
- ...

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. In other word, we want to find a line such that when the data is projected onto that line, it has the maximum variance. Then we find a new line, orthogonal to the first one, that has maximum projected variance. We repeat this process until we have covered a certain percentage of input variance or we have reached a given number of dimensions(lines).

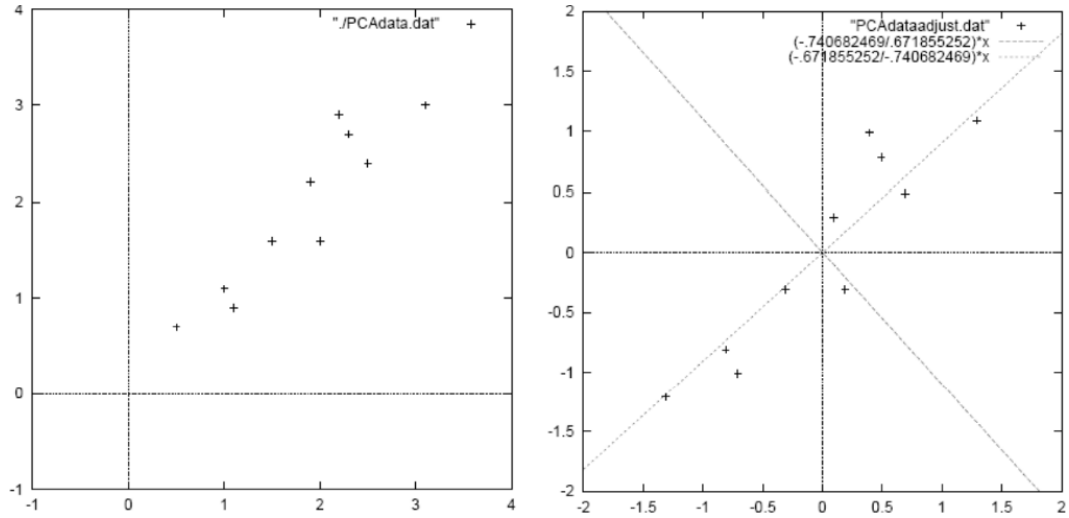


In a more formal way, the complete process consists of,

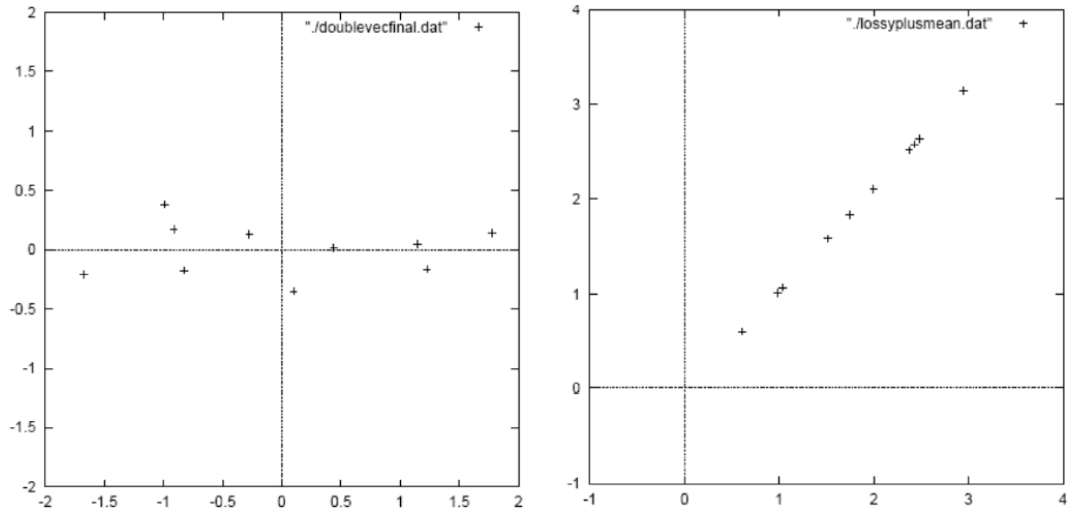
1. Compute the data mean $\overline{x} = \frac{1}{N}\sum_{n=1}^{N} x_n$ to later center the data on the origin
2. Compute the covariance matrix $S = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \overline{x})(x_n - \overline{x})^T$
3. Get the eigen values λ_i and eigen vector e_i of S
4. Select the first k largest eigen values. The relative eigen vector are the PCA components. $\frac{\lambda_i}{\sum_i \lambda_i}$ is the percentage of variance captured by the i^{th} principal component(PC)

Once we have found the new PCs, we can project the data onto the new orthogonal base $E_k = \begin{bmatrix} e_1 & \dots & e_k \end{bmatrix}$ $[M \times k]$.

$$X' = XE_k, \quad [N \times k] \tag{56}$$



Left: Original data, Right: Mean centered data with PCs overlayed



Left: Original data projected into full PC space, Right: Original data reconstructed with first PC

PCA is very useful to reduce computational complexity. In supervised learning reducing the number of dimension leads to smaller hypothesis space resulting in models that are less prone to overfitting. PCA can be seen as a noise reduction method. This technique have some defects,

- Fails when data consists of multiple cluster
- Directions of greatest variance may not be most informative

- Computational problems with many dimensions
- PCA computes linear combination of features, but data often lies on a nonlinear manifold

3.4 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 only. 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. Their basic idea is to learn several models and combine them, instead of learning only one model. Typically they greatly improve accuracy.

3.4.1 Bagging

As we have said before bagging reduces the variance. We do so by averaging multiple models together. We know that

$$Var[\bar{X}] = \frac{Var[X]}{N} \tag{57}$$

In order to be able to apply this method we need to find a way to generate multiple models from one dataset. To do so we use bootstrap. This statistical technique consists in generating samples of size B (called bootstrap samples) from an initial dataset of size N by randomly drawing with replacement B observations.

After boostraping we train a model for each bootstrap sample. 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. Bagging improves performance for unstable learners which vary significantly with small changes in the data set. In practice we want to average multiple overfitting model. 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. In practice bagging almost always helps.

3.4.2 Boosting

The objective of boosting is to minimize bias. This is accomplished by training weak learners in a sequential manner[18]. You might be wondering what 'sequential' implies. Sequential training means that each model is trained based on the predictions of the preceding model. Here are the steps to implement boosting:

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 samples 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

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.

3.4.3 Bagging vs Boosting

- Bagging reduces **variance**
- Boosting reduces **bias**
- Bagging doesn't work so well with **stable models**. 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 it is also more common for boosting to hurt performance
- The weights grow **exponentially**
- Bagging is easier to **parallelize**

[18] A weak learner is a learner that has a slightly better performance than chance prediction on any training set

4 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 way easier 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 hypothesis space we are less prone to overfitting. A question arises naturally. Can we estimate how many samples are necessary given an hypothesis space? We can answer this question in a theoretical way. Please remind that from a practical point of view it is rarely used.

4.1 PAC-Learning

To introduce the ingredients of the theoretical setting we use a character recognition task. Given an array of n bits encoding a binary-valued image we have

- 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 **Hypothesis 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".
- $\mathcal{P}$ **Probability distribution over X**. Training instances are generated by a fixed, unknown probability distribution over X

The learner observes a sequence $\mathcal{D}$ of training example $< x, c(x) >$ for some target concept $c \in C$ and x is drawn from $\mathcal{P}$. The learner must output a hypothesis h estimating c. h is evaluated by its performance on subsequent instances drawn according to $\mathcal{P}$

$$L_{true} = P_{x \in \mathcal{P}}(c(x) \neq h(x))$$

Our objective is to bound L_{true} given L_{train}. Now we introduce the so called **version space** $VS_{H,\mathcal{D}}$. It is a subset of H where the training error L_{train} is zero. So the hypothesis h are always correct on training instances. For now, we assume that VS is non-empty. Making some consideration we can bound L_{train} inside VS.

Theorem 4.1 (L_{train} bound in VS)**.** *If the hypothesis 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 then ϵ is less than $|H|e^{-\epsilon N}$:*

$$P(\exists h \in H : L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \leq |H|e^{-\epsilon N}$$

Proof.

$$P((L_{train}(h_1) = 0 \wedge L_{true}(h_1) \geq \epsilon) \vee \cdots \vee (L_{train}(h_{|H|}) = 0 \wedge L_{true}(h_{|H|}) \geq \epsilon))$$

$$\begin{aligned}
&\leq \sum_{h \in H} P(L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) && \text{Union bound} \\
&\leq \sum_{h \in H} P(L_{train}(h) = 0 | L_{true}(h) \geq \epsilon) && \text{Bound using Bayes' rule} \\
&\leq \sum_{h \in H_{bad}} (1-\epsilon)^N && \text{Bound on individual } h_i s \\
&\leq |H|(1-\epsilon)^N && |H|_{bad} \leq |H| \\
&\leq |H| e^{-\epsilon N} && (1 - \epsilon \leq e^{-\epsilon}, for \quad 0 \leq \epsilon \leq 1)
\end{aligned}$$

□

We can notice that the dimension of the hypothesis space influences negatively the bound, in fact a larger searching space will give us less guarantees on the value of L_{true}. 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}. Now we can bound the probability of $P(\exists h \in H : L_{train}(h) = 0 \wedge L_{true}(h) \geq \epsilon) \leq |H| e^{-\epsilon N}$. We can set a parameter δ

$$|H| e^{-\epsilon N} \leq \delta$$

After choosing δ we can calculate N or ϵ.
Given ϵ and δ

$$N \geq \frac{1}{\epsilon}\left(ln(|H|) + ln\left(\frac{1}{\delta}\right)\right) \tag{58}$$

Given N and δ

$$\epsilon \geq \frac{1}{N}\left(ln(|H|) + ln\left(\frac{1}{\delta}\right)\right) \tag{59}$$

Note $|H|$ can be very large. If we take as an example a binary decision problem with M binary inputs(features), the size of H will be 2^{2^M}. So N will have a exponential relationship with M. This is related to the curse of dimensionality.

Example Suppose H contains conjunctions of constraints on up to M boolean attributes (i.e., M literals). In this case $|H| = 3^M$. How many examples are sufficient to ensure with probability at least $(1 - \delta)$ that every h in $VS_{H,\mathcal{D}}$ satisfies $L_{true}(h) \leq \epsilon$?

$$N \geq \frac{1}{\epsilon}\left(Mln(3) + ln\left(\frac{1}{\delta}\right)\right)$$

Now we are ready to define formally what PAC[19]is. Consider a class C of possible target concepts defined over a set of instances X of length n, and a learner L using hypothesis space H.

Definition 4.1. *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 4.2. *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 size(c)*[20]

Now we need to generalize to problems where the VS is empty, because usually the train error is not equal to zero(agnostic learning). We can simply bound the difference between L_{train} and L_{true} in H.

$$L_{true}(h) \leq L_{train} + \epsilon$$

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 4.3. *For N i.i.d. coin flips $X_1, \ldots, X_N$, where $X_i \in \{0,1\}$ and $0 < \epsilon < 1$, we define the empirical mean $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$, obtaining the following bound*

$$P(E[\bar{X}] - \bar{X} > \epsilon) \leq e^{-2N\epsilon^2}$$

Theorem 4.2. *Given an hypothesis 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 is very similar to what we have found in the non-empty case. Like we did before, we can calculate the number of example needed given ϵ and δ

$$N \geq \frac{1}{2\epsilon^2}\left(ln(|H|) + ln\left(\frac{1}{\delta}\right)\right)$$

Or ϵ given N and δ.

$$\epsilon \geq \sqrt{\frac{ln(|H|) + ln(\frac{1}{\delta})}{2N}}$$

[19]Probably Approximately Correct learning. Probably refers to δ and it is the confidence. Approximately refers to ϵ and it is the accuracy

[20]size(c): Number of bit necessary to express c. It comes from information theory

Now we can rewrite the gap between L_{train} and L_{true} as

$$L_{true}(h) \leq \underbrace{L_{train}(h)}_{Bias} + \underbrace{\sqrt{\frac{ln(|H|) + ln(\frac{1}{\delta})}{2N}}}_{Variance} \tag{60}$$

Once more, we can see how $|H|$ influences the loss function. For large $|H|$ we assume a low bias because it's more probable to find a good h and a high variance. 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).

4.2 VC Dimension

So far we have considered only finite hypothesis space. If we use the bound that we have just found in an infinite[21]H, we would have infinite variance. This is not the case, for infinite H the previous bound is too pessimistic. So we need to find a new one. 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**. To lay the ground for our theoretical discussion, we need to introduce two definitions,

Definition 4.4 (Dichotomy)**.** *A dichotomy of a set S is a partition of S into two disjoint subsets*

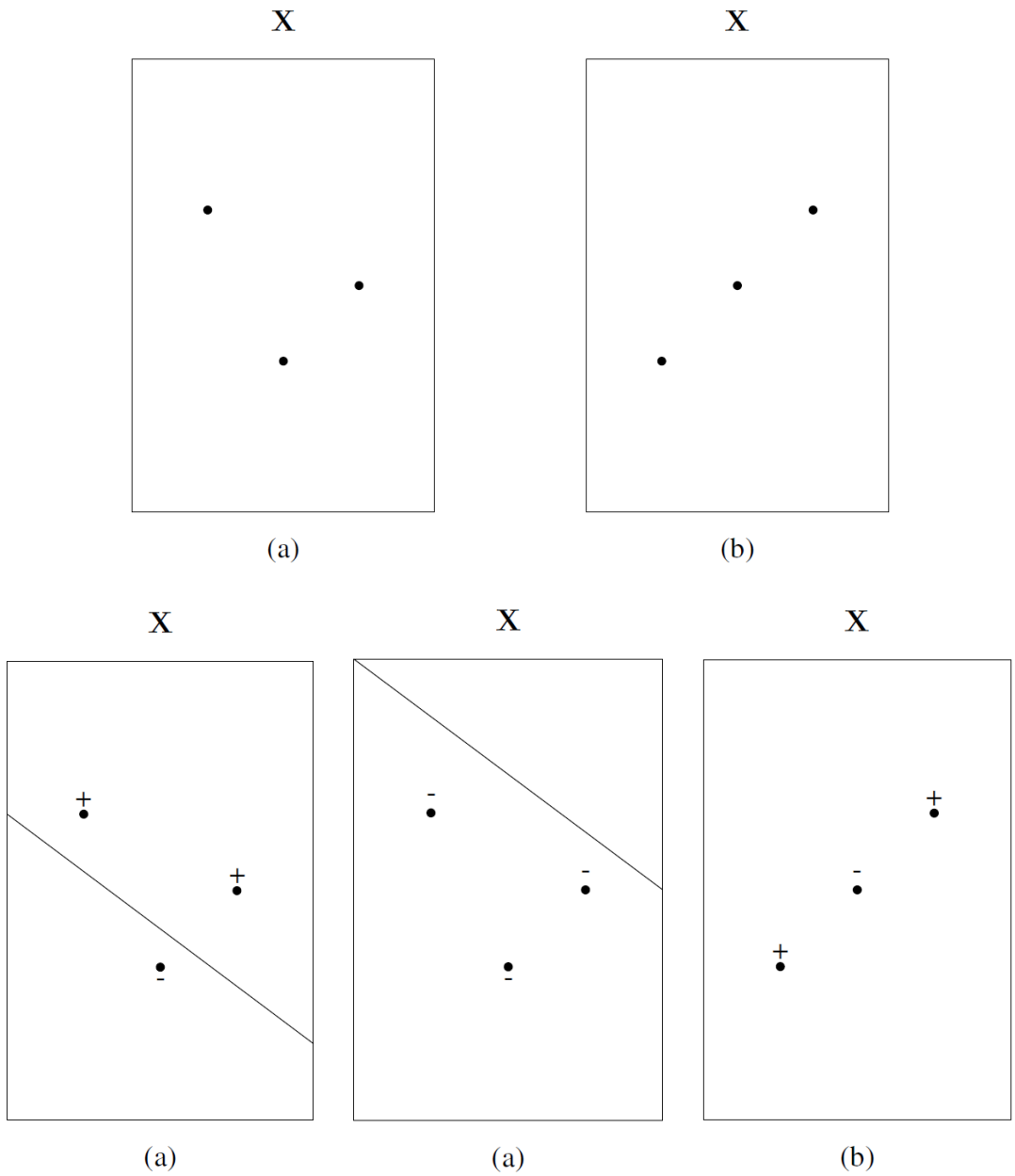
Definition 4.5 (Shattering)**.** *A set of instances S is shattered by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy*

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. As a reminder we are still considering only binary classification problems.

Example We consider an instances set with three example[22]and an hypothesis 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 doesn't shatter (b).

[21]Infinite hypothesis spaces are very common. For example linear regression or classification have infinite hypothesis spaces

[22]This is still infinite because every instance can have a different "position" in the instances set



Note that for four instances, H won't shutter S.[23] Now we can define what is the VC dimension

Definition 4.6 (VC dimension)**.** *The Vapnik-Chervonenkis dimension, $VC(H)$, of hypothesis 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. In fact, we can have a configuration of three instances whereby every dichotomy is perfectly separable. This doesn't hold for four instances.

[23]This is the XOR problem. It's not linearly separable

Example Few examples of VC dimensions,

- Linear classifier: $VC(H) = M + 1$, for M features plus the constant term.
- Neural networks: $VC(H) = \#parameters$
- 1-Nearest Neighbor: $VC(H) = \infty$
- SVM with Gaussian Kernel: $VC(H) = \infty$. We will see SVM in future chapters.

Now 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 $(1 - \delta)$

$$N \geq \frac{1}{\epsilon}\left(4log_2\left(\frac{2}{\delta}\right) + 8VC(H)log_2\left(\frac{13}{\epsilon}\right)\right) \tag{61}$$

Equally we can express this as an upper bound for L_{true}

$$L_{true}(h) \leq L_{train}(h) + \sqrt{\frac{VC(H)(ln(\frac{2N}{VC(H)}) + 1) + ln(\frac{4}{\delta})}{N}} \tag{62}$$

Properties

Theorem 4.3. *The VC dimension of a hypothesis space $|H| < \infty$ is bounded from above*

$$VC(H) \leq log_2(|H|)$$

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

$$\begin{aligned} |H| &\geq 2^d \\ |H| &\geq 2^{VC(H)} \\ VC(H) &\leq log_2(|H|) \end{aligned}$$

□

Theorem 4.4. *Concept class C with $VC(C) = \infty$ is not PAC-learnable.*

5 Kernel methods

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.

Example - KNN A very famous example of non-parametric method is the k-nearest neighbour[24]. 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 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**.

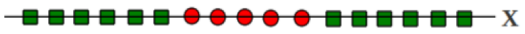
5.1 Kernels

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

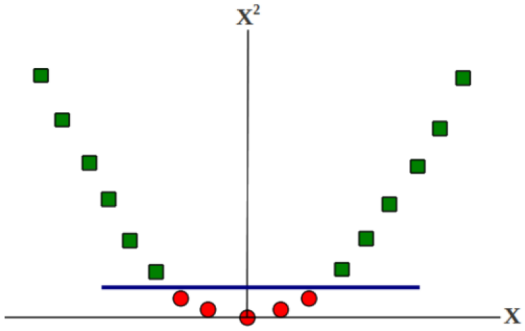
[24]It worth mentioning that k-nearest neighbour is not a kernel method. It's used only as an example for non-parametric methods

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



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 $\{x\} \rightarrow \{x, x^2\}$ the data are linearly separable.



This was the standard approach for extending linear models to non-linear problems

5.1.1 Kernel functions

Consider the following mapping Φ for an example $x = \{x_1, \ldots, x_M\}$

$$\Phi : x \rightarrow \{x_1^2, x_2^2, \ldots, x_M^2, x_1x_2, x_2x_3, \ldots, x_1x_M, \ldots, x_{M-1}x_M\}$$

This particular mapping is called second order monomial. Each new feature uses a pair of the original features. We can observe that the mapping will increase quadratically the number of features. This will have an impact on complexity because computing the mapping itself can be inefficient. Moreover, using the mapped representation could be inefficient too. 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. 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.

$$k(x, x') = \Phi(x)^T \Phi(x') \tag{63}$$

This kernel function is the metric of our non-parametric method and it measure the similarity between the points x and x'. A good consequence of being a scalar product is symmetry($k(x, x') = k(x', x)$)

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 product[25]between the two original samples. The result of the scalar product is maximum when the two vector are pointing in the same direction.
Besides the linear kernel, there are other type of kernel, commonly used in practice:

- **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 distance between arguments

$$k(x, x') = k(\|x - x'\|)$$

5.1.2 Dual representation

Many linear models for regression and classification can be reformulated in terms of dual representation, in which the kernel function arises naturally. We want this in order to be able to apply the kernel trick. In practice we want to describe our model not using feature but with a kernel. For every linear model exist a dual representation involving kernels. Let's take as an example ridge regression. We recall that the loss function for ridge regression is

$$L_w = \frac{1}{2}\sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)^2 + \frac{\lambda}{2}w^T w$$

and its gradient is

$$\begin{aligned}\overset{w}{\nabla} L &= \frac{1}{2}2\sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T + \frac{\lambda}{2}2w^T \\ &= \sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T + \lambda w^T\end{aligned}$$

Putting $\overset{w}{\nabla} L = 0$ we have

$$\begin{aligned}-\lambda w^T &= \sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T \\ w^T &= -\frac{1}{\lambda}\sum_{n=1}^{N}(w^T\Phi(x_n) - t_n)\Phi(x_n)^T\end{aligned}$$

[25] $x \cdot x' = \|x\|\|x'\|\cos\theta$, where θ is the angle between x and x'

$$
\begin{aligned}
&\text{Define } a_n = -\frac{1}{\lambda}(w^T\Phi(x_n) - t_n) \quad [1 \times 1] \\
&= \sum_{n=1}^{N} a_n \Phi(x_n)^T \\
w &= \left(\sum_{n=1}^{N} a_n \Phi(x_n)^T \right)^T \\
&= \sum_{n=1}^{N} (a_n \Phi(x_n)^T)^T \\
&= \sum_{n=1}^{N} (\Phi(x_n) a_n) \\
&\text{Define } \Phi = \begin{bmatrix} [\Phi^T(x_1)] \\ \vdots \\ [\Phi^T(x_N)] \end{bmatrix}, \text{ and } a = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \\
&= \Phi^T a \quad [M \times 1]
\end{aligned}
$$

Now we can substitute w in the loss function. To make the computation simpler, we switch to full matrix notation

$$
\begin{aligned}
L_w &= \frac{1}{2}(\Phi w - t)^T(\Phi w - t) + \frac{\lambda}{2} w^T w \\
&= \frac{1}{2}(\Phi\Phi^T a - t)^T(\Phi\Phi^T a - t) + \frac{\lambda}{2}(\Phi^T a)^T \Phi^T a \\
&= \frac{1}{2}(\Phi\Phi^T a - t)^T(\Phi\Phi^T a - t) + \frac{\lambda}{2} a^T \Phi\Phi^T a \\
&= \frac{1}{2}((\Phi\Phi^T a)^T - t^T)(\Phi\Phi^T a - t) + \frac{\lambda}{2} a^T \Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T - t^T)(\Phi\Phi^T a - t) + \frac{\lambda}{2} a^T \Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T\Phi\Phi^T a) + \frac{1}{2} t^T t - \frac{1}{2} a^T\Phi\Phi^T t - \frac{1}{2} t^T\Phi\Phi^T a + \frac{\lambda}{2} a^T\Phi\Phi^T a \\
&= \frac{1}{2}(a^T\Phi\Phi^T\Phi\Phi^T a) + \frac{1}{2} t^T t - a^T\Phi\Phi^T t + \frac{\lambda}{2} a^T\Phi\Phi^T a
\end{aligned}
$$

We can observe how Φ is present only when multiplied by its transpose. In this way we can use the kernels we have define before to substitute the features. In order to have a simpler notation, we can observe that the kernel function is a Gram matrix[26] $K = \Phi\Phi^T$ $[N \times N]$, where each element is

$$
\begin{aligned}
K_{nm} &= \Phi(x_n)^T\Phi(x_m) = k(x_n, x_m) \\
K &= \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{bmatrix}
\end{aligned} \tag{64}
$$

Notes

- Φ $[N \times M]$ and K $[N \times N]$
- K is a matrix of similarities of pairs of samples(metric)
- K is symmetric

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

$$L_a = \frac{1}{2}(a^T K K a) + \frac{1}{2}t^T t - a^T K t + \frac{\lambda}{2}a^T K a$$

Solving for a by combining $w = \Phi^T a$ and $a_n = -\frac{1}{\lambda}(w^T\Phi(x_n) - t_n)$

$$a = (K + \lambda I_N)^{-1} t \quad [N \times 1] \tag{65}$$

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 (5.1.2) 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 not features. It practically means that now we have to invert a $[N \times N]$ matrix instead of a $[M \times M]$.

When we make a prediction for a new x we have our linear regression model. If we substitute $w = \Phi^T a$.

$$\begin{aligned} y(x) &= w^T\Phi(x) \\ &= a^T\Phi\Phi(x) \\ &\text{Define } k(x) = \begin{bmatrix} k(x, x_1) \\ \vdots \\ k(x, x_N) \end{bmatrix} \\ &= k(x)^T(K + \lambda I_N)^{-1} t \end{aligned}$$

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

[26] Given N vectors, the Gram Matrix is the matrix of all inner products. A matrix by its transpose is always a Gram matrix

- 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 $[N \times N]$ matrix and not a $[M \times M]$ matrix. This is good when the number of features is very high
- 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

5.1.3 Kernel construction

In this section we will see how we can construct a kernel. If you are wondering how the heck we can avoid to compute Φ directly, even if $K = \Phi^T\Phi$, you are in the right section. So 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 Φ.

$$k(x, x') = \Phi(x)^T\Phi(x') = \sum_{i=1}^{M} \Phi_i(x)\Phi_i(x')$$

Where $\Phi(x)$ are basis functions.

It exist a second and much more interesting method to construct kernels. We can choose a kernel function that correspond to a scalar product in some space. We make an example to better understand what it means.

Example - Kernel construction Suppose to have the kernel function $k(x, z) = (x^T z)^2$. 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.

$$\begin{aligned} k(x, z) &= (x^T z)^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ &= \begin{bmatrix} x_1^2 & \sqrt{2}x_1 x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} z_1^2 & \sqrt{2}z_1 z_2 & z_2^2 \end{bmatrix}^T \\ &= \Phi(x)^T\Phi(z) \end{aligned}$$

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 $\begin{bmatrix} x_1^2 & \sqrt{2}x_1 x_2 & x_2^2 \end{bmatrix}$, 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). Computing directly the kernel is way cheaper than computing Φ. In this case we have

- **Naive kernel construction**: compute 6 features values and 9 multiplication 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 $k(x, z) = (x^T z + c)^p$, we can demonstrate that the feature expansion that represent this kernel includes all the possible monomial from degree 0 to p.

- **Naive kernel construction**: exponential grow in number of operation
- **Direct estimation of valid kernel**: linear grow in number of operation

We can represent features expansion that include billions of elements 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.

Now we can define more formally how we can demonstrate that a given kernel is valid. 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-definite[27]for all possible choices of the set $\{x_n\}$.

Theorem 5.1 (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

1. $k(x, x') = ck_1(x, x')$
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. $k(x, x') = k_a(x_a, x'_a)k_b(x_b, x'_b)$

[27]Positive semi-definite means that $x^T Kx \geq 0$, $\forall x : x_i \in \mathbb{R}^+$

Example - Gaussian kernel A commonly used homogeneous kernel is the Gaussian kernel.

$$k(x, x') = exp\bigg(- \frac{\|x - x'\|^2}{2\sigma^2} \bigg) \tag{66}$$

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

$$\|x - x'\|^2 = x^T x + x'^T x' - 2x^T x'$$

To give

$$\begin{aligned} k(x, x') &= exp(-\frac{1}{2\sigma^2} k_1(x, x')), \quad \text{where} \\ k_1(x, x') &= x^T x + x'^T x' - 2x^T x' \end{aligned}$$

We know that the exponential of a valid kernel is still valid(4) so we need to demonstrate that $-\frac{1}{2\sigma^2} k_1(x, x')$ is valid. For composition (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 (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(1), so they are valid. □

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

$$k(x, x') = exp\bigg(- \frac{1}{2\sigma^2} (k_i(x, x) + k_i(x', x') - 2k_i(x, x')) \bigg) \tag{67}$$

Object kernels We have said how kernels can be defined over real vector numbers but also object such as sets, graph, strings and text.

Example - Sets To define a simple metric over sets we can use

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|} \tag{68}$$

In practice, we find the number elements included in both A_1 and A_2. 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

$$k(x, x') = p(x)p(x') \tag{69}$$

p(x) represent a probability. 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".

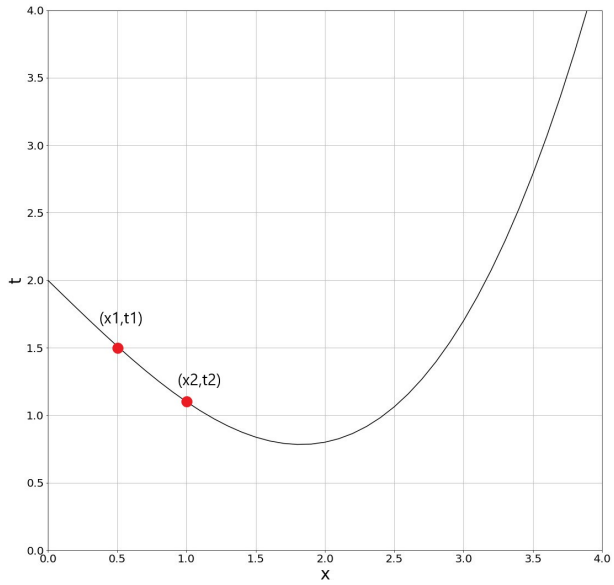
5.2 Gaussian processes

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 model for regression like ridge regression. We can extend the dual formulation to probabilistic discriminative models.

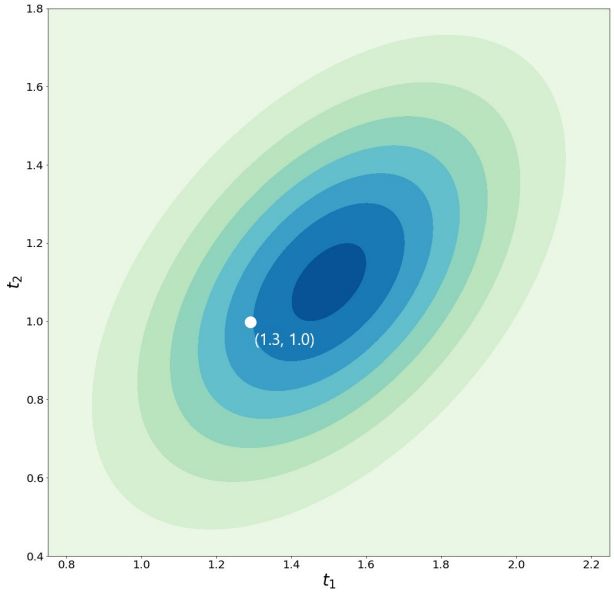
In Bayesian linear regression we have introduced a prior distribution over the parameters. Given a training set, we evaluate a new distribution over the parameters(posterior) by combining the prior and the new data observed(likelihood). From the posterior we can find a predictive distribution $p(t|x)$ for a new input x. With Gaussian processes, we define directly a prior distribution over functions[28]. 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 function 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.

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. 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.

[28]With function we indicate the model in the parametric world. So in linear regression it would be the hyperplane defined by the parameters

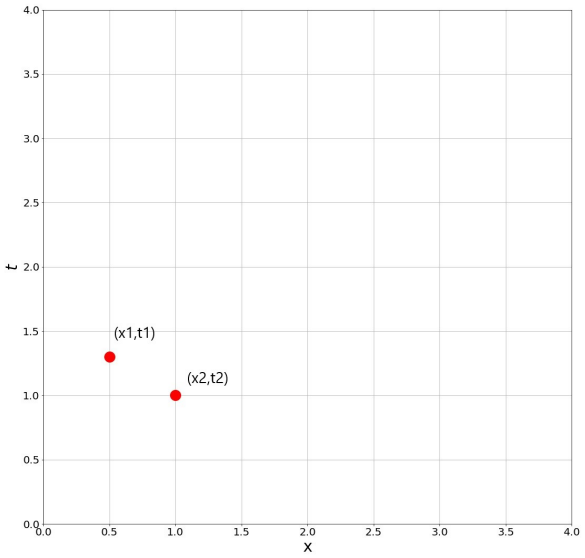


(a) Function we want to estimate



(b) Multivariate Gaussian over t

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)

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

$$w \sim \mathcal{N}(w|0, \tau I)$$

So how is y distributed?[29] We know that the linear combination of Gaussian is still Gaussian. So knowing that y is a linear combination of w, we are sure that it is distributed as a Gaussian. Now we can calculate its mean and variance.

$$\begin{aligned} E[y] &= \Phi E[w] = 0 \\ Cov[y] &= E[yy^T] = \Phi E[ww^T]\Phi^T = \tau \Phi \Phi^T = K, \text{ Gram matrix} \\ K_{nm} &= k(x_n, x_m) = \tau \Phi(x_n)^T \Phi(x_m) \end{aligned}$$

So we have

$$y \sim \mathcal{N}(y|0, K) \tag{70}$$

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 y. Now we can define a more formal and organized definition of Gaussian processes.

Definition 5.1 (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* $\{x_1, \ldots, x_N\}$*, jointly have a Gaussian distribution*

Being a Gaussian, the distribution can be completely specified by the mean and covariance.

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 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.

[29] As $p(w)$ is the prior for liner Bayesian regression, $p(y)$ is the prior for Gaussian processes

5.2.1 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

$$P(t_n|y_n) = \mathcal{N}(t_n|y_n(x_n), \sigma^2) \tag{71}$$

We can also assume that the noise is independent on each data point. So the joint distribution of t is still Gaussian

$$p(t|y) = \mathcal{N}(t|y, \sigma^2 I) \tag{72}$$

Since $p(y) = \mathcal{N}(0, K)$, we can compute the marginal distribution $p(t)$

$$p(t) = \int p(t|y)p(y)dy = \mathcal{N}(t|0, C), \text{where } C = K + \sigma^2 I_N \tag{73}$$

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 $p(t_{N+1}|t^{(N)}, x_1, \ldots, x_{N+1})$[30] We can calculate that

$$\begin{aligned} p(t^{(N+1)}) &= \mathcal{N}(t^{(N+1)}|0, C^{(N+1)}), \text{ where} \\ C^{(N+1)} &= \begin{bmatrix} C^{(N)} & k \\ k^T & c \end{bmatrix}, \quad [(N+1)x(N+1)] \\ k &= \begin{bmatrix} k(x_1, x_{N+1}) & \ldots & k(x_N, x_{N+1}) \end{bmatrix}^T, \quad [N \times 1] \\ c &= k(x_{N+1}, x_{N+1}), \quad [1 \times 1] \end{aligned}$$

We can observe that for the Gaussian distribution properties, the predictive distribution is still a Gaussian. From that, we can apply the properties over conditional Gaussian distribution to obtain $p(t_{N+1}|t^{(N)}, x_1, \ldots, x_{N+1})$.

$$p(t_{N+1}|t^{(N)}, x_1, \ldots, x_N + 1) \sim \mathcal{N}(t_{N+1}|k^T {C^{(N)}}^{-1} t, c - k^T {C^{(N)}}^{-1} k) \tag{74}$$

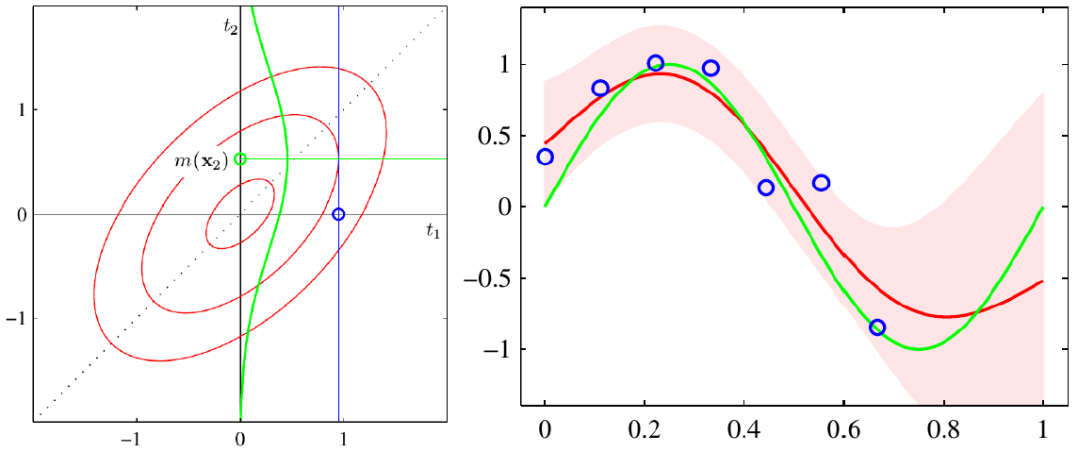
We can observe two things. The mean and the variance depend on x_{N+1}. More interestingly, the mean of the prediction is actually what we obtain for the kernel version of ridge regression. This shouldn't be a surprise, because we have already said that in the parametric word, ridge regression is a particular 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 $\sigma^2 I$, we are sure that C is positive definite and so it is for sure invertible.

[30] $t^{(N)}$ is the t vector when we have N sample. The professor in the lecture uses $\boldsymbol{t}_N$ 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

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 one 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. 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.

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) (red ellipses). We assume a zero mean distribution, 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. Now we observe in x_1 a value for t_1(blue dot). 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 $p(t_2|t_1, x_1, x_2)$. Now we can take the mean to estimate t_2(green point).

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. In some cases, the kernel have some hyperparameters. How can we find the optimal values for this hyperparameters? 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. Another approach uses the maximization of the marginal likelihood using gradient optimization. In practice, you want to find the hyperparameters for which the observed target variables are more likely. This is faster, but it can be stuck in local minima. Usually the gradient approach is the go to method for hyperparameters optimization.



6 Support Vector Machines

Support Vector Machines(SVM) is a kernel method that was very famous during the 90s. Is one of the most theoretical complete method in the machine learning world. Its theoretical guarantee are very good, and they reflect also in practice. The topic is very complex, so the goal of this chapter is to explain the most salient points. SVM 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 of training samples to reduce complexity and find sparser solutions. This subset is called **support vectors**
2. **Vector of weight $\boldsymbol{\alpha}$** This vector is used to weight the training samples subset.
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, the class prediction for a new example x_q is

$$f(x_q) = sign\Big(\sum_{m\in\mathcal{S}} \alpha_m t_m k(x_q, x_m) + b\Big), \text{ where} \tag{75}$$

$\mathcal{S}$ set of indices of the support vectors

α vector of weights

t target vector

$k(x_q, x_m)$ kernel

We can see that the solution formulation is somewhat similar to the one of the perceptron. In fact, usually SVMs are presented as their generalization. Now we will try to explain how we can derive this solution by revisiting the perceptron. We know that the prediction for perceptron is define as

$$f(x_q) = sign(w^T\Phi(x_q)) = sign\Big(\sum_{j=1}^{M} w_j\Phi_j(x_q)\Big)$$

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

$$w_j = \sum_{n=1}^{N} \alpha_n t_n \Phi_j(x_n)$$

Now we can put our new formulation of w_j into the perceptron function

$$f(x_q) = sign\Big(\sum_{j=1}^{M}\Big(\sum_{n=1}^{N}\alpha_n t_n \Phi_j(x_n)\Big)\Phi_j(x_q)\Big)$$
$$= sign\Big(\sum_{n=1}^{N}\alpha_n t_n (\Phi(x_n)\cdot\Phi(x_q))\Big) \tag{76}$$

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 as 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. Our prediction is now a weighted average over the target value and the similarity between the input and the training data samples. What we are doing is like a weighted kNN. 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.

To obtain a SVM from the perceptron we can replace the dot product with an arbitrary kernel $k(x, x')$. A very good property of SVM is that the usage of kernels ensure a convex weight optimization problem. In practice, we can always find the global optimum.

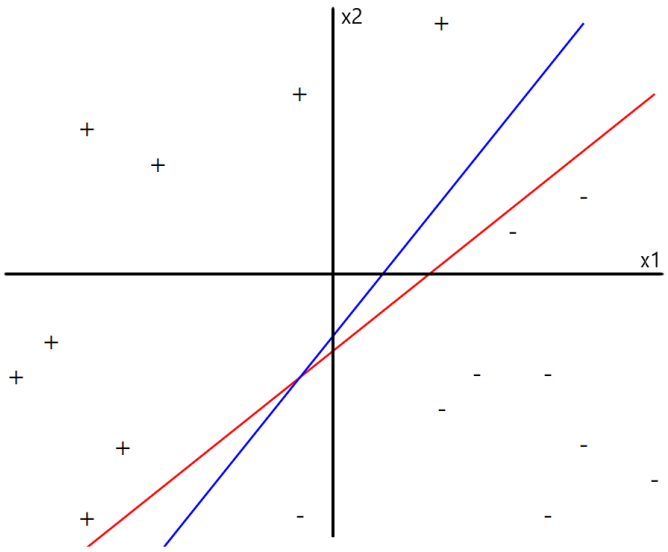
6.1 Learning phase

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
- **Training subset** We will see how this is a side effect of choosing the weights. In short, if a weight relative to a sample is zero, it will be excluded from the subset
- **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. 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.

Definition 6.1 (Margin)**.** *The margin is the minimum distance between the decision boundary hyperplane and the nearest point.*



2-dimensional input space with two classes (+) and (-).
Both red and blue line are linear separators of the two classes

In formulas we have

$$margin = \min_{n}(t_n(w^T\Phi(x_n) + b)) \tag{77}$$

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.
To find the optimal weights we need to maximize the margin.

$$w^* = \underset{w,b}{argmax}\left(\frac{1}{\|w\|_2}\min_{n}(t_n(w^T\Phi(x_n) + b))\right)$$

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 hyperplane(decision boundary) can be expressed in an infinite number of ways[31]. We can solve this by fixing the scale of the parameters. 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 $margin \geq 1$.

[31]For example $3x_1 + \frac{1}{2}x_2 = c$ is the same hyperplane as $6x_1 + x_2 = 2c$

From this considerations we can formulate the new problem. First, we can eliminate the margin from $w*$.

$$w^* = max\left(\frac{1}{\|w\|_2}\right),$$
$$\text{where } t_n(w^T\Phi(x_n) + b) \geq 1$$

For notations purposes, we switch from a maximization problem to a minimization one. We also slightly modify it to have simpler calculi later on

$$w^* = min\left(\frac{1}{2}\|w\|_2^2)\right)$$

Finally the new problem is

$$\begin{aligned} \textbf{Minimize} \quad & \frac{1}{2}\|w\|_2^2 \\ \textbf{Subject to} \quad & t_n(w^T\Phi(x_n) + b) \geq 1, \quad \forall n \end{aligned}$$

Note - Constraint optimization basics Suppose we have

$$\begin{aligned} \textbf{Minimize} \quad & f(w) \\ \textbf{Subject to} \quad & h_i(w), \quad i = 1, 2, \dots \end{aligned}$$

Where $f(w)$ is a convex function. If f and h_i are linear we can use linear programming, but in our case the function to minimize is actually quadratic. 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

$$L(w, \lambda) = f(w) + \sum_i \lambda_i h_i(w) \tag{78}$$

The λ_is 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

$$\nabla L(w, \lambda) = 0$$

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

$$\begin{aligned} \textbf{Minimize} \quad & \frac{1}{2}(w_1^2 + w_2^2) \\ \textbf{Subject to} \quad & w_1 + w_2 = 1 \end{aligned}$$

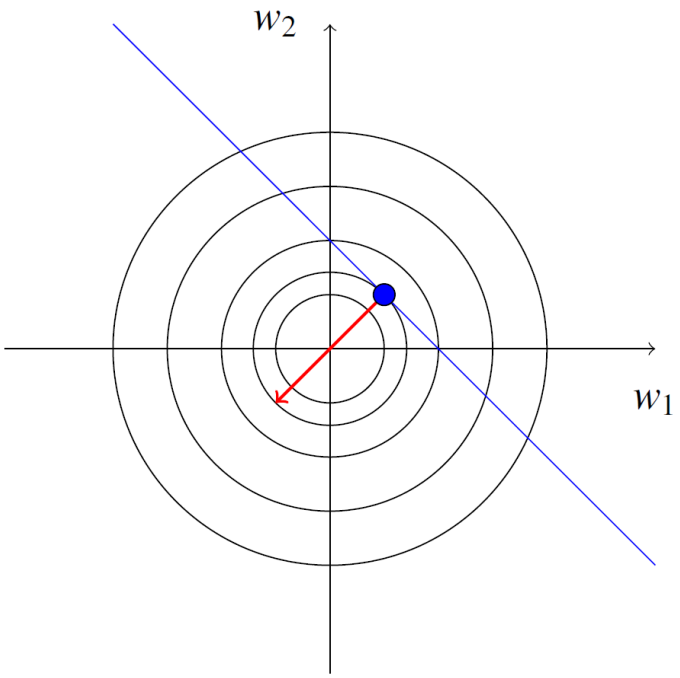
The Lagrangian function would be

$$L(w, \lambda) = \frac{1}{2}(w_1^2 + w_2^2) + \lambda(w_1 + w_2 - 1)$$

So we find the gradient of $L(w^*, \lambda^*) = 0$

$$\nabla L(w, \lambda) = 0 \rightarrow \begin{cases} \frac{\partial L}{\partial w_1} = w_1 + \lambda = 0 \\ \frac{\partial L}{\partial w_2} = w_2 + \lambda = 0 \\ \frac{\partial L}{\partial \lambda} = w_1 + w_2 - 1 = 0 \end{cases}$$

$$\begin{cases} w_1 = \frac{1}{2} \\ w_2 = \frac{1}{2} \\ \lambda = -\frac{1}{2} \end{cases}$$



Black line are the isoline of our objective function.
The blue line is our constraint.
The red arrow is the gradient of the objective function

We can notice from the figure above, that the gradient of the objective function is orthogonal to the constraint. This 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.
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

$$\begin{aligned} \textbf{Minimize} \quad & f(w) \\ \textbf{Subject to} \quad & g_i(w) \leq 0, \quad i = 1, 2, \dots \\ & h_i(w) = 0, \quad i = 1, 2, \dots \end{aligned}$$

We define a Lagrange multiplier α_i for the inequalities. The Lagrangian function becomes:

$$L(\mathbf{w}, \alpha, \lambda) = f(\mathbf{w}) + \sum_i \alpha_i g_i(\mathbf{w}) + \sum_i \lambda_i h_i(\mathbf{w})$$

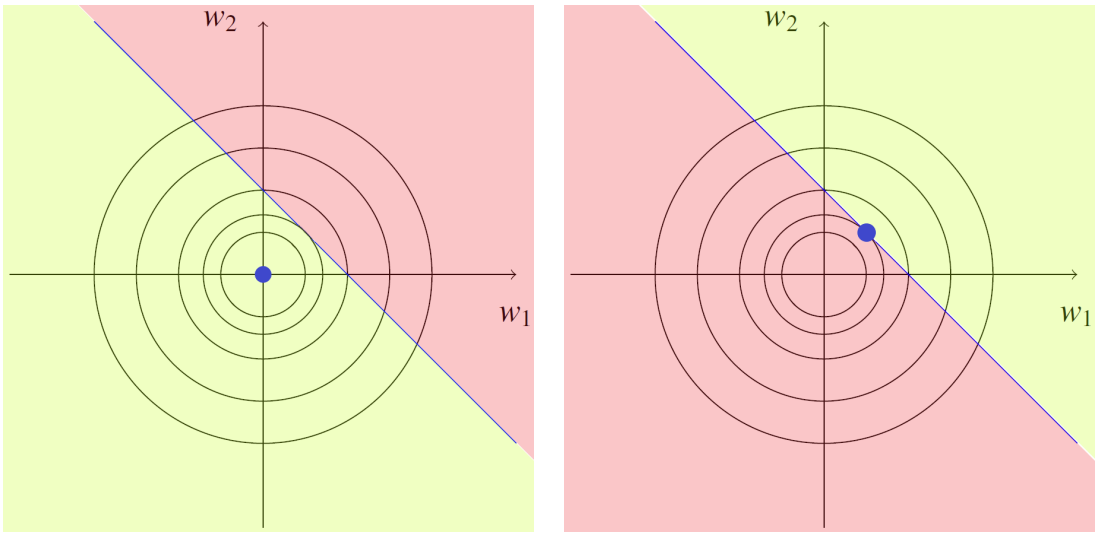
To find the optimal solution we can exploit the **KKT conditions**[32] (necessary conditions):

$$\begin{aligned} \nabla L(w^*, \alpha^*, \lambda^*) &= 0 \\ h_i(w^*) &= 0 \\ g_i(w^*) &\leq 0 \\ \alpha_i &\geq 0 \\ \alpha_i^* g_i(w^*) &= 0 \end{aligned}$$

The most interesting constraint is the last one and it is called **complementarity condition**. We can have two cases

1. $\alpha_i^* = 0 \wedge g_i(w^*) \leq 0$. If $\alpha_i^* = 0$, $g_i(w^*)$ can assume infinite different values
2. $\alpha_i^* \geq 0 \wedge g_i(w^*) = 0$. If $g_i(w^*) = 0$, α_i^* can assume infinite different values

Let's use the previous example to see the two cases



(a) Case 1 (b) Case 2

[32] KKT stands for Karush-Kuhn-Tucker conditions, which are necessary conditions for optimization problems with inequality constraints.

In the first case we have

$$\begin{aligned} \textbf{Minimize} \quad & \frac{1}{2}(w_1^2 + w_2^2) = \frac{1}{2}\|w\|_2 \\ \textbf{Subject to} \quad & w_1 + w_2 \leq 1 \end{aligned}$$

We can see from the picture (a) 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.

In the second case we have

$$\begin{aligned} \textbf{Minimize} \quad & \frac{1}{2}(w_1^2 + w_2^2) = \frac{1}{2}\|w\|_2 \\ \textbf{Subject to} \quad & w_1 + w_2 \geq 1 \end{aligned}$$

Here the solution is no longer in the global optimum, but lies on the constraint. In fact, we have that $g(w^*) = 0$ and $\alpha \geq 0$.

When the solution lies on the constraint, it is called active constraint. When a constraint is active its Lagrange multiplier is positive. On the other hand, if the solution is inside the region defined by the constraint, its Lagrange multiplier will be 0.

6.2 Dual representation

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. 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.

$$\begin{aligned} & \textbf{Primal} \\ \textbf{Minimize} \quad & \frac{1}{2}\|w\|_2^2 \\ \textbf{Subject to} \quad & t_n(w^T\Phi(x_n) + b) \geq 1, \quad \forall n \end{aligned}$$

Now our objective is to find its dual kernel representation. Let's consider the Lagrangian function of our primal problem[33]

$$L(w, b, \alpha) = \frac{1}{2}\|w\|_2^2 - \sum_{n=1}^{N} \alpha_n(t_n(w^t\Phi(x_n) + b) - 1) \tag{79}$$

[33]Here we have a minus sign. The sign of the summation doesn't affect the solution of the Lagrangian function

As we did before, we put the gradient of L equal to zero, with respect to w and b [34]. For w we have,

$$\frac{\partial L(w,b,\alpha)}{\partial w} = \frac{1}{2}2w - \sum_{n=1}^{N} \alpha_n t_n \Phi(x_n) = 0$$

$$w = \sum_{n=1}^{N} \alpha_n t_n \Phi(x_n)$$

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 $L(w,b,\alpha)$

Dual

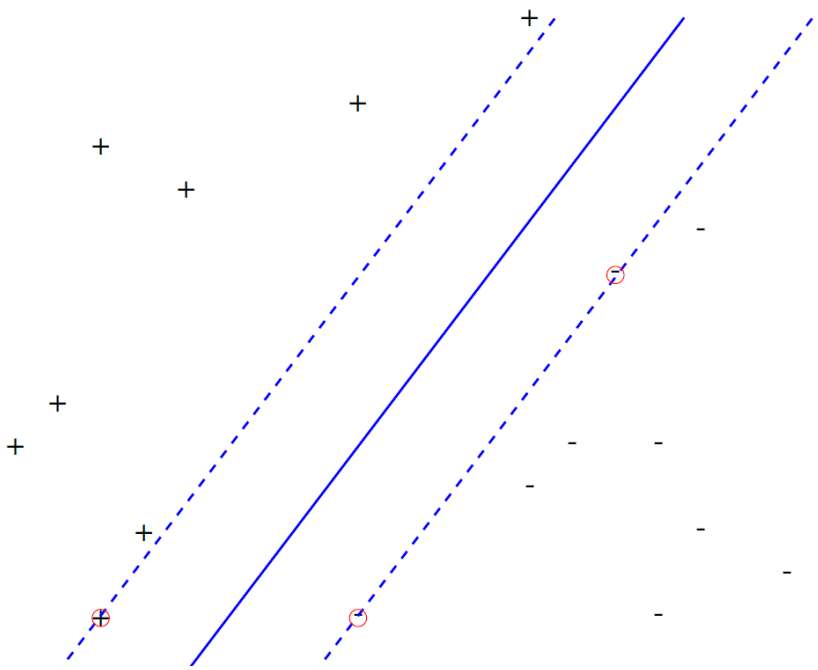
$$\begin{aligned} \textbf{Maximize} \quad & \tilde{L}(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(x_n, x_m) \\ \textbf{Subject to} \quad & \alpha_n \geq 0, \quad \text{for } n = 1, \ldots, N \\ & \sum_{n=1}^{N} \alpha_n t_n = 0 \end{aligned}$$

In this new formulation we don't have neither parameters nor features. It is an instance based formulation.

6.3 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. 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.

[34] $\frac{\partial \|w\|_2^2}{\partial w} = \frac{\partial \sum_{n=1}^{N} w_n^2}{\partial w} \rightarrow \frac{\partial}{\partial w_j} \sum_{k=1}^{n} w_k^2 = \sum_{k=1}^{n} \underbrace{\frac{\partial}{\partial w_j} w_k^2}_{\substack{=0, \text{ if } j \neq k, \\ =2w_j, \text{ else}}} = 2w_j$. It follows that $\frac{\partial \|w\|_2^2}{\partial w} = 2w$



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 all the support vector 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.
The classification of new points is performed as

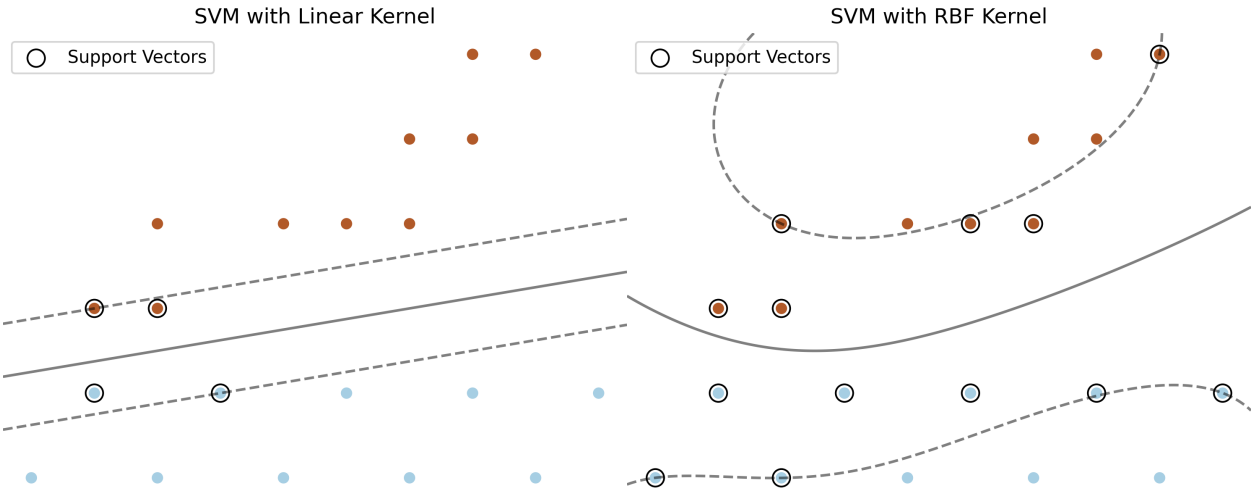
$$y(x) = sign\left(\sum_{n=1}^{N} \alpha_n t_n k(x, x_n) + b\right) \tag{80}$$

This is very similar to the perceptron case. Be aware, that we don't have a fast way to find the optimal value for b. We can estimate it with

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left(t_n - \sum_{m \in \mathcal{S}} \alpha_m t_m k(x, x_m)\right)$$

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. This leads to poor generalization and to some scalability issue. In this example we see the difference between a linear and a gaussian kernel (RBF). The Gaussian kernel can easily overfit the data, the support vectors are too many, while the linear kernel has a few and is more robust to noise and outliers: We have seen how we are able to construct bounds about



(a) Linear kernel has 4 support vectors (b) Gaussian kernel has 13 support vectors

the loss function of our models. For SVMs exists a very handy way to construct such bound. It's 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 case we will not misclassify any point. In the worst case, when we "leave out" a support vector, we can misclassify a point. The loss function will be upper bounded by the probability of misclassify a point, which in our case is [35]

$$L_h \leq \frac{E[|\mathcal{S}|]}{N} \tag{81}$$

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. 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 can solve the quadratic problem of finding the weights α. There are more efficient way to calculate the weights. For example **SMO (Sequential Minimal Optimization)**. Instead of calculating all the weights at the same time, we can find them in couples[36]. We can apply the following iteratively until convergence,

[35] $|\mathcal{S}|$ is the number of support vectors
[36] Note that if we calculate one α_i at the time we would violate the constraints

1. Find example x_i that violates KKT conditions
2. Select second example x_j heuristically
3. Jointly optimize α_i and α_j

6.4 Noisy data

So far we have assumed that the data are always linearly separable. 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. Now we have to reformulate our primal problem to include the slack variables.

$$
\begin{aligned}
& \textbf{Primal} \\
\textbf{Minimize} \quad & \frac{1}{2}\|w\|_2^2 + C\sum_i \xi_i \\
\textbf{Subject to} \quad & t_i(w^T\Phi(x_i) + b) \geq 1 - \xi_i, \quad \forall i \\
& \xi_i \geq 0, \quad \forall i
\end{aligned}
$$

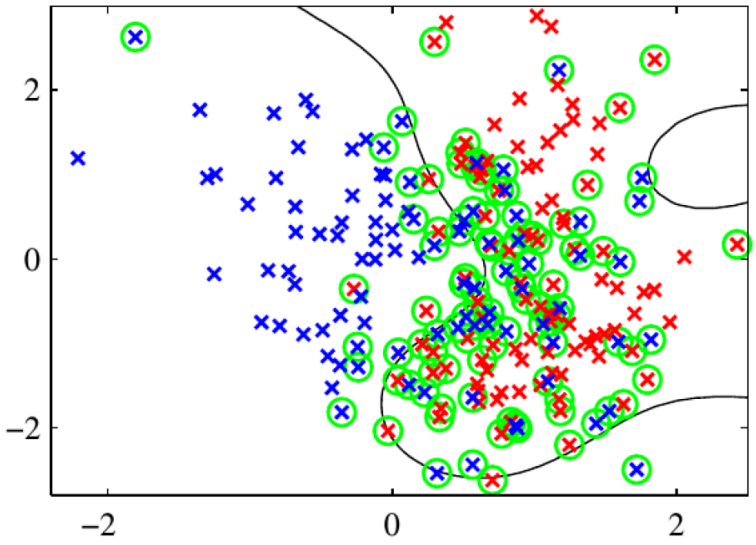
In practice ξ_i shift the margin relative to a sample x_i in order to respect the constraint. But ξ_i also appears in the minimization formula, in order to limit the exploitation of the margin shift. The term C is a coefficient that influences how much the slack variables will penalize the objective function. If we put C close to zero, we are probably making a lot of mistake, because the misclassification 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. The risk is that no solution can be found, because the problem is not linearly separable. In practice, we can control the bias-variance tradeoff by manipulating this coefficient. We can find the value of C with cross-validation. As we did before we can find the dual representation

$$
\begin{aligned}
& \textbf{Dual} \\
\textbf{Maximize} \quad & \tilde{L}(\alpha) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m t_n t_m k(x_n, x_m) \\
\textbf{Subject to} \quad & 0 \leq \alpha_n \leq C, \quad \text{for } n = 1, \ldots, N \\
& \sum_{n=1}^{N}\alpha_n t_n = 0
\end{aligned}
$$

We can see how C has become the upper bound of α_n. 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 lies on the margin($\xi_i = 0$)
- $\alpha_n = C$ the point lies inside the margin, and it can be either correctly classified($0 < \xi_i \leq 1$) or misclassified($\xi_i > 1$)[37].

[37]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



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 - SVM uses So far, we have seen the SVMs applied to classification problems. We can also use them for

- Regression
- Ranking
- Feature selection
- Clustering
- Semi-supervised learning

7 Markov Decision Processes

7.1 Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforce. 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 object 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 counterreplies—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. At each step t the agents can execute an **action** a_t. As a result the environment will provide an **observation** o_t about its state and a **reward** r_t to the agent. The goal of the agent is to maximize the reward. 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.

$$h_t = a_1, o_1, r_1, \ldots, a_t, o_t, r_t$$

The history influences what will happen next, because future choices, and so rewards, are dependent on the past decisions. 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

$$s_t = f(h_t)$$

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,

- $h_2^1 = (\underbrace{\text{do nothing}}_{a_1}, \underbrace{\text{double flash}}_{o_1}, \underbrace{\text{null}}_{r_1}$, pull lever, bell rings, electroshock)
- h_2^2 =(do nothing, bell rings + single flash, null, double lever, null, lasagna)
- h_2^3 =(pull lever, single flash, null, pull lever, bell rings, ?)

If you were Bisc, what would you expect? The answer changes based on how we define our state. 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.

We can distinguish between two states. The **environment state** s_t^e 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. The non-observability of the real state greatly increases the difficulty of the problem. The **agent state** s_t^a is the agent internal representation. 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($s_t^a = f(h_t)$). 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, $o_t = s_t^e$. 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**. 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. 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. Another advantage of RL compared to classical control theory is the ability of generating approximate, but yet efficient, solutions. 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.

7.2 MDP

7.2.1 MDP model

In this chapter 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 7.1 (Markov assumption)**.** *A stochastic process* X_t[38] *is said to be Markovian if and only if*

$$P(X_{t+1}|X_t = k_t, X_{t-1} = k_{t-1}, \dots, X_0 = k_0) = P(X_{t+1}|X_t = k_t)$$

This means that the future is independent of the past given the present.

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 an so the next moves. 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.

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. The MDPs we will analyze in this chapter will be **stationary**[39]. It means that the the environment doesn't change through time, so the problem is time invariant.
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.

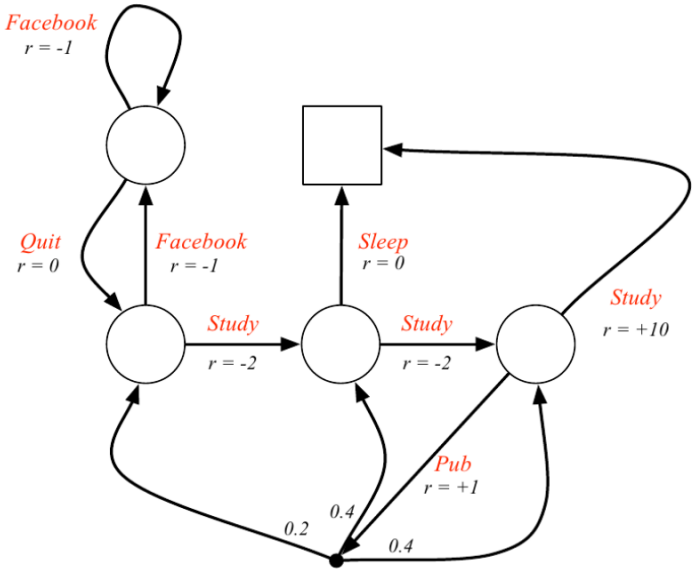
[38] A stochastic or random process can be defined as a collection of random variables that, in our case, is indexed by time. We can see it as a sequential observation of a time series or model

[39] In a more formal way we have $p_{ij} = P(X_{t+1} = j|X_t = i) = P(X_1 = j|X_0 = i)$

Definition 7.2 (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* $R(s, a)$. *The reward in performing action* a *in* s
- γ, *discount factor. Weights how important are future rewards.* $\gamma \in [0, 1]$
- μ, *set of initial probability. Describe the probability for every state to be the starting state*

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

We can represent a MDP with a directed graph. We can see how taking an action sometimes have deterministic consequences(Facebook), or non-deterministic consequences(Pub).

A very important phase of every reinforcement problem is goal definition. When we use MDPs the goal can be encoded in the reward definitions. In particular we have

Definition 7.3 (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, 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. From a practical point of view, a goal can be defined with many different reward functions. 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 frequently the agent receive feedback and on how much the reward function is shaped[40].

7.2.2 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.
  - **Finite** We perform a finite and fixed number of steps
  - **Indefinite** We don't know how many steps will take to reach the goal. We stop when a stopping criteria is met
  - **Infinite** We will perform an infinite number of steps
- **Cumulative reward** Define how we aggregate all the rewards from each step.
  - **Total reward** $V = \sum_{i=1}^{\infty} r_i$
  - **Average reward** $V = \lim_{n\to\infty} \frac{r_1+\cdots+r_n}{n}$
  - **Discounted reward** $V = \sum_{i=1}^{\infty} \gamma^{i-1} r_i$
  - **Mean-variance reward**

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 7.4. *The return v_t is the total discounted reward from time–step t.*

$$v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{82}$$

[40]Shaped means that the reward are different from zero. Shaped reward functions lead to faster learning

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, on the other side if γ is close to 1 we will equally evaluate immediate and future rewards. 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 will go on, so that we will play another 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 represents animal/human behavior preference for immediate reward.

7.2.3 Policies

So far we have define 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. 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
- 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

From now on, we will focus on stationary stochastic Markovian policies. More formally we have

Definition 7.5 (Stationary Stochastic Markovian policies)**.** *A policy π is a distribution of actions over the state*

$$\pi(a|s) = P(a|s)$$

In few words, π describes the probability of doing action a in s.

7.2.4 Value functions

Given a policy π, 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 policy. The utility can be formalized as

Definition 7.6 (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*

$$V^\pi(s) = E_\pi[v_t|s_t = s]$$

For control purposes, rather than the value of each state, it is easier to consider the value of each action in each state

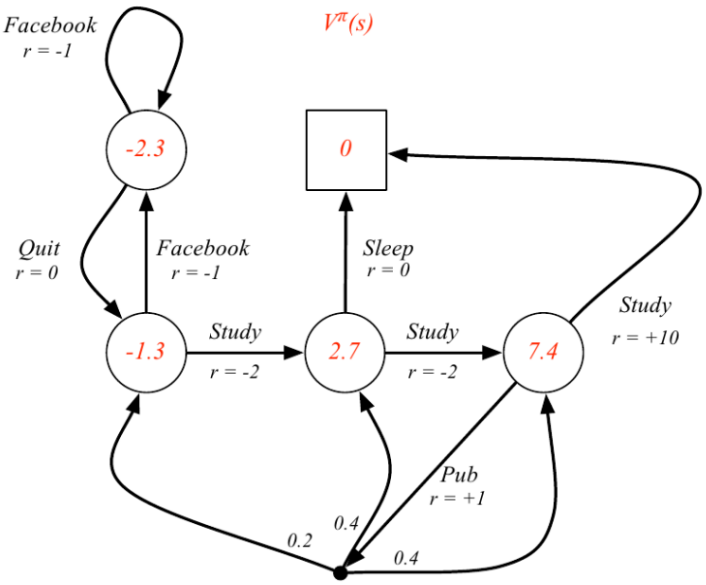
Definition 7.7 (Action-Value function)**.** *The action–value function $Q^\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π*

$$Q^\pi(s, a) = E_\pi[v_t|s_t = s, a_t = a]$$

It's worth mentioning that we can calculate $V^\pi(s)$ from $Q^\pi(s, a)$, but not vice versa. We have that

$$V^\pi(s) = \sum_a \pi(a|s)Q^\pi(s, a) \tag{83}$$

Practically, we are doing a marginalization[41]. 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 $\pi(s, a) = \frac{1}{2}$ [42] and a discount factor $\gamma = 1$.



[41] Given 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. $P(x) = \sum_y p(y)P(x, y)$

[42] $\pi(s, a) = \frac{1}{2}$ because from every state we always have to choose between two actions

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 value of successor state,

$$\begin{aligned} V^\pi(s) &= E_\pi[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s] \\ &= \sum_{a\in A} \pi(a|s)(R(s,a) + \gamma \sum_{s'\in S} P(s'|s,a)V^\pi(s')) \end{aligned}$$

$R(s,a)$ 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. We can see that the Bellman equation is recursive, because $V^\pi(s)$ appears both on the left and right side of the equation[43]. 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.

$$\begin{aligned} R^\pi(s) &= \sum_{a\in A} \pi(a|s)R(s,a) \\ P^\pi(s'|s) &= \sum_{a\in A} \pi(a|s)P(s'|s,a) \end{aligned}$$

$R^\pi(s)$ 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

$$V^\pi(s) = R^\pi(s) + \gamma \sum_{s'\in S} P^\pi(s'|s)V^\pi(s')$$

For every term of the Bellman equation, we can define its matrix version

$$\begin{aligned} V^\pi &= \begin{bmatrix} V^\pi(s_1) & \dots & V^\pi(s_{|S|}) \end{bmatrix}^T \quad [|S|\text{x}1] \\ R^\pi &= \begin{bmatrix} R^\pi(s_1) & \dots & R^\pi(s_{|S|}) \end{bmatrix}^T \quad [|S|\text{x}1] \\ P^\pi &= \begin{bmatrix} P^\pi(s_1,s_1) & \dots & P^\pi(s_1,s_{|S|}) \\ \vdots & \ddots & \vdots \\ P^\pi(s_{|S|},s_1) & \dots & P^\pi(s_{|S|},s_{|S|}) \end{bmatrix} \quad [|S|\text{x}|S|] \end{aligned}$$

In matrix notation we can rewrite the system of Bellman equation as

$$V^\pi = R^\pi + \gamma P^\pi V^\pi \tag{84}$$

[43] 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

To solve the system we can isolate V^π

$$\begin{aligned} V^\pi &= R^\pi + \gamma P^\pi V^\pi \\ V^\pi - \gamma P^\pi V^\pi &= R^\pi \\ V^\pi &= (I - \gamma P^\pi)^{-1} R^\pi \end{aligned} \tag{85}$$

This is our solution to the policy evaluation problem. Given a policy π we can evaluate the value of every state.

Note - 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 $\gamma < 1$. 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 of going from a fixed state s to every state in S. The stochasticity of the matrix ensures that its eigen values are between -1 and 1. If we define $A = I - \gamma P^\pi$, the eigen values 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.

As we did before, we can define the Bellman equation using the action-value function

$$\begin{aligned} Q^\pi(s,a) &= E_\pi[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \\ &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \\ &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in A} \pi(a'|s') Q^\pi(s',a') \end{aligned}$$

The expression of V^π can be compacted even more using the **Bellman operator**. 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 7.8. *The Bellman operator T^π for V^π is defined as $T^\pi : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$. This operator takes as an argument a value function and it returns a value function* [44]

$$T^\pi(V) = R^\pi + \gamma P^\pi V$$

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. We can demonstrate that

$$T^\pi(V^\pi) = V^\pi \tag{86}$$

We can say that V^π is the only fixed point[45] of the operator T^π. This implies that if we

[44] The 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

[45] Fixed point it means that, if we apply the operator to the input, the result will be the same as the input

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. Formally we have

$$\lim_{k\to\infty} (T^\pi)^k V = V^\pi, \quad \forall V \tag{87}$$

This is very useful when we have a large number of states because the complexity is dropped from $\mathcal{O}(|S|^3)$ to $\mathcal{O}(\#iteration|S|^2)$. 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 (85) produces an exact solution, on the other hand (87) produces an approximation.
We have seen how the value function "measures" the value of each state. The values are influenced by the policy we have chosen. We would like to find the policy which leads to the maximum possible value in each state. This value function is called **optimal value function**.

Definition 7.9 (Optimal state-value function)**.** *The optimal state–value function $V^*(s)$ is the maximum value function over all policies*

$$V^*(s) = \max_\pi V^\pi(s) \tag{88}$$

Definition 7.10 (Optimal action-value function)**.** *The optimal action–value function $Q^*(s,a)$ is the maximum value function over all policies*

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{89}$$

We can say that we have solved the MDP when we have found the optimal value function. From the value functions, we can define a partial ordering over policies. It means that we can evaluate how good a policy is compared to another one, by looking at the relative value functions ($\pi \geq \pi'$ if $V^\pi(s) \geq V^{\pi'}(s)$, $\forall s \in S$). The value function and policies in a MDP have very special properties

Theorem 7.1 (MDP optimality)**.** *For any Markov Decision Process*

1. *There exists an optimal policy π^* that is better than or equal to all other policies, $\pi^* \geq \pi$, $\forall \pi$*
2. *All optimal policies achieve the optimal state-value function, $V^{\pi^*}(s) = V^*(s)$, $\forall s \in S$*
3. *All optimal policies achieve the optimal action-value function, $Q^{\pi^*}(s,a) = Q^*(s,a)$, $\forall s \in S$, $\forall a \in A$.*
4. *There is always a deterministic optimal policy for any MDP*

The most interesting property is the last one. It means that they may exist multiple optimal policies. Between them, surely exists at least one which is stationary, deterministic, and Markovian. A deterministic policy can be found by maximizing over $Q^*(s,a)$

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \underset{a \in A}{argmax}\big(Q^*(s,a)\big) \\ 0, & \text{otherwise} \end{cases} \tag{90}$$

For every state, we give probability 1 to the action a that leads to the best action-value function in the given state s. This naive approach is unfeasible, because to find the optimal policy we would need to check a number of policies which is exponential in the number of states. To find the optimal policy, we can use the optimal value function. To find it we can use the **Bellman optimality equation**.

Definition 7.11 (Bellman optimality equation for V^*)**.**

$$\begin{aligned} V^*(s) &= \max_a \{Q^*(s,a)\} \\ &= \max_a \left\{ R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right\} \end{aligned} \tag{91}$$

Definition 7.12 (Bellman optimality equation for Q^*)**.**

$$\begin{aligned} Q^*(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \\ &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a'} Q^*(s',a') \end{aligned} \tag{92}$$

If we compare the Bellman operator to the Bellman optimality equation we can notice that the summation over all the action has been substituted with a max operator. This implies that our equation is no longer linear and so no closed form solution exists. As we did before, we can use the Bellman operator to create an iterative method for computing V^*. this new operator is called **Bellman optimality operator**

Definition 7.13 (Bellman optimality operator for V^*)**.** *The Bellman operator T^* for V^* is defined as $T^* : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$*[46]*.*

$$(T^* V^*)(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right) \tag{93}$$

Definition 7.14 (Bellman optimality operator for Q^*)**.** *The Bellman operator T^* for Q^* is defined as $T^* : \mathbb{R}^{|S|x|A|} \to \mathbb{R}^{|S|x|A|}$.*

$$(T^* Q^*)(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a'} Q^*(s',a') \tag{94}$$

[46] $(T^*V^*)(s)$ means that we apply T^* to $V^*(s)$. We can't use the previous matrix notation because equation is no longer linear

The Bellman optimality operator have the same properties as the Bellman operator. If applied to a starting value function, it will produce an increasingly precise approximation of the true optimal value function.

Note - Bellman operators properties Here we recap the Bellman operators properties.

- **Monotonicity** If $V_1 < V_2$ component-wise.

$$T^\pi(V_1) \leq T^\pi(V_2)$$
$$T^*(V_1) \leq T^*(V_2)$$

- **Max-Norm contraction** For two value functions V_1 and V_2 [47]

$$\|T^\pi(V_1) - T^\pi(V_2)\|_\infty \leq \gamma \|V_1 - V_2\|_\infty$$
$$\|T^*(V_1) - T^*(V_2)\|_\infty \leq \gamma \|V_1 - V_2\|_\infty$$

- **Fixed point uniqueness** V^π is the unique fixed point of T^π. V^* is the unique fixed point of T^*

$$T^\pi(V^\pi) = V^\pi$$
$$T^*(V^*) = V^*$$

- **True value function convergence** For any value function V and any policy π

$$\lim_{k\to\infty} (T^\pi)^k V = V^\pi$$
$$\lim_{k\to\infty} (T^*)^k V = V^*$$

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 $\epsilon = \|T^*(V) - V\|_\infty$. We can bound the distance of our approximation to the real value as

$$\|V^* - V\|_\infty \leq \frac{2\gamma\epsilon}{1-\gamma} \tag{95}$$

7.3 Dynamic programming

To find the optimal policy we have seen that the simple approach of enumerating all possible policies is unfeasible. In fact, we would need to evaluate $|A|^{|S|}$. We can use dynamic programming to find a better way to fond π^*. Dynamic Programming is a very general solution method for problems which have two properties

[47] $\|V_1 - V_2\|_\infty = \max_s |V_1(s) - V_2(s)|$

- **Optimal substructure** Optimal solution can be decomposed into subproblems and the principle of optimality applies. A problem is said to satisfy the principle of optimality, if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.
- **Overlapping subproblems** Subproblems recur many times and solutions can be cached and reused

Markov decision processes satisfy both properties. Bellman equation gives recursive decomposition. Value function stores and reuses solutions.

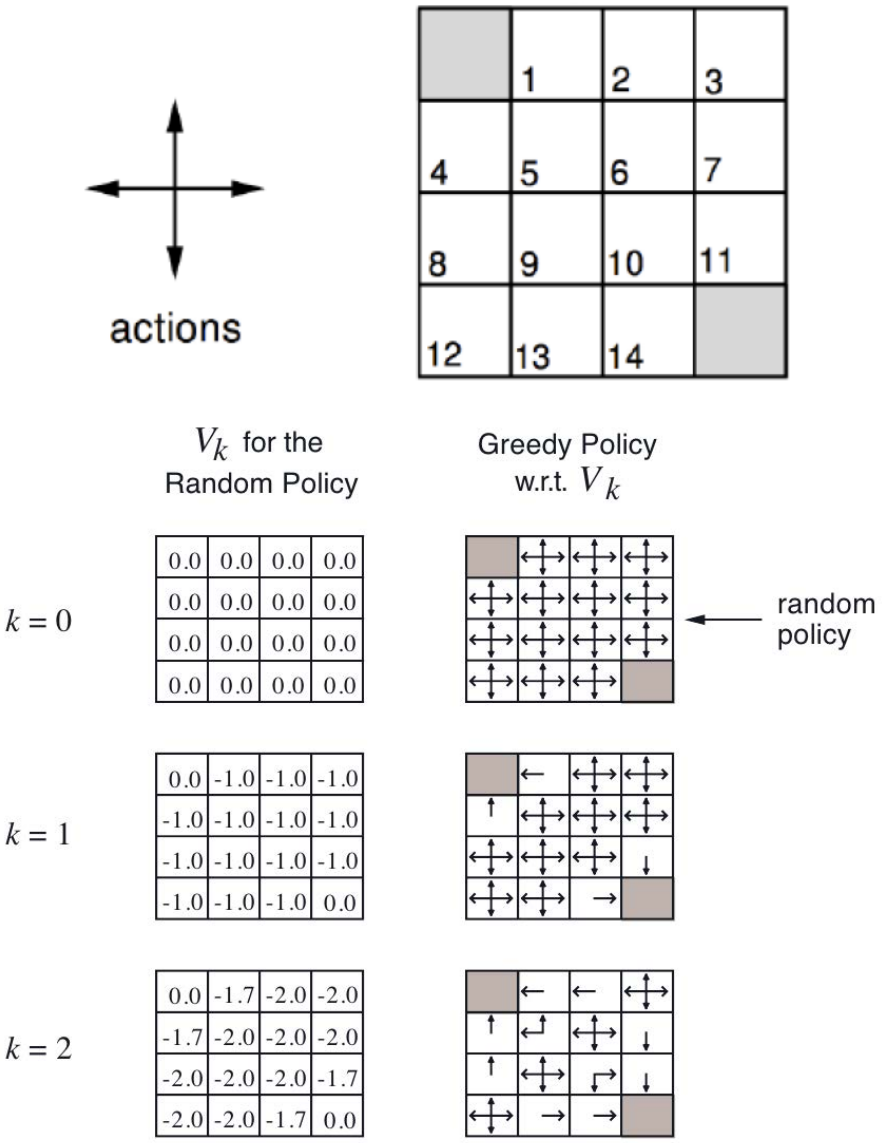
7.3.1 Policy iteration

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 of a policy. We have already seen how we can estimate it. In closed form with (85), or with an iterative approach (87).
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. The action will have deterministic results. The top left and bottom right state are absorbing states(goals). If we reach these states the episode is concluded. The immediate reward is -1 until an absorbing state is reached. We use $\gamma = 1$.



Notice that here we are doing only policy evaluation. We start 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. We apply the Bellman operator to find an approximation of V^{π_0}. To update the state-value function in a state we use

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(a|s) \Big[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s') \Big]$$

To declutter 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 $\pi^0(a|5) = \frac{1}{4}$

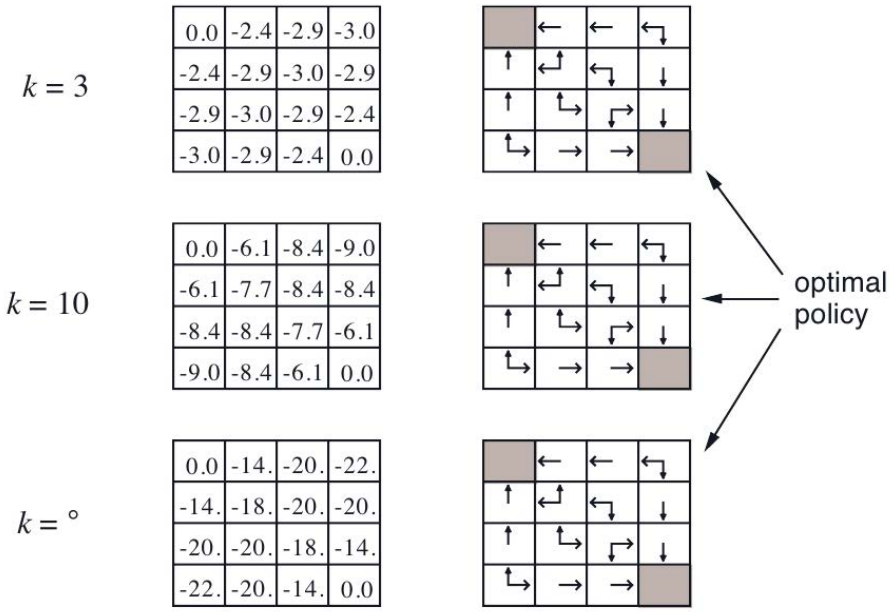
$$V_1(5) = \sum_{a \in A} \frac{1}{4} \Big[-1 + \underbrace{\gamma \sum_{s' \in S} P(s'|s,a) V_0(s')}_{V_0(s')=0} \Big] = -1$$

This holds true for every state, excluded the absorbing state where the state-value function is always zero. For $k = 2$ we have.

$$\begin{aligned} V_2(5) &= \sum_{a \in A} \pi(a|s) \Big[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_1(s') \Big] \\ &= \sum_{a \in A} \frac{1}{4} \Big[-1 + \sum_{s' \in S} P(s'|s,a)(-1) \Big] \\ &= \sum_{a \in A} \frac{1}{4} \Big[-1 - 1 \Big] \\ &= -2 \end{aligned}$$

Applying the Bellman operator on the random policy we are simply averaging the state-value function of the neighbours and adding the immediate reward.

$$V_2(1) = -1 + \frac{1}{4}(0 - 1 - 1 - 1) = -1.75$$



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. 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.

As we have seen in the example above, once we have found V^{π_0} we can calculate a new policy π_1. We can improve the previous policy by acting greedily in every state

$$\pi_1(s) = \underset{a \in A}{argmax} \Big\{ Q^{\pi_0}(s, a) \Big\}$$

This improves the value from any state s over one step

$$Q^{\pi_0}(s, \pi^1(s)) \geq Q^{\pi_0}(s, \pi^0(s)) = V^{\pi_0}$$

Theorem 7.2 (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.

$$\begin{aligned} V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) = E_{\pi'}[r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s] \\ &\leq E_{\pi'}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) | s_t = s] \\ &\leq E_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 Q^{\pi}(s_{t+2}, \pi'(s_{t+2})) | s_t = s] \\ &\leq E_{\pi'}[r_{t+1} + \gamma r_{t+2} + \ldots | s_t = s] = V^{\pi'}(s) \end{aligned}$$

□

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^*.

Generalized policy iteration 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. Generalized policy iteration simplify the policy evaluation stopping the estimation step earlier, in order to reduce the amount of computation needed.

7.3.2 Value iteration

Dynamic programming gave us another methodology to find the optimal policy of an MDP. 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 ($\lim_{k\to\infty}(T^*)^k V = V^*$). 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 (95).
Once we have obtained a $V_k \approx V^*$, we can find the relative greedy policy to estimate the true optimal policy

$$\tilde{\pi}^*(s) = \underset{a\in A}{argmax}\left\{ R(s,a) + \gamma \sum_{s'\in s} P(s'|s,a)V_k(s') \right\} \tag{96}$$

8 RL in finite domains

So far we have seen the very basics of MDPs and some dynamic programming method to predict V^* and π^*. In this chapter, we will focus on RL methodologies which aim to estimate as well V^* and π^*, but in unknown environment where the MDP structure is not known. RL theory provides us a vast catalogue of algorithm that can be identified by the following characteristics

- **Model-free vs Model-based** Model free approaches try to directly estimate V^*, without computing the MDP model. On the other hand, model-based approach estimate the MDP model, and then they apply classic dynamic programming methods to estimate the state-value function.
- **On-policy vs Off-policy** On policy means that we estimate some metric, like state-value function, of the policy that we are using to collect the data. 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 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. This is feasible only on small problems. Function approximation doesn't store directly the values of the value function, but it approximate it with some function. 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. Policy-based search in policy space to find the optimal policy. 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.

8.1 Model-free prediction

With Model–free prediction we want to estimate the value function of an unknown MRP[48]. 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

[48] MDP + policy

8.1.1 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 learn directly from episodes of experience. It must be used in episodic MDPs[49] because the learning phase is based on the entire trajectory[50], from the starting state to the goal state. MC can't be used in MDPs that don't terminate. MC can be used for both prediction and control. For prediction we have

- **Input** Episodes $\{s_1, a_1, r_1, \dots s_T\}$, generated by following policy π in given MDP
- **Output** Value function V^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 for which

$$E[\hat{\mu}_n] = \mu, \quad Var[\hat{\mu}_n] = \frac{Var[X]}{n}$$

As we have said, our goal is to compute V^π. We know that based on the return v_t

$$V^\pi(s) = E[v_t | s_t = s]$$
$$v_t = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-1} r_{t+T}$$

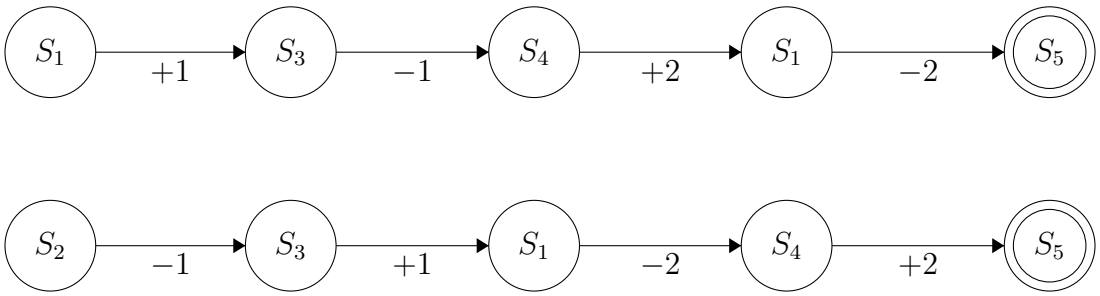
Monte Carlo policy evaluation uses empirical mean return instead of expected return to estimate V^π. In practice, we perform various episodes and we collect the returns for every state. Then, we use the empirical mean of the returns collected in s to estimate $V^\pi(s)$. The averaging of the returns can be performed in two ways

- **First visit** Average returns only for the first time s is visited.
- **Every visit** Average returns for every time s is visited.

[49] An 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

[50] A trajectory is a sequential combination of states, actions, and rewards $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

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

$$\hat{V}_1^\pi(s_1) = 1 - 1 + 2 - 2 = 0$$
$$\hat{V}_2^\pi(s_1) = -2 + 2 = 0$$
$$\hat{V}^\pi(s_1) = \frac{0+0}{2} = 0$$

Using every visit, we consider all the occurrences of s_1

$$\hat{V}_{11}^\pi(s_1) = 1 - 1 + 2 - 2 = 0$$
$$\hat{V}_{12}^\pi(s_1) = -2$$
$$\hat{V}_{21}^\pi(s_1) = -2 + 2 = 0$$
$$\hat{V}^\pi(s_1) = \frac{(0) + (-2) + (0)}{3} = -\frac{2}{3}$$

This two methods have different properties. First visit is unbiased. We know for the bias-variance tradeoff that, if we have low bias, we have high variance. So first visit is suited for problems with many many training samples. First visit will produce noisy estimation of the state-value function even with a decent amount of samples. This limits a lot the learning speed, because we need to calculate a lot of episodes. Every visit is consistent[51]. 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. From a computational point of view, it would be nice if we could calculate incrementally the empirical mean of the return. If we observe a new episode, we can update the mean without recalculating it from scratch.

[51] An estimator is consistent if $\lim_{n\to\infty} \hat{x}_n = E[x]$. If we have infinite samples the estimator converges to the true value

$$\begin{aligned}\hat{\mu}_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\ &= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\ &= \frac{1}{k}\left(x_k + (k-1)\hat{\mu}_{k-1}\right) \\ &= \hat{\mu}_{k-1} + \frac{1}{k}(x_k - \hat{\mu}_{k-1}) \end{aligned} \tag{97}$$

In the MC case, using (97) we have

$$V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}(v_t - V(s_t)) \tag{98}$$

where $N(s_t)$ is the number of times we have calculated the return for s_t. This updates can be extended to non-stationary problems. In this cases is useful to forget old episodes.

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t)) \tag{99}$$

where α is a learning rate between zero and one. This has the result to weight more the new samples, 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 - Characteristic recap

- Must work on episodic MDPs. To learn we use the entire episode
- Unlike dynamic programming, MC can evaluate at each episode only one choice at each state
- MC doesn't bootstrap. This term will be clear later
- Time required to estimate one state does not depend on the total number of states, but on the variance of the samples

8.1.2 Temporal difference learning (TD(0))

Before explaining this new algorithm, it's worth recalling a very important learning rate property.

Proposition 8.1. *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 estimator*

$$\mu_i = (1 - \alpha_i)\mu_{i-1} + \alpha x_i$$

with $\mu_i = x_1$ and α_i's are learning rates.
If $\sum_i \alpha_i = \infty$ and $\sum_i \alpha_i^2 < \infty$, then $\hat{\mu}_n \overset{a.s.}{\to} \mu$. The estimator is consistent.

For example $\frac{1}{i}$ satisfies the above condition, and so produces a consistent estimator. 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. TD are a model-free methods which learn directly from the episodes experience. 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

$$V(s_t) \leftarrow V(s_t) + \alpha(\mathbf{v_t} - V(s_t))$$

For TD we simply substitute v_t with $r_{t+1} + \gamma V(s_{t+1})$

$$V(s_t) \leftarrow V(s_t) + \alpha(\mathbf{r_{t+1}} + \gamma \mathbf{V}(\mathbf{s_{t+1}}) - V(s_t))$$

Doing so, we can use partial rollout of an episodes, because we don't need the entire episode to estimate r_t. This practice is called bootstrapping. $r_{t+1} + \gamma V(s_{t+1})$ is called **TD target**. $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called **TD error**.

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

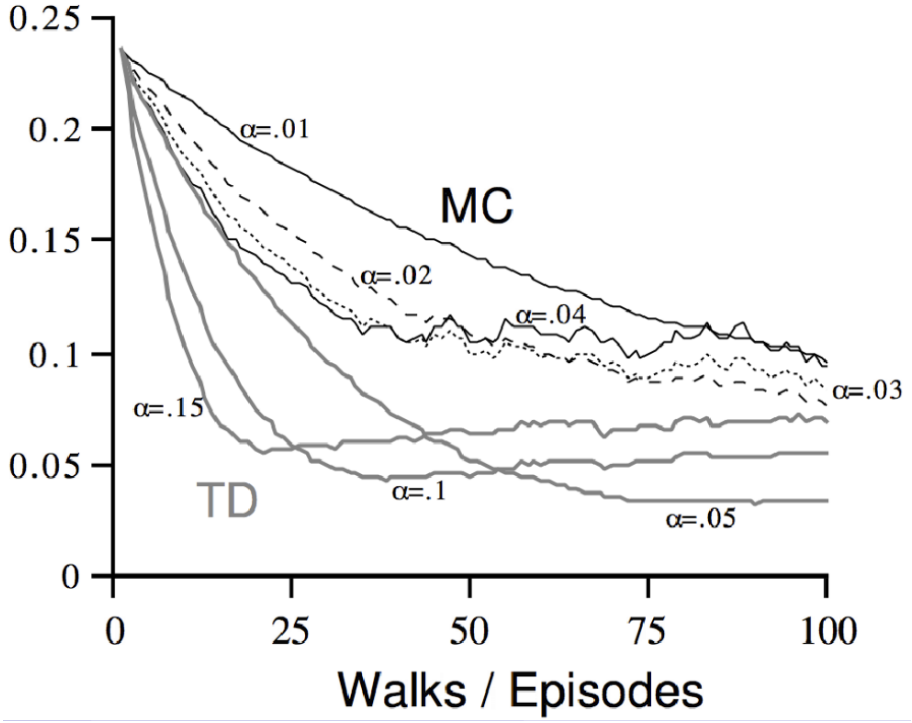
$$\begin{aligned} V(s_t) &\leftarrow V(s_t) + \alpha(v_t - V(s_t)) \\ &\leftarrow V(s_t) + \alpha v_t - \alpha V(s_t) \\ &\leftarrow (1 - \alpha)V(s_t) + \alpha v_t \end{aligned}$$

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** TD can learn online after every step. MC must wait until end of episode before return is known. TD can be use in continuing(non–terminating) environments. MC only works for episodic(terminating) environments
- **Bias-Variance** TD has low variance, some bias. MC has high variance, zero bias. TD is more used when the problem is Markovian. MC returns in non-Markovian problems.

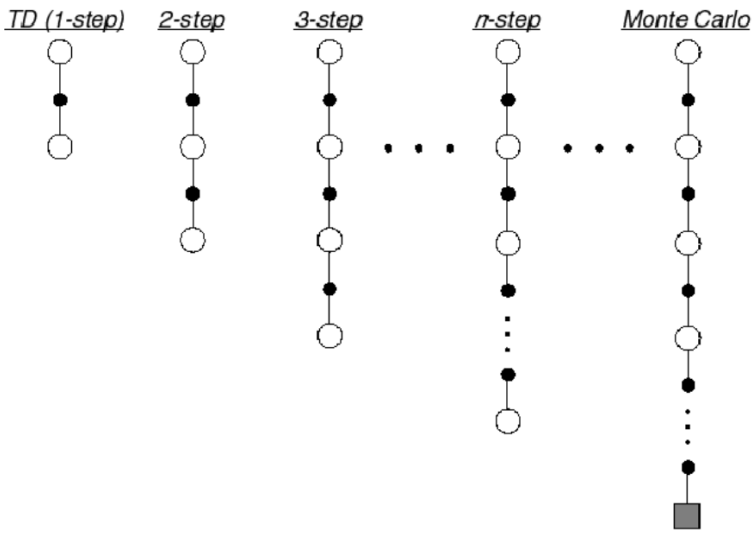


RMS error averaged over states

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.

8.1.3 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 $\lambda = 1$.



Circles are states and 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 $n = 1, 2, \ldots, \infty$ the n-step return is

$$v_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}) \tag{100}$$

The n-step return is composed by n immediate rewards, plus the estimate of the state-value function in the upcoming state. If we replace this estimator in the TD learning we get

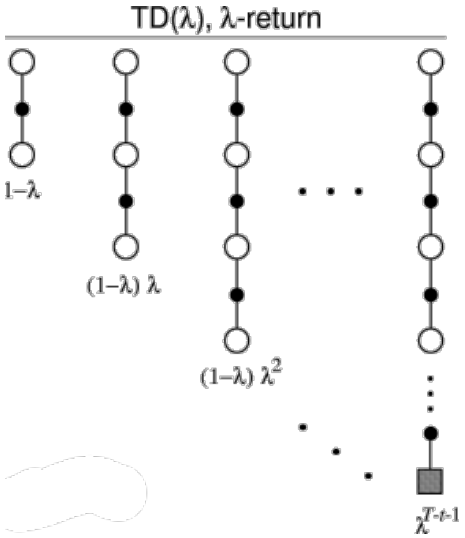
$$V(s_t) \leftarrow V(s_t) + \alpha(v_t^{(n)} - V(s_t))$$

This isn't the TD(λ) formula, but it gives us an intuition about the procedure. This approach has a problem. To calculate the n-step return we must perform n steps. We can't update our estimate of the state-value function at each step. To solve it, we can average the n-step returns after each step, to combine information from all steps. In practice, we perform a weighted average called λ-return v_t^λ. Using weight $(1-\lambda)\lambda^{n-1}$,

$$v_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} v_t^{(n)} \tag{101}$$

We can use this new estimator in the TD learning formula

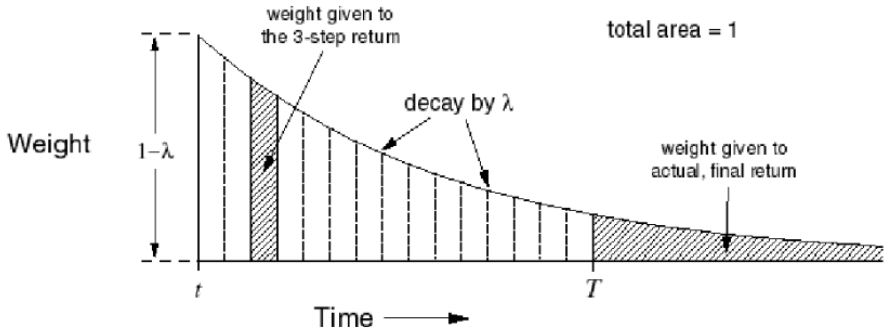
$$V(s_t) \leftarrow V(s_t) + \alpha(v_t^\lambda - V(s_t)) \tag{102}$$



This approach is called **Forward-view** TD(λ). We can see that for $\lambda = 0$, only the 1-step return have a weight different from zero. 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 tradeoff by changing λ.

Note - Weights As one can imagine, the weight formulation is very important. One of the most important property is that the weight must sum to 1. In our case, the sum of the weight is

$$\begin{aligned} WeightSum &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1} \\ &= \underbrace{\sum_{n=1}^{\infty}(1-\lambda)\lambda^{n}}_{\text{geometric series}} = 1 \end{aligned}$$

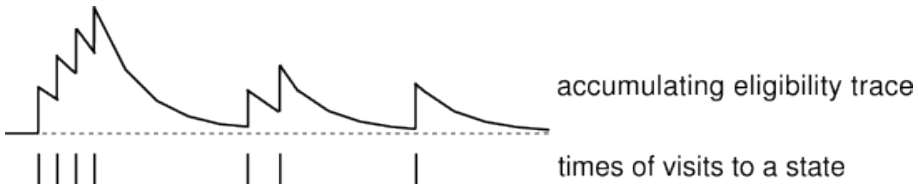


To recap, we want to estimate the state-value function of a state. Starting from the given state, at each step, we perform an action, stretching the trajectory length. Every step, we calculate the n-step return. To estimate the starting state value function, we average over the n-step returns obtained at each step, weighted with the hyperparameter λ. With this approach(forward-view), we still need to perform every step until we reach the complete episodes, because v_t^λ still need the n-step return associated to the complete episode. To solve this problem 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
- **Recency heuristics** We assign credit to the most recent states
- **Eligibility traces** We combine the frequency and recency heuristic[52]

$$e_{t+1}(s) = \underbrace{\gamma\lambda e_t(s)}_{\text{recency term}} + \underbrace{1(s = s_t)}_{\text{frequency term}} \tag{103}$$

The eligibility trace is calculated in every state, and is update at each step. The frequency term makes the trace decay over time because γ(discount factor) and λ(TD hyperparameter) are less than one. The recency term adds one to the trace every time the state is visited.



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. For TD we had

$$\begin{aligned} V(s_t) &\leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) && \text{TD} \\ &\leftarrow V(s_t) + \alpha(\delta_t) \\ V(s_t) &\leftarrow V(s_t) + \alpha(\delta_t e_t(s)) && TD(\lambda) Backward - view \end{aligned}$$

TD(λ) uses the eligibility traces to give more importance to the most recently or more frequently visited states. The states with high eligibility trace will rely more on newly seen data, by giving a boost to the learning rate.

[52] $+1(s = s_t)$ means that we add one only if the current state s is equal to s_t

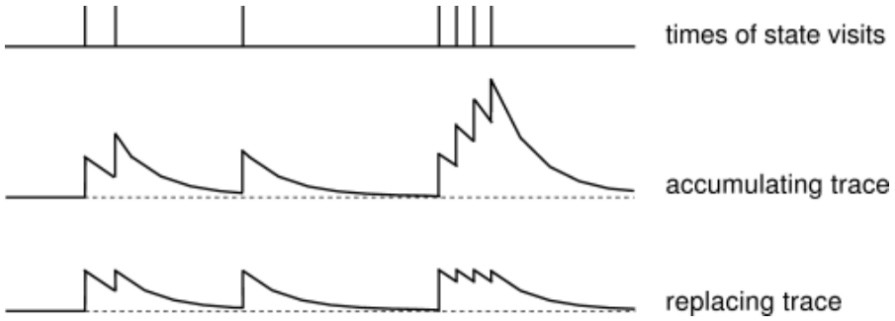
Here we have the algorithm for backward-view TD(λ)

Algorithm 3: backward-view TD(λ)

Output : V^π
Input : S, A, π, γ, λ
Initialize: V(s) arbitrarily
for *all episodes* **do**
  $e(s) \leftarrow 0, \forall s \in S$
  $s \leftarrow$ starting state
  repeat
    $a \leftarrow$ action given by π for s
    Take action a, observe r, and next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for *all* $s \in S$ **do**
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    end
    $s \leftarrow s'$
  until s *is terminal*;
end

Using accumulating traces, frequently visited states can have eligibilities greater than one. This can be a problem for convergence. **Replacing traces**, instead of adding 1 when you visit a state, set that trace to 1.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), & \text{if } s \neq s_t \\ 1, & \text{if } s = s_t \end{cases} \tag{104}$$



8.2 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. 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.

8.2.1 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. Once we have an estimate, we can perform the second step, policy improvement. From the value function estimate, we can calculate the relative greedy policy, which we are sure is better then the previous policy. We iterate this two step until two consecutive policy evaluation are the same. The relative policy is the optimal one.
The policy evaluation step uses 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. 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.

$$\pi'(s) = \underset{a \in A}{argmax} \left\{ R(s,a) + \gamma \sum_{s' \in S} \mathbf{P}(\mathbf{s}'|\mathbf{s},\mathbf{a}) V(s') \right\}$$

We can do policy improvement model-free by using the action-value function

$$\pi'(s) = \underset{a \in A}{argmax} \left\{ Q(s,a) \right\} \tag{105}$$

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 $Q = Q^\pi$. 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**. The policy will be calculated as follow

$$\pi(s,a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a^* = \underset{a \in A}{argmax}\{Q(s,a)\} \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases} \tag{106}$$

Simplest idea for ensuring continual exploration. All the m actions have non-zero probability of being chosen. The greedy action have a large probability of being chosen. ϵ manages the exploration-exploitation dilemma. Larger values of ϵ will generate policies which explore more, because it gives more probability to the non-greedy actions. The very good thing about ϵ-greedy policy is its theoretical guarantees

Theorem 8.1 (ϵ-greedy policy improvement)**.** *For any ϵ-greedy policy π, the ϵ-greedy policy π' with respect to Q^π is an improvement*

$$\begin{aligned} Q^\pi(s, \pi'(s)) &= \sum_{a \in A} \pi'(a|s) Q^\pi(s,a) \\ &= \frac{\epsilon}{m} \sum_{a \in A} \big(Q^\pi(s,a)\big) + (1-\epsilon) \max_{a \in A} Q^\pi(s,a) \\ &\geq \frac{\epsilon}{m} \sum_{a \in A} \big(Q^\pi(s,a)\big) + (1-\epsilon) \sum_{a \in A} \frac{\pi(a|s) - \frac{1}{\epsilon}}{1-\epsilon} Q^\pi(s,a) \\ &= \sum_{a \in A} \pi(a|s) Q^\pi(s,a) = V^\pi(s) \end{aligned}$$

Therefore from policy improvement theorem, $V^{\pi'}(s) \geq V^\pi(s)$

This is great news, the ϵ-greedy policy with respect to a policy is for sure better. To recap we have,

- **Policy evaluation** MC policy evaluation, $Q = Q^\pi$
- **Policy improvement** ϵ-greedy policy improvement

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, $Q \approx Q^\pi$

To completely define on-policy MC control, we still need some definitions.

Definition 8.1 (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*

$$\lim_{k\to\infty} N_k(s,a) = \infty, \quad \forall(s,a)$$

- *The policy converges on a greedy policy*

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = \underset{a'\in a}{argmax}\{Q_k(s',a')\})$$

In GLIE MC control we have. Given sample k^{th} episode using $\pi : \{s_1, a_1, r_1, \ldots, s_T\} \sim \pi$, for each state and action in the episode we have

$$N(s_t,a_t) \leftarrow N(s_t,a_t) + 1$$
$$Q(s_t,a_T) \leftarrow Q(s_t,a_t) + \frac{1}{N(s_T,a_t)}(v_t - Q(s_t,a_t))$$

The policy improvement step based on the new action-value function will use

$$\epsilon \leftarrow \frac{1}{k}$$
$$\pi \leftarrow \epsilon - greedy(Q)$$

Theorem 8.2 (GLIE Monte-Carlo)**.** *GLIE Monte-Carlo converges to the optimal action-value function,* $Q(s,a) \to Q^*(s,a)$

Now we have a way to find the optimal policy with a model-free approach.
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. $\frac{1}{1-\gamma}$
- **Sampling** related to the learning rate α for the estimation of Q
- **Exploration** related to ϵ, for the ϵ-greedy strategy

To have good result during the learning phase, these three hyperparameters 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. 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 $\epsilon \sim (1 - \frac{m}{M})^2$. 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.

8.2.2 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. It has lower variance, it's online and it can work with incomplete sequences. The most simple way to adapt the control problem using TD is by applying temporal difference to Q in the policy evaluation step. Then, we can use ϵ-greedy policy improvement. These steps are done every time we take a new action.
This on-policy control is called **SARSA**[53]. As in the MC case we need to evaluate the action-value function

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

At each time-step we perform

- **Policy evaluation** SARSA, $Q \approx Q^{\pi}$
- **Policy improvement** ϵ-greedy improvement

Algorithm 4: SARSA On-Policy control

Output : Q^*
Input : S, A, π_0, γ, ϵ
Initialize: $Q(s,a)$ arbitrarily
$\pi \leftarrow \pi_0$
do
  $s \leftarrow$ starting state
  $a \leftarrow$ action given by π for s
  repeat
    Take action a, observe r, s'
    $a' \leftarrow$action given by π for s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
    $\pi \leftarrow \epsilon$-greedy(π)
  until *s is terminal*;
while *convergence*;

To ensure that the SARSA algorithm converges to the optimal solution, we have to check if some conditions are respected

[53]SARSA stands for: State Action Reward State Action

Theorem 8.3. *SARSA converges to the optimal action-value function, $Q(s,a) \to Q^*(s,a)$, under the following conditions*

- *GLIE exploration strategy(sequence of policies $\pi_t(s,a)$)*
- *Robbins-Monro sequence of step-sizes α_t*

$$\sum_{t=1}^{\infty} \frac{1}{\alpha_t} = \infty$$
$$\sum_{t=1}^{\infty} \frac{1}{\alpha_t^2} \leq \infty$$

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.

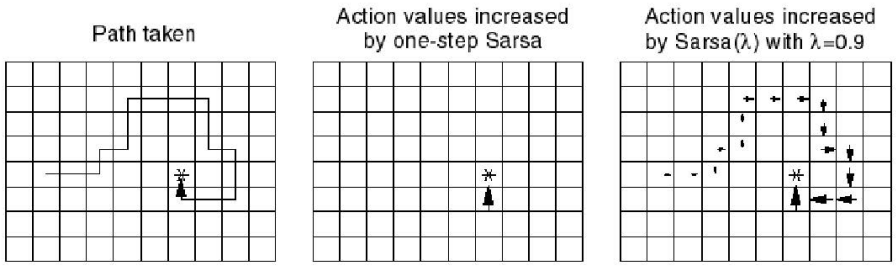
$$e_t(s,a) = \gamma\lambda e_{t-1}(s,a) + \mathbf{1}((s_t,a_t) = (s,a))$$

Algorithm 5: Backward-view SARSA(λ)

Output : Q^*
Input : S, A, π_0, γ, λ, ϵ
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 π for s
  repeat
    Take action a, observe r, and next state s'
    $a' \leftarrow$ action given by π 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
    $s \leftarrow s'$
    $a \leftarrow a'$
    $\pi \leftarrow \epsilon$-greedy(π)
  until *s is terminal*;
while *convergence*;

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. 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.

8.2.3 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. For example, splitting the two policy can give us an advantage when learning from observing humans or other agents, or reuse experience generated from old policies$(\pi_1, \dots, \pi_{t-1})$. How can we estimate a policy from another one? We can use the concept of **importance sampling**. This 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[54]. 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 $E_{x\sim p}[f(x)]$. 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.

$$\begin{aligned} E_{x\sim p}[f(x)] &= \int p(x)f(x)dx \\ &= \int q(x)\left(\frac{p(x)}{q(x)}f(x)\right)dx \end{aligned}$$

[54]Note 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

$$= E_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right]$$

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. When p gives a big importance to a region, and q doesn't, we must amplify the sample, if we have taken it from q. 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 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 sampling[55]. 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}.

$$v_t^{\mu} = \frac{\pi(a_t|s_t)\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_t|s_t)\bar{\pi}(a_{t+1}|s_{t+1})}\cdots\frac{\pi(a_T|s_T)}{\bar{\pi}(a_T|s_T)}v_t \tag{107}$$

Once we obtained the re-weighted return, we can update the action-value function using v_t^{μ}

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(v_t^{\mu} - Q(s_t,a_t)) \tag{108}$$

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 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

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma\frac{\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_{t+1}|s_{t+1})}Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right) \tag{109}$$

With SARSA we have much lower variance than Monte–Carlo importance sampling and policies only need to be similar over a single step.

[55] $f(x) \to v_t$, $p(x) \to \pi$, $q(x) \to \bar{\pi}$

Off–Policy Control with Q–learning Q-learning is the most used RL algorithm. It's very similar to SARSA,

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in A}\{Q(s',a')\} - Q(s,a)) \tag{110}$$

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 6: Q-learning

Output : Q^*
Input : S, A, π_0, γ, ϵ
Initialize: Q(s,a) arbitrarily
$\pi \leftarrow \pi_0$
do
  $s \leftarrow$ starting state
  repeat
    $a \leftarrow$ action given by π for s
    Take action a, observe r, and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in A}\{Q(s',a')\} - Q(s,a))$
    $s \leftarrow s'$
    $\pi \leftarrow \epsilon$-greedy(π)
  until *s is terminal*;
while *convergence*;

9 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' know the Q function. We have to learn it **online**[56]. 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.

[56]Online 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

9.1 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, $A = \{a_1, \ldots, a_N\}$
- 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.

$$\pi(a_i|s) = \begin{cases} 1-\epsilon & \text{if } \hat{Q}(a_i|s) = \max_{a\in A} \hat{Q}(s|s) \\ \frac{\epsilon}{|A|-1} & \text{otherwise} \end{cases} \tag{111}$$

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**

$$\pi(a_i|s) = \frac{e^{\frac{\hat{Q}(a_i|s)}{\tau}}}{\sum_{a\in A} e^{\frac{\hat{Q}(a|s)}{\tau}}} \tag{112}$$

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.

9.1.1 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 arms 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 $T \to \infty$. 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 9.1 (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 $T = N_t(a_1) + \cdots + N_t(a_N)$. 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 9.1. *Given a MAB stochastic problem, any algorithm satisfies*

$$\lim_{T \to \infty} L_T \geq log(T) \sum_{a_i | \Delta_i > 0} \frac{\Delta_i}{KL(R(a_i), R(a^*))} \tag{113}$$

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 $a_{i_t} = \underset{a}{argmax} \hat{R}_t(a)$ 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** $R(a_1), \ldots, R(a_N)$ are considered as unknown parameters. A policy selects at each time step an arm based on the observation history
- **Bayesian formulation** $R(a_1), \ldots, R(a_N)$ are considered as random variables with prior distributions $f_1, \ldots, f_N$. 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 9.2 (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

$$P(R(a_i) > \hat{R}_t(a_i) + B_t(a_i)) \leq e^{-2N_t(a_i)B_t(a_i)^2} \tag{114}$$

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 $p = t^{-4}$

$$B_t(a_i) = \sqrt{\frac{2log(t)}{N_t(a_i)}} \tag{115}$$

This ensures to select the optimal action as the number of samples increases. $\lim_{t\to\infty} B_t(a_i) = 0 \Rightarrow \lim_{t\to\infty} U_t(a_i) = R(a_i)$
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
$$B_t(a_i) = \sqrt{\frac{2log(t)}{N_t(a_i)}}, \quad \forall a_i$$
3. Play arm
$$a_{i_t} = \underset{a_i \in A}{argmax}\left(\hat{R}_t(a_i) + B_t(a_i)\right)$$

From that we can derive the specific upper bound of UCB1

Theorem 9.2 (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*

$$L_T \leq 8log(T) \sum_{i|\Delta_i>0} \frac{1}{\Delta_i} + \left(1 + \frac{\pi^2}{3}\right) \sum_{i|\Delta_i>0} \Delta_i$$

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 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 $f_1, \ldots, f_N$ as a starting point. At each round t sample from each one of the distributions $\hat{r}_i, \ldots, \hat{r}_N$. Pull the arm a_{i_t} with the highest sampled value $i_t = \underset{i}{argmax}\{\hat{r}_i\}$. 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 9.3 (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.

9.1.2 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 arms 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 9.3 (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 9.4 (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

$$\pi_t(a_i) = (1-\beta)\frac{w_t(a_i)}{\sum_j w_t(a_j)} + \frac{\beta}{N} \tag{116}$$

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 9.5 (EXP3 Upper Bound)**.** *At time T, the expected weak regret of EXP3 algorithm applied to an adversarial MAB problem with* $\beta = \eta = \sqrt{\frac{Nlog(N)}{(e-1)T}}$ *is*

$$E(L_T) \leq O\big(\sqrt{TNlog(N)}\big)$$

where the expectation is taken with respect to both the random generation rewards and the internal randomization of the forecaster

9.2 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)