Formulas and concepts of Machine Learning

Barbera Chiara

September 2019

This document puts togeheter the M.Restelli lessons (A.A. 2018/2019) of Machine Learning at Politecnico of Milan and the exercises of the teaching assistant F. Trovò. Some material has been taken online too, from the recap of Polimi DataScience and from the recap by Simone Staffa. This was made only for study purposes, with no intention of selling.

Contents

Chapter 1

Introduction

1.1 Definitions and Notation

1. **Machine Learning**: discipline used to create programs that automatically solve problems related to the analysis of data in order to extract already existing information.

2. **Supervised Learning**: given a dataset $\mathcal{D} = \{< x, t >\}$ want to find the function $f()$ such that $f(x) = t$. The inputs/attributes are x and the outputs/targets/labels are t. Inputs and targets can be **scalar** or **vectors**. For ease of notation:

   (a) **Scalar input and target**: x, t

   (b) **Vectorial input and target**: $\mathbf{x}, \mathbf{t}$. Input dimension is n, target dimension is k, the number of input vectors is N, the number of features is M.

   (c) **Random Variables**: X, Y...

   (d) **Matrices**: $\mathbf{A}, \mathbf{B}, ..., \mathbf{\Phi}$

   (e) **Sets**: $\mathcal{A}, \mathcal{B}, ...$

   Categories of supervised learning problems:

   (a) **Regression**: output t is continuous

   (b) **Classification**: output t is discrete and there is no order relationship between the different values of the target

   (c) **Probability estimation**: output t is a probability

   In case of vectorial output $\mathbf{t}$ of dimension k, the problem can be decomposed in k problems, each one with only one scalar target. This is useful especially if some components of $\mathbf{t}$ are discrete and others are continuous. Generally a SL problem matches a vectorial input $\mathbf{x}$ and a scalar output t.

3. **Unsupervised Learning**: given a dataset of inputs $\mathcal{D} = \{x\}$ want to find a function $f()$ that creates a new structure for the data.

   Categories: of unsupervised learning problems:

(a) **Compression**: remove unuseful data

(b) **Clustering**: reorder data in new categories

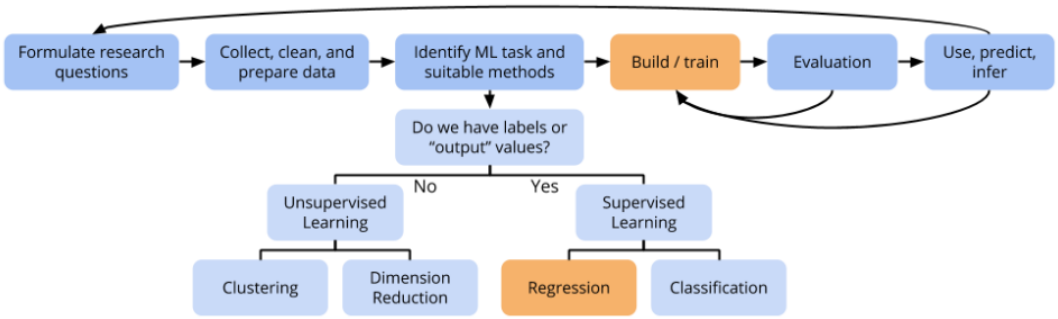


Figure 1.1: Supervised and Unsupervised learning

4. **Reinforcement Learning**: find the optimal policy in order to maximize the reward on a certain task

5. **Basic method for Supervised Learning problems**:

   (a) Define an **Hypotesis Space**: category of functions in which to search for $f()$, e.g. polynomial functions of degree 2.

   (b) Define a **Loss Function**: function $L(f(), y(x))$ used to specify how far the model $y(x)$ is from the true model $f(x)$. Given that the true model is not known, usually we only have an **empirical loss**, i.e. based on the data which is thus represented as $L(t, y(x))$. If data are few, the loss can be very noisy, whereas with lots of data it is more precise.

   (c) Define an **optimization method**: the model $y(x)$ that best estimates $f()$ is the one that minimizes the loss. In order to find it need to decide how to optimize the loss based on the choice of the loss itself

   The idea is to use part of the dataset as **training data**, i.e. to estimate the model, and part as **test data**, i.e. use the model to predict the output and compare the result with the real target.

   There are two main categories of errors:

   (a) **Approximation Error**: the error between the targets and the estimated model computed on the training data.

   (b) **Estimation Error**: computed on the test data.

   There is a trade-off between big and small hypotesis space. As the size of hypotesis space H varies:

(a) H gets **bigger**: approximation error ↓, estimation error ↑. The model performs well on training data, but if H is too big it encompasses the noise and thus it is not robust wrt new data. This can be noticed with e.g. polynomial models with too big values of parameters. This problem is called **overfitting**.

(b) H gets **smaller**: approximation error ↑, estimation error ↓. The model does not perform well on train data, but is quite robust on new data as it is simpler. If the space is too small, however, the model may be too simple. This problem is called **underfitting**.

Normally, with **few data** is better to use a **simple** model, while with a **lot** of data can use a **complex** model too because the high number of data allows to decide with higher confidence if a complex model is valid and thus will perform well on new data (the overall impact of the noise is smaller).

1.2 Statistics Background

1. **Probability**: measure of the possibility that an event E verifies.

$$P(E) \in [0, 1]$$

2. **Dependency of events**: two events A, B are independent if the fact that A verifies does not change the probability of B to verify:

$$P(AB) = P(A)P(B)$$

3. **Compatibility of events**: two events A and B are **compatible** if they can verify together, i.e.

$$P(AB) \neq 0$$

NB: incompatibility $\neq$ independence.

4. **Conditional Probability**: $p(A|B)$ measures the probability that B verifies knowing that A has verified.

$$P(A|B) = \frac{P(AB)}{P(B)}$$

The conditional probability is expressed by the **Bayes Theorem**

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

5. **Total Probability theorem**:given N disjoint events $A_1, ...A_N$ which are all the possible causes of B (not impossible),

$$P(B) = \sum_{i=1}^{N} P(B|A_i)P(A_i)$$

6. **Random Variable**: a variable whose value is related to the result of a stochastic event. Practically, a variable whose value is not known *a priori*, but it can vary inside an interval (**continuous variable**) or inside a finite set of values (**discrete variable**). Random variables are described with **probabilistic measures**, in particular:

   (a) **Probability Distribution**: $F_{\mathrm{X}}(x)$ represents $P(\mathrm{X} < x)$. If X is discrete, the F_{X} is made by steps, otherwise is continuous. In all cases it **never decreases** and:

   $$\lim_{x \to +\infty} F_{\mathrm{X}}(x) = 1$$

   $$\lim_{x \to -\infty} F_{\mathrm{X}}(x) = 0$$

   (b) **Density of Probability**: $f_{\mathrm{X}}(x)$ represents the first derivative of the probability distribution. If the variable is discrete, the density of probability is a set of **Delta of Dirac** centered in the possible values which represent the probability of that value to be assumed. In particular:

   $$f_{\mathrm{X}}(x) = \frac{d}{dx} F_{\mathrm{X}}(x)$$

   $$\int_{-\infty}^{+\infty} f_{\mathrm{X}}(x)dx = 1$$

   $$\int_{-\infty}^{x} f_{\mathrm{X}}(t)dt = F_{\mathrm{X}}(x)$$

   For discrete variables, the f is turned into the probability of the value and the integral is turned into a summation.

   (c) **Average/Expected value**: represents the value that is expected to be assumed when no other information are available. In particular:

   $$\mathrm{E[X]} = \int_{-\infty}^{+\infty} x f_{\mathrm{X}}(x)dx$$

   The continional expected value, i.e. the expected value of X when Y has verified is a function of Y = y

   $$\mathrm{E[X|Y = y]} = \int_{-\infty}^{+\infty} x f_{\mathrm{X|Y}}(x)dx$$

   The expected value can be estimated by N samples $x_i$with

   $$\bar{\mathrm{X}} = \frac{\sum_{i=1}^{N} x_i}{N}$$

   (d) **Variance**: represents how much the values are probable to be distant from the expected value. The higher the variance, the flatter and "less precise" is the estimation ; the smaller the variance, the narrower and "more precise" is the estimation.

$$\mathrm{Var[X]} = \int_{-\infty}^{+\infty} (x - \mathrm{E[X]})^2 f_{\mathrm{X}}(x) dx$$

$$\sigma_{\mathrm{X}}^2 = \mathrm{Var[X]} = \mathrm{E[X^2]} - \mathrm{E[X]}^2$$

The variance is always positive and can be estimated by N samples x_i with

$$\mathrm{S} = \frac{\sum_{i=1}^{N} (x_i - \bar{\mathrm{X}})^2}{N-1}$$

(e) Given **two random variables** X, Y s.t. $\mathrm{Y} = a\mathrm{X} + b$, where a, b, are scalars, the two variables are identically distributed and

$$\mathrm{E[Y]} = a\mathrm{E[X]} + b$$

$$\mathrm{Var[Y]} = a^2 \mathrm{Var[X]}$$

(f) **Joint variables**: two random variables X, Y are **joint** if they are taken together in an array $\begin{bmatrix} X \\ Y \end{bmatrix}$. We can define the joint density and the marginal densities:

$$f_{\mathrm{XY}}(x, y) = \frac{d^2}{dxdy} F_{\mathrm{XY}}(x, y)$$

$$f_{\mathrm{X}}(x) = \int_{-\infty}^{+\infty} F_{\mathrm{XY}}(x, y) dy$$

$$f_{\mathrm{Y}}(y) = \int_{-\infty}^{+\infty} F_{\mathrm{XY}}(x, y) dx$$

If X, Y are independent,

$$f_{\mathrm{XY}}(x, y) = f_{\mathrm{X}}(x) f_{\mathrm{Y}}(y)$$

As regards the expected value,

$$E \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} E[X] \\ E[Y] \end{bmatrix}$$

while the Variance is expressed in the **Covariance Matrix**

$$Cov[X, Y] = \begin{bmatrix} Var[X] & E[XY] \\ E[XY] & Var[Y] \end{bmatrix}$$

where $E[XY] = Corr(X, Y)$. NB :

$$\mathrm{Cov(X, Y)} = \mathrm{E[XY]} - \mathrm{E[X]E[Y]}$$

$$Corr(X, Y) = 0 \equiv \mathrm{X, Y\, not correlated}$$

$$\mathrm{X, Y}\ \textit{not correlated} \not\Longrightarrow \mathrm{X, Y}\ \textit{independent}$$

$$\mathrm{X, Y}\ \textit{independent} \Longrightarrow \mathrm{X, Y}\ \textit{not correlated}$$

Given X, Y, try to represent extracted samples in a plane

i. **Uncorrelated, same variance** = points form a **Circle** centred in the expected value

ii. **Uncorrelated, different variance**= points form an **horizontal Ellipse** centred in the expected value. The higher the variance, the larger the axis.

iii. **Correlated, different variance** = points form a **rotated ellipse** centred in the expected value. Positive slope implies positive correlation, negative slope implies negative correlation.

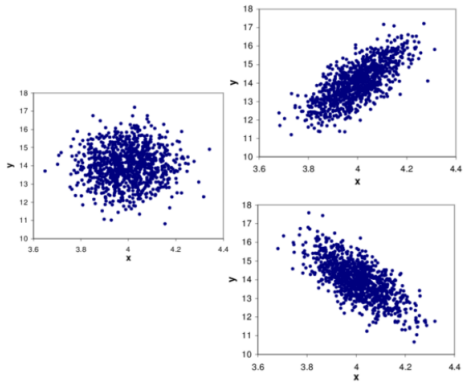


Figure 1.2: How samples are distributed according to correlation equal to 0 (left), positive (top right), negative (bottom right)

The **(normalized) correlation coefficient** varies in $[-1, 1]$ and is $r_{\mathrm{XY}} = \frac{\mathrm{E[XY]}}{\mathrm{Var[X]Var[Y]}}$

(g) Given X, Y **vectorial** random variables where $\mathrm{Y} = \mathbf{A}\mathrm{X} + \mathbf{b}$

$$\mathrm{E[Y]} = \mathbf{A}\mathrm{E[X]} + \mathbf{b}$$
$$\mathrm{Var[Y]} = \mathbf{A}\mathrm{Var[X]}\mathbf{A}^T$$

(h) **Random Variables Jointly Gaussian**. The distribution of a gaussian random scalar variable X is

$$f_{\mathrm{X}}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mathrm{E[X]})^2}{2\sigma^2}}$$

Some properties are variable for **jointly Gaussian variables**:

i. X, Y are jointly gaussian if the joint probability is gaussian

ii. if X, Y are jointly gaussian:
   · independence and uncorrelation are equivalent
   · any linear combination of them is still gaussian
   · the marginal densities are gaussians

Part I

Supervised Learning

Chapter 2

Linear Regression

2.1 Definition

The objective of linear regression is to find a model that is **linear in parameters** that maps inputs into continuous targets . The objective is thus to define the model through the vector of parameters $\mathbf{w}$. The model can be written in various ways

1. **Model linear in parameters and inputs**

   $$y(\mathbf{x}) = \mathbf{x}\mathbf{w}$$

   where:

   (a) $\mathbf{x} = \begin{bmatrix} 1 & x_1 & ... & x_{n-1} \end{bmatrix}$ is a vector 1xn (the first is set to 1 by default)

   (b) $\mathbf{w} = \begin{bmatrix} w_0 \\ ... \\ w_{n-1} \end{bmatrix}$ is the parameter vector *nx1*

   (c) $y(x)$ is the estimated target which is scalar. This model maps an input of n entries into a scalar target.

   With N vectorial inputs $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_N}$ we can form a matrix $\mathbf{X} = \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ ... \\ \mathbf{x_N} \end{bmatrix}$ which is Nxn, while the vector of parameters is still the same. The result will be a vector of targets Nx*1*:

   $$\mathbf{y}(\mathbf{X}) = \mathbf{X}\mathbf{w}$$

2. **Model linear in parameters with basis functions**: the model can be extended with M **features** or **basis functions** $\varphi_i(\mathbf{x})$ s.t. for each vectorial input $\mathbf{x}$ the model becomes

   $$y(\mathbf{x}) = \mathbf{\Phi}(\mathbf{x})\mathbf{w}$$

   where:

(a) $\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} 1 & \varphi_1(\mathbf{x}) & ... & \varphi_{M-1}(\mathbf{x}) \end{bmatrix}$ is a vector 1xM (the first is set to 1 by default). Each feature uses a vectorial input $\mathbf{x}$ to generate a scalar value

(b) $\mathbf{w} = \begin{bmatrix} w_0 \\ ... \\ w_{M-1} \end{bmatrix}$ is the parameter vector $M\mathrm{x}1$

(c) $y(x)$ is the estimated target which is scalar.

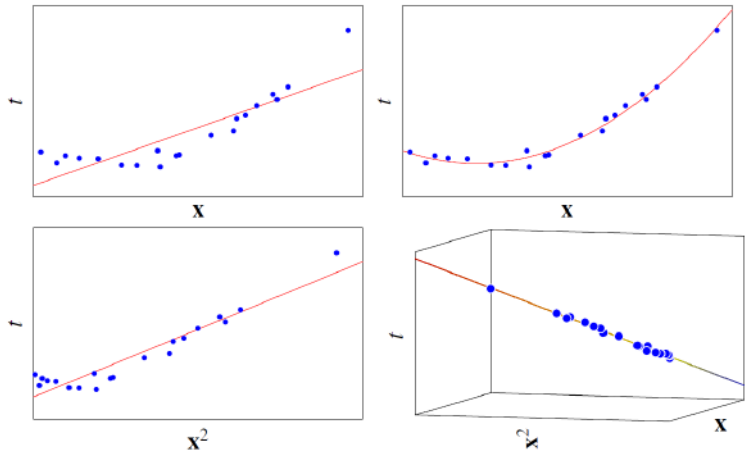


Figure 2.1: Linear model in input space (top left) and in feature space (bottom right)

The basis functions are used to comprehend models that are not linear in inputs but that are linear in the feature space. Example of basis functions are :

i. **sigmoidal:** $\frac{1}{1+exp(-\frac{x-\mu}{\sigma})}$

ii. **gaussian:** $exp(-\frac{(x-\mu)^2}{2\sigma^2})$

iii. **polynomial:** x^k

3. **Loss Function** Once the model has been defined, we need to define the Loss Function. Generally, the common choice is the **square loss**, i.e. (considering 1 input vector and 1 target value) :

$$L(t, y(\mathrm{x})) = (t - y(\mathrm{x}))^2$$

This is an **empirical loss** computed from the target and the inputs, which are random variables. This implies that the loss has a distribution, i.e. the joint distribution $p(t, \mathrm{x})$. In order to evaluate how bad the model performs on a sample $< \mathrm{x}, t >$, we consider the expected loss

$$E[L(t, y(\mathrm{x}))] = \int \int (t - y(\mathrm{x}))^2 p(t, \mathrm{x}) dx dt$$

4. **Optimization** This value is minimized by $y(\mathrm{x}) = \mathrm{E}[\mathrm{t}|\mathrm{x}]$, but need the conditional probability to find it.

2.2 Type of approaches

1. **Direct Approach**: find the optimal model $y(\mathbf{x})$ without passing from the definition above with probabilities, but estimate it directly from the data. Example : **Least Square Method, Gradient Descent**
2. **Discriminative Approach**: estimation of conditional probability and thus computation of the optimal value according to $y(\mathrm{x}) = \mathrm{E}[\mathrm{t}|\mathrm{x}] = \int tp(t|\mathbf{x})dt$. Example : **Maximum Likelyhood Method**
3. **Generative Approach**: estimation of joint probability and, form this one, of the conditional probability. Is so-called because it allows to generate new data.

2.3 Direct approach: LS Method

1. **Model definition**: $\mathbf{y = \Phi w}$ where the matrix Φ is N x M , the parameter vector is M x 1 and the result is N x 1
2. **Loss definition**:

$$L(\mathbf{t},\mathbf{y}) = \frac{1}{2}RSS(\mathbf{w}) = \frac{1}{2}(\mathbf{t} - \mathbf{\Phi}(\mathbf{x})\mathbf{w})^T(\mathbf{t} - \mathbf{\Phi}(\mathbf{x})\mathbf{w})$$

3. **Optimization method**: put gradient of Loss = 0 and check that the eigenvalues of the Hessian are all >= 0. The optimal value for the parameter vector is

$$\mathbf{w^{LS}} = (\mathbf{\Phi^T\Phi})^{-1}\mathbf{\Phi^T t}$$

The method **cannot** be applied if:

1. The matrix $\mathbf{\Phi^T\Phi}$, which is MxM, is singular (i.e. some features are dependent)
2. The number of data is smaller than the number of features, i.e. the system $\mathbf{y} = \mathbf{\Phi}(\mathbf{X})\mathbf{w}$ is not solvable wrt $\mathbf{w}$
3. There are too many data (computationally heavy): in this case use the **Gradient Descent Method**, a direct approach in which the parameters are updated iteratively according to the gradient of Loss and to the learning rate α, s.t.

$$\mathbf{w^{(k+1)}} = \mathbf{w^{(k)}} + \alpha^{(k)}\nabla L = \mathbf{w^{(k)}} + \alpha^{(k)}[-\mathbf{\Phi^T}(\mathbf{t} - \mathbf{\Phi w^{(k)}})]$$

Geometric interpretation: projection of target on the space generated by the features. The norm of the residual, i.e. $(\mathbf{t} - \mathbf{\Phi}(\mathbf{x})\mathbf{w})^T(\mathbf{t} - \mathbf{\Phi}(\mathbf{x})\mathbf{w})$ represents the euclidean distance between the target vector and its estimation, which is indeed minimized with the projection.

2.4 Discriminative approach: Likelyhood Method

This is a **discriminative method** in which the $P(t|\mathbf{x},\mathbf{w}, noise)$, i.e. the likelyhood, is defined.

1. **Model definition**: $\mathbf{y = \Phi w} + \varepsilon$, where ε is a gaussian noise $\varepsilon \sim N(0, \sigma^2)$. This implies that the estimated target is a gaussian too, as well as the likelyhood of each single entry of the target vector.

2. **Loss Definition** : Given that the entries are all **iid**, the global likelyhood for N samples is the product of the likelyhood of the single entries:

$$P(t|\Phi w, \sigma^2) = \prod_{i=1}^{N} N(t_i|\Phi(x_i)w, \sigma^2)$$

   Taking the **ln** transforms the product in a sum

$$ln(P(t|\Phi w, \sigma^2)) = \sum_{i=1}^{N} ln(N(t_i|\Phi(x_i)w, \sigma^2))$$

3. **Optimization**: putting the gradient of the **log-likelyhood** to zero we find the optimal parameters as

$$\mathbf{w^{ML} = (\Phi^T \Phi)^{-1} \Phi^T t = w^{LS}}$$

The ML estimation has the minimum variance among all the unbiased estimators (**Gauss-Markov Theorem**). The variance is

$$\mathbf{Var}[\mathbf{w^{ML}}] = (\Phi^T \Phi)^{-1} \sigma^2$$

If σ^2 is not known, it can be estimated with

$$\hat{\sigma}^2 = \frac{1}{N - M - 1} \sum_{i=1}^{N} (t_i - \Phi(x_i)\mathbf{w}^{ML})^2$$

2.5 Regularization

Addition of correction factor to the loss in order to avoid determinant of $\Phi^T \Phi$ to be near zero. The corrective factor takes into account the complexity of the model.

1. **Ridge:**

$$\mathbf{L(w) = \frac{1}{2} RSS(w) + \frac{\lambda}{2} ||w||^2}$$

   The optimal value of $\mathbf{w}$ is

$$\mathbf{w^{Ridge}} = (\lambda \mathbf{I} + \mathbf{\Phi^T \Phi})^{-1} \mathbf{\Phi^T t}$$

where $\lambda > 0$. This forces the eigenvalues of the inverted matrix to be > 0 and thus there are no big values of parameters (smoother function)

2. **Lasso:**

$$\mathbf{L(w)} = \frac{\mathbf{1}}{\mathbf{2}} \mathbf{RSS(w)} + \frac{\lambda}{\mathbf{2}} \sum_{\mathbf{i=1}}^{\mathbf{M}} |\mathbf{w_i}|$$

where $\lambda > 0$. There is no close-form solution but the non-important features are set to zero.

2.6 Bayesian Estimation

The bayesian estimation combines *a priori* knowledge with data information, which modifies it creating a **posterior** distribution (the joint probability is updated with the conditional probability of the targets, i.e. the likelyhood). Thanks to **Bayes theorem**,

$$posterior \propto prior \cdot likelyhood$$

In order to apply the method several times and to use the posterior as the next prior, both prior and posterior must be of the same type. This happens if the prior and the posterior are **conjuate priors**. At the end of each iteration, the parameters are found taking the **Maximum At Posterior**:

$$w^{MAP} = argmax\,\{posterior\} = \mathbf{w_N}$$

1. **prior**: $N(\mathbf{w_0}, \mathbf{S_0})$
2. **likelyhood**: $N(t|\mathbf{\Phi w}, \sigma^2 I)$
3. **posterior**: $N(\mathbf{w_N}, \mathbf{S_N})$ where

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

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

   (a) If $S_0 \to \infty$, i.e. the prior is uniform (non informative), the result is again $\mathbf{w^{ML}}$.

   (b) If $\mathbf{w_0} = \mathbf{0}, \mathbf{S_N} = \tau^2 \mathbf{I}$, w_N reduces to Ridge Estimation with $\lambda = \frac{\sigma^2}{\tau^2}$.

Practical example and meaning

Initially there is a prior knowledge of how the parameters are distributed. In order to have certain observed data, only some values of the parameters are sensible: this means that the distribution of the parameters change. This modification is expressed with the product of the prior and the knowledge. Iteratively multiplying by the likelyhood means adding more and more knowledge on the data, having a distribution of the data which is more and more precise (with smaller variance). This process is **discriminative** as it uses the conditional probability (the likelyhood):

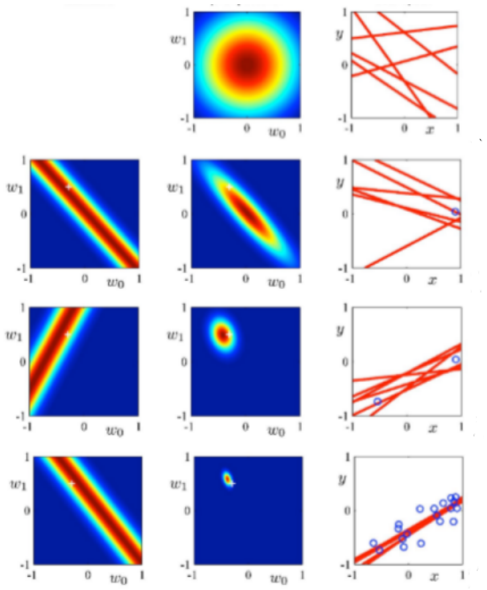


Figure 2.2: Example of Bayes estimation iterations: likelyhood on the left column, prior/posterior on the middle column. With no data, likelyhood does not exist. The right column shows the plot of data (blue dots) and the possile models according to the parameter values (red lines).

With **no data observed**, we only have a prior gaussian knowledge of the parameters. After **one data** is observed, we have a likelyhood that, multiplied by the prior, updates the parameter knowledge, resulting in a narrower **posterior** gaussian distribution. The same process is repeated with **2 data points** and then up to **20 data points**. The more data are available, the narrower and more precise the posterior becomes (keeping always a gaussian shape). When the posterior is precise enough, its maximum is taken.

Chapter 3

Linear Classification

3.1 Definitions and representation

The target t represents the **class** in which each input belongs. To classify the points, need to define the **decision boundaries** of the classes, i.e. the surfaces that divide them, assuming that:

1. All the classes are disjoint
2. Each input belongs to only one class
3. The decision surfaces must be linear in the input space or, if not possible, in the feature space.

When this has been done, given a new input $\mathbf{x}$ we can predict the class C_k in which it is placed.

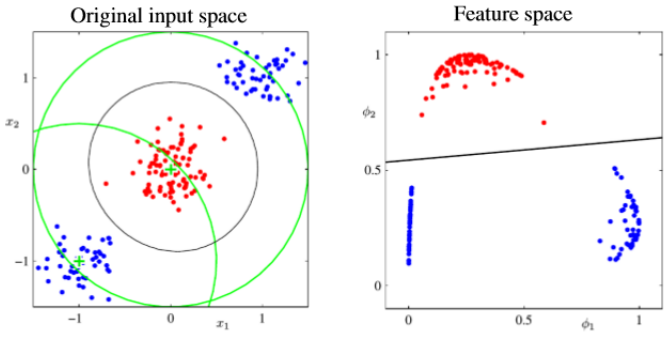


Figure 3.1: Non-linear boundaries in input space (top left) and linear boundaries in feature space (bottom right). Different colours represents different classes (two in this case).

The classification problem uses a **generalized linear model**, i.e. a non-linear function $f()$ of a linear model:

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

This means that the model is **non-linear** in parameters. With the basis functions, the model becomes:

$$y(\mathbf{x}, \mathbf{w}) = f(\mathbf{w}^{\mathbf{T}}\mathbf{\Phi}(\mathbf{x}))$$

The parameters, the matrix Φ and the inputs are defined in the same way of the linear regression: the first input is 1 by default so that the parameter w_0 represents a **translation factor**. The **target** instead can be represented in different ways according to the number of classes:

1. **2 classes**: t=0 implies class C_1 (negative class), t=1 implies class C_2 (positive class)
2. $K > 2$ **classes**: t is a vector with all entries = 0 except for one entry = 1, which represents the class that the input belongs.

   E.g. with K = 3 and an input that belongs to the class C_2 : $t = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

   In this case, we need to use K classifiers **one VS all** s.t. the decision boundaries have the origin point in common. The problem is thus decomposed in K binary classification problems:

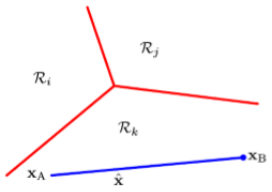


Figure 3.2: Boundaries for 3 classes

3.2 Types of approaches

1. **Discriminant approach**: the generalized linear model is used directly to define the class. The **discriminant function** $f()$ determines different methods. There is no usage of probabilities. Examples of discriminant approaches : **LS**, **Perceptron**, **K-NN**
2. **Probabilistic approach**: the generalized linear model is used to represent:
   (a) The conditional probability $p(C_k|\Phi)$ (**discriminative approach**). Examples: **Logistic Regression**
   (b) The joint probability $p(C_k, \Phi)$ (**generative approach**) Example: **Naive Bayes**

3.3 Discriminant Methods

3.3.1 LS approach

The LS method tries to find a line that separates the points, but it is very sensitive to outliers and thus it fails:

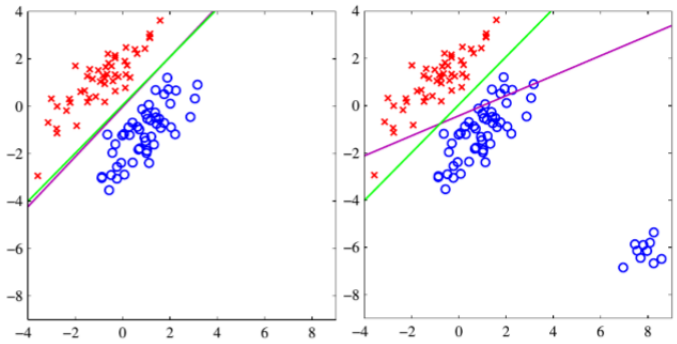


Figure 3.3: The LS method for classification

3.3.2 Perceptron

The perceptron algorithm uses the **sign** as **discriminant function**, i.e.

$$y(\mathbf{x}, \mathbf{w}) = f(\mathbf{\Phi}(\mathbf{x})\mathbf{w}) = \begin{cases} +1 & \mathbf{\Phi}(\mathbf{x})\mathbf{w} > 0 \\ -1 & \mathbf{\Phi}(\mathbf{x})\mathbf{w} < 0 \end{cases}$$

The perceptron algorithm tries to minimize the overall distance of all misclassified points from the decision boundary. The model describes an hyperplane in the feature space: if the model returns a positive value, the point is above the plane; if it returns a negative value, the point is below the plane. In the last case, the point belongs to the plane. Given that t_n is the true value for the classification, the product between the estimated target and the real target will be:

$$\begin{cases} \mathbf{\Phi}(\mathbf{x})\mathbf{w}t > 0 & correctly\ classified\ point \\ \mathbf{\Phi}(\mathbf{x})\mathbf{w}t < 0 & misclassified\ point \end{cases}$$

The **Loss function** that says how bad the classification performs is

$$L(\mathbf{t}, \mathbf{\Phi}(\mathbf{x})\mathbf{w}) = -\sum_{i \in \mathcal{M}} \mathbf{\Phi}(\mathbf{x_i})\mathbf{w}\, t_i$$

where $\mathcal{M}$ is the set of misclassified points. The higher the number of misclassified points, the bigger the value of the loss. The $-$ before the summation is used to have a positive value for the loss. If all the points are correctly classified, the loss is zero. What practically happens is that the orientation of the classifier changes so that all the points are correctly classified.

There is no close-form solution for the perceptron as the loss is minimized through **gradient descent** with **learning rate** $\alpha = 1$, checking one misclassified point at time. The learning rate is in $[0, 1]$ and represents how much the misclassification influence the change of the orientation.:

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

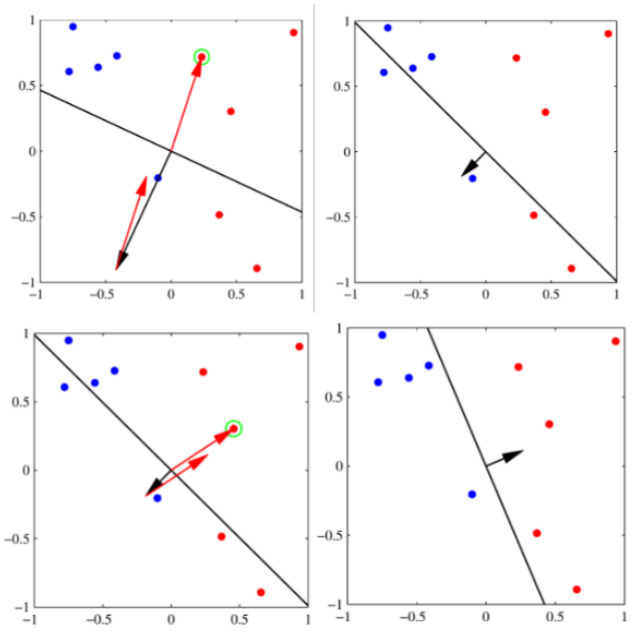


Figure 3.4: The perceptron method for classification: take the first red point, add to **w** its distance from the decision boundary (the gradient); repeat with the second point.... until error is zero.

Considerations:

1. The convergence to an exact solution in a finite number of steps is guaranteed if there is a linear decision boundary in the feature space.
2. In case of multiple solution, do not always find the same: it depends on the order in which the points are considered.
3. The convergence may exist but may be very slow.

3.3.3 KNN, a.k.a K-Nearest Neighbours

Discriminant approach that uses the **Euclidean Distance** as metric to define the class. The algorithm is:

1. Given a point P, find the K nearest points
2. Check the classes of these K neighbours
3. Assign as class of P the one that contains the majority of the neighbours.

This is a **non parametric method** because no estimation of **w** is required. Thus no training phase is used, but the prediction may be very slow.

It is affected by the **curse of the dimensionality**, i.e. the higher K the more precise and the slower the evaluation.

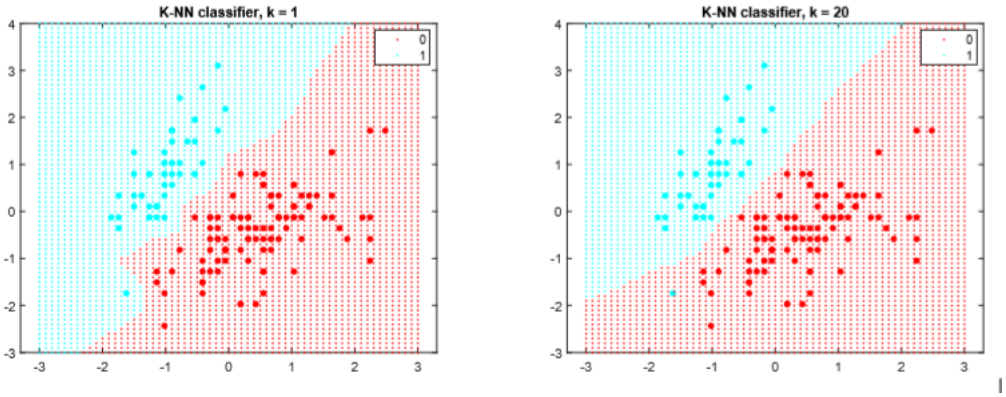


Figure 3.5: The KNN method for classification with K=1 (left) and K=20 (right). The higher K, the smoother the boundary.

3.4 Discriminative Methods

3.4.1 Logistic Regression

The logistic regression is a **discriminative method** that uses a sigmoid function of the linear model $\sigma(\mathbf{\Phi w})$ to estimate the prior probability of of belonging or not to class C_k. Given a sample $< \mathbf{x_i}, t_i >$:

$$p(C_k|model, \mathbf{x_i}) = \frac{1}{1 + exp(-\mathbf{\Phi}(\mathbf{x_i})\mathbf{w})}$$

$$p(\bar{C_k}|model, \mathbf{x_i}) = 1 - p(C_k|model, \mathbf{x_i})$$

The probability distribution of belonging or not to class C_k can be represented as a **Bernoulli distribution**:

According to this, we can write for a single sample in which $t_i \in \{0, 1\}$ the following likelyhood:

$$p(t_i|\mathbf{x_i}, model) = \sigma^{t_i}(\mathbf{\Phi}(\mathbf{x_i})\mathbf{w})(1 - \sigma(\mathbf{\Phi}(\mathbf{x_i})\mathbf{w}))^{1-t_i}$$

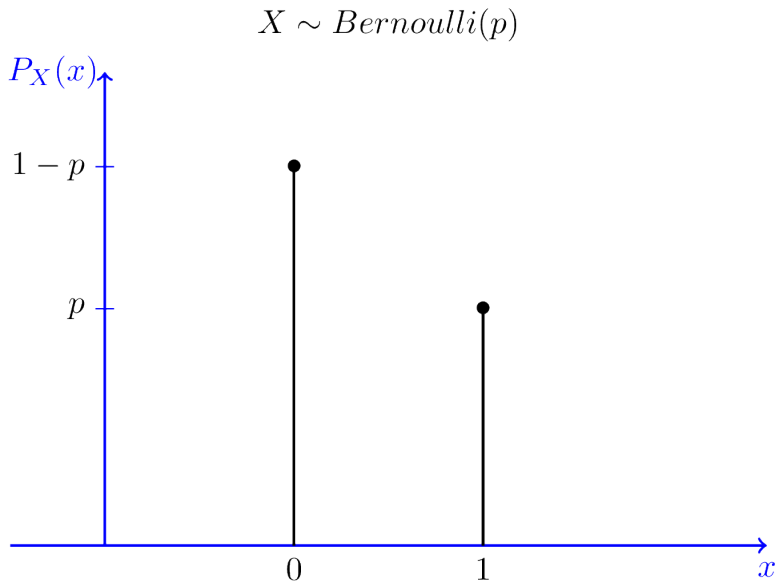


Figure 3.6: A generic Bernoulli distribution in which x can be either 1 or 0 with probability p and *1-p* written as $p^x(1-p)^{1-x}$, where the two factors are represented with Dirac Deltas centered in 0 and 1.

Considering N samples and assigning $p = \sigma(\mathbf{\Phi(x_i)w})$, thus, we have the **Likelyhood** :

$$p(\mathbf{t}|\mathbf{X}, \mathbf{model}) = \prod_{i=1}^{N} p^{t_i}(1-p)^{1-t_i}$$

The optimization is the **Max Likelyhood** which is found through **gradient descent**. Practically, what is found is the minimum of the **cross-entropy error function**, i.e. $argmin\{-ln(likelyhood)\}$

3.5 Generative Methods

3.5.1 Naive Bayes

The Naive Bayes is the **Bayesian approach** for the classification in which, for each class C_k need to :

1. Take the prior knowledge $p(C_k)$
2. Compute the likelyhood $p(data|C_k)$ which, under the assumption of samples **i.i.d** can be rewritten as $p(\mathbf{X}|C_k) = \prod_{i=1}^{N} p(\mathbf{x_i}|C_k)$
3. compute the posterior as $posterior \propto prior \cdot likelyhood = p(C_k) \prod_{i=1}^{N} p(\mathbf{x_i}|C_k)$

After having done this for all the classes, simply take class with maximum probability (**MAP**),

Chapter 4

The Bias-Variance dilemma

4.1 "No Free Lunch" theorem

Consider the set of learning algorithms $\mathcal{L}$, the set of all possible models $f()$ to be learned $\mathcal{F}$ and the ***accuracy*** *of a learner $L \in \mathcal{L}$ over non-training samples, i.e. $Acc(L)$. Considering all the possible models to be learned, the* ***average accuracy*** *is*

$$\frac{1}{|\mathcal{F}|} \sum_{\mathcal{F}} Acc(L) = \frac{1}{2}$$

This theorem lies at the basis of ML and has some important consequences:

1. There is **no perfect learner** that performs better of the others on all possible problems
2. Given a task to solve, a learner L_1 will be better than a learner L_2, but the opposite will occur for a different task.
3. The training samples *on average* are not useful for the prediction as the average performance is the same of random sampling. However, not all models are equally probable and so in practice the training data are meaningful.

Since there are no perfect learners, the choice must be done facing a **trade-off**. The typical trade-off is the **Bias-Variance dilemma**.

4.2 Bias-Variance dilemma

This trade-off involves the concepts of **precision**, represented by the variance, and **accuracy**, represented by the **bias**, which is the difference between the true model and the average value of different models taken from different datasets($B = f(x) - E[y(x)]$):

The ideal solution would be both precise and accurate (i.e. low bias and low variance), but this is not possible. The model with high variance and bias is too inaccurate and unprecise so it is useless. So, what is the best solution for the choice of an estimated model?

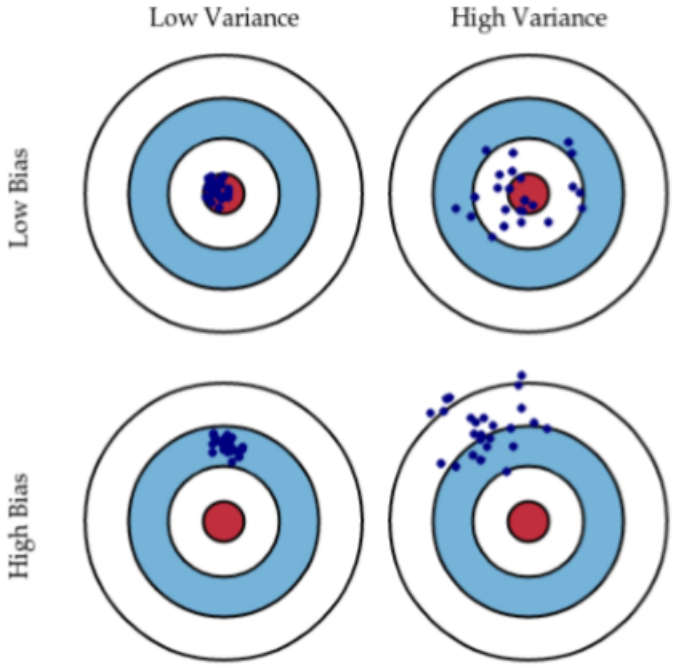


Figure 4.1: Graphical representation of precision and accuracy wrt Bias and variance

1. a model with small variance (i.e. more precise) but with bigger bias (i.e. less accurate) in which the estimated expected value is not near to the true one
2. a model with bigger variance (i.e. less precise) but with lower bias (i.e. more accurate)

Answer: it depends on the task.

The formula at the basis of the **Bias-Variance dilemma** is represented by the **error on an unseen sample**:

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

According to this formula, the error on unseen data is due to:

1. **Irreducible noise**, i.e. the noise on the data itself (Var[t] = σ^2)
2. **Precision of the model**, i.e. $Var[y(x)]$. Actually, the model itself is deterministic, but it is considered stochastic because the input x has noise.

3. **Accuracy of the model** , i.e. $B^2 = (f(x) - E[y])^2$

All these three terms are strictly positive and the error is thus > 0. Even with a perfect model, there would still be noise on the data so the error would not be zero. What can be done is the **minimization of the error** through the minimization of Bias and Variance of the model:

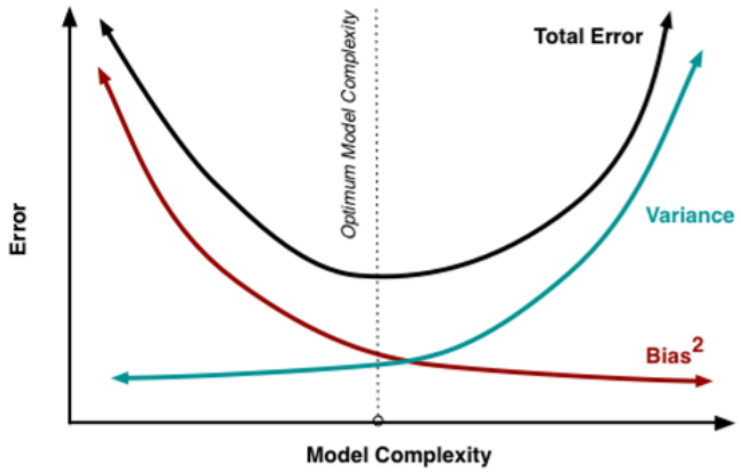


Figure 4.2: Bias and Variance representation

What can be seen by Fig 4.2 is that:

1. Simple models have big Bias and small Variance.
2. Complex models have small Bias and big Variance.

The optimal model is the one that minimizes the sum of them. NB : the minimum total error is **NOT** in the intersection! The true model is not known; there is no close form solution for the error on new data, except for the KNN method:

$$Error = \sigma^2 + \frac{\sigma^2}{K} + (f(x) - \frac{1}{K}\sum_{i=1}^{K} f(x_i))^2$$

In order to have check the performances of the model, the dataset is divided into **training set** and **testing set**. The **training error** can be considered as an optimistic biased estimation of the prediction error, which is evaluated through the testing set:

Complex models will have a small train error but, given that they incorporate noise, they will be not robust on new data, unless big datasets are used. Still, the model will perform **worse on testing than on training**

Simple models will be more robust on new data as the train error will approximate well the prediction error as the dataset size increases. Still, there will be a constant error (the bias) that could not be eliminated.

Figure 4.3: Error behaviour with complex models (left) and simple models (right).

4.3 Model Selection

Objective: finding the most meaningful features in order ho have a small error. The methods used to choose the model are:

1. **Feature selection**: Best Subset of feature. Use only some features.
2. **Regularization**: Ridge and Lasso. Use all the features, but the least important ones are put to zero.
3. **Dimensionality reduction**: PCA.

4.3.1 Feature Selection

The **Feature selection** can be implemented with:

1. **Best Subset**
2. **Regularization**
3. **PCA**, a.k.a. Principal Component Analysis

This is necessary because having too many features implies:

1. Large variance
2. Need of lot of samples
3. High computational cost

This problem is called **curse of dimensionality**.

Best Subset of Features

The **Best Subset** algorithm selects the model with smallest test error considering all the possible cases with at most M features. If M is too large, the computation cost is high. Before applying this approach, need to restrict the M features using:

1. **Ranking** in order of importance
2. **Forward and Backward Stepwise Regression**
3. **Built-in approaches**, e.g. Lasso, Ringe, ...

Once the M has been established, we can use the **Cross-Validation** to find the best model and to have an idea of the error on new data.

First, the dataset is split into 3 parts (usually with proportions 60% - 20% - 20%):

1. **Training data**: used to find the best model in each category
2. **Validation data**: used to select the optimal model
3. **Test data**: used to have a generalization of the error on new data.

The approach is divided into 3 phases:

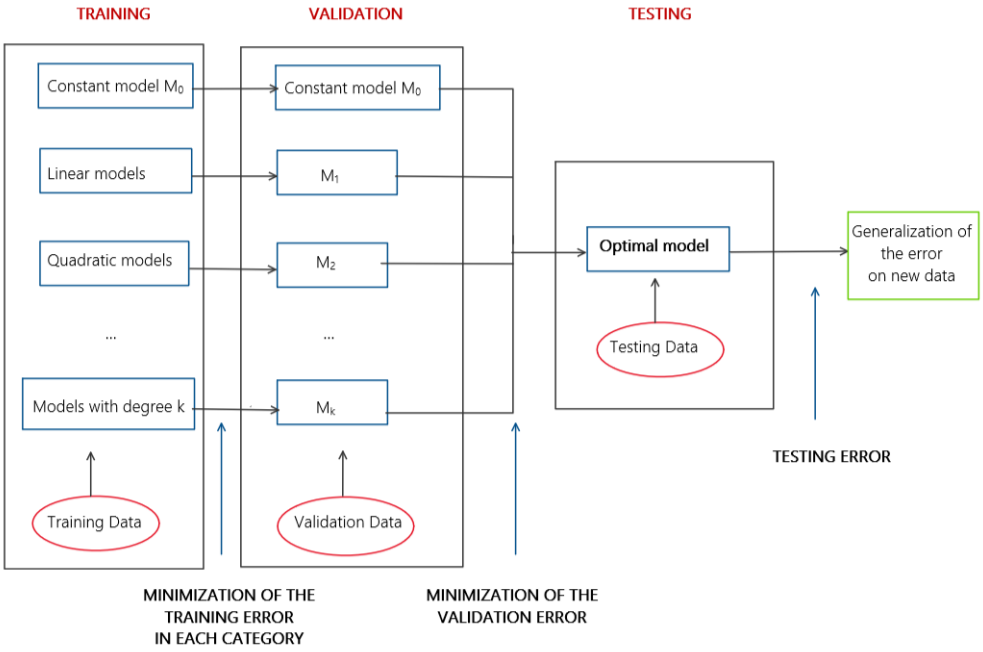


Figure 4.4: Training, Validation and Testing

1. **Training**: consider the $\binom{M}{k}$ possible models with k features, for $k \in 1...M$, and $\mathcal{M}_0$ as the constant model. For each value of k, use the training set to find the model with k features $\mathcal{M}_k$ that minimizes the training error. In this phase we obtain M+1 models $\mathcal{M}_0, \mathcal{M}_1, ...\mathcal{M}_K$, , i.e. the best model in each category.
2. **Validation**: Among the models $\mathcal{M}_0, \mathcal{M}_1, ...\mathcal{M}_K$, obtained by the training phase need to choose only one, i.e. need to decide how many features to use. To do this, use the validation set as input of the models and compute the **validation error**. The best model is the one that minimizes it.

3. **Testing**: with the optimized model, use the test data as input and compute the average **test error**: this will be an estimation of the error on new unseen data.

The estimation of the error on new data cannot be done neither with the training error (because the learnt model is based on them) nor with the validation error (as the selection of the final type of model depends on the validation data). Alternative ways to split the data into training and validations are the **LOO** (Leave One Out) approach and the **k-fold Cross Validation** (see the Appendix).

An alternative way of having a generalization of the error on new data does not use validation set: the **adjustment technique** splits the data into training and test but, to choose the best model among $\mathcal{M}_0, \mathcal{M}_1, ...\mathcal{M}_K$, uses some indices as evaluation method (see the Appendix)

4.3.2 Regularization

Ringe and Lasso increase the **Bias** because they add a correction term which cannot be removed even with infinite data, and because of this they should not be used with lots of samples. In order to select the best subset of features, use the Cross Validation using different values of λ. If very few features are important, Lasso gives better performances as the non important features are set to zero.

4.3.3 Principal Component Analysis

The idea is to select the features which carry the highest amount of variance (the **principal components**)and to ignore the ones that (reasonably) carry only noise. Because of this, the PCA approach is not reversible, i.e. returning to the original space will add some error due to the simplification of the features. The PCA is performed by finding the **eigenvectors** of the **covariance matrix**, ordered by eigenvalues: the highest the *eigenvalue*, the more important the direction is.

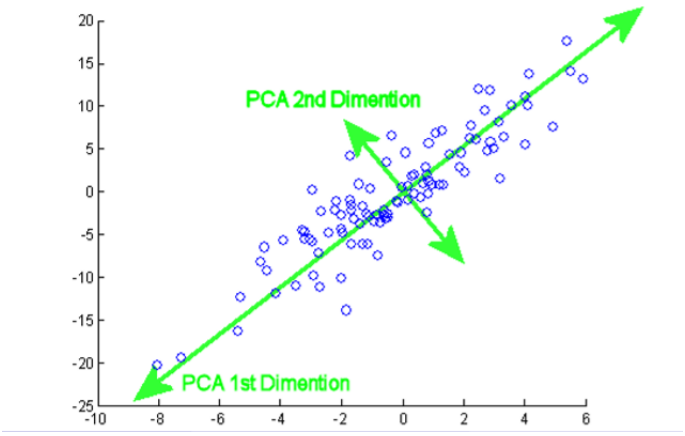


Figure 4.5: Principal Components for a linear model

The covariance matrix is *nxn*, where n is the dimension of vector $\mathbf{x}$:

$$S = \frac{1}{N-1}\sum_{i=1}^{N}(\mathbf{x_i} - \bar{\mathbf{x}})^T(\mathbf{x_i} - \bar{\mathbf{x}})$$

The portion of variance captured by the *k-th* dimension is $\frac{\lambda_k}{\sum \lambda_i}$. The data can be represented in the new space by translating them by $\bar{\mathbf{x}}$ and by projecting them on the principal components.

PCA allows to reduce computation but cannot capture non-linearities in the feature space. Moreover, it can be used to reduce the noise.

4.4 Bagging and Boosting

Bagging and Boosting are techniques used to reduce the variance without increasing the bias (or viceversa) by putting together several models (otherwise with only one model this could not be possible):

1. **Bagging**: reduce the variance without increasing the bias, i.e. it improves precision but does not change the accuracy. Given a dataset $\mathcal{D}$, generate B datasets by randomly extracting data from it (**bootstrapping**), learn B different models and then:

   - **Regression**: make an average of parameters
   - **Classification**: take the prediction with the majority of outcomes

   The variance is reduced at $Var' = \frac{Var}{N}$.

2. **Boosting**: iterate the training of a very simple model using **weighted samples**, i.e. samples are ranked by importance. At the beginning, all the samples have the same weight. At each iteration use a different dataset by re-extracting the data(whose probability has been increased or decreased through weights), re-learn the model and compute the error. The more error has been made on a sample on the previous iteration, the more it is important. The idea is to put together the models to create the final one.

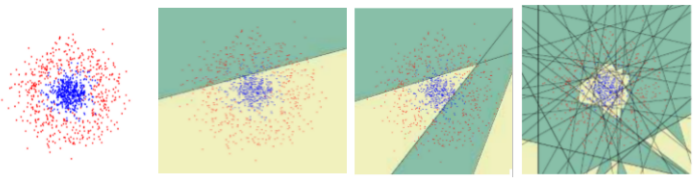


Figure 4.6: AdaBoost algorithm

Chapter 5

PAC Learning

5.1 Objective

The **Probably Approximatively Correct (PAC)** is used to define:

1. An *a priori* idea of the error on unseen data directly from the training data.
2. The minimum number of data data N that are needed to have a limited error

In case of **finite Hypothesis Space** we have:

1. Simple case: all the training data are correctly classifiable (PAC Learning)
2. Generic case: some of the training data are misclassified (Agnostic Learning)

In case of **infinite Hypothesis Space** the same quantities are expressed through the **VC dimension**.

5.2 Finite H: Simple Case

Consider a classification problem and the hypothesis space H of the classifiers. Each hypothesis h can be in one of the following cases:

	$L_{train} = 0$	$L_{train} \neq 0$
$L_{true} = 0$	Perfection (impossible)	No sense
$L_{true} \neq 0$	**VS space**	Generic Case

Table 5.1: Possible values for train and true Loss

We want to consider the **VS-space**, the subset of H in which the classifier is **consistent with the training data**. In the VS we can specify the **chance to have a big true error even if the classifier performs perfectly on the training data**. We also want to know the **number of samples** necessary to put a bound on this probability.

5.2.1 Mathematical definition of VS space

Given a binary classification problem with:

1. dataset $\mathcal{D}$ of N independent random data, drawn from probability **P**
2. The set of possible concepts that generate the real targets $\mathcal{C}$ of $c()$ such that $t = c(x)$
3. a **finite** hypothesis space H that contains the functions $h()$, used to predict the targets.

The **Version Space** $VS_{H,D}$ is the subset of H in which $L_{train} = 0 \wedge L_{true} \neq 0$.

In the Version Space :

- for any $\varepsilon \in [0, 1]$
- considering $L_{true} = P(c(x) \neq h(x)) \in [0, 1]$ computed on new data $x$drawn from the same probability distribution **P**

the probability that an hypothesis $h \in H$ will have $L_{train} = 0$ and $L_{true} \geq \varepsilon$ is limited:

$$P(\exists h \in VS_{H,D} \wedge L_{true} \geq \varepsilon) \leq |H|e^{-\varepsilon N}$$

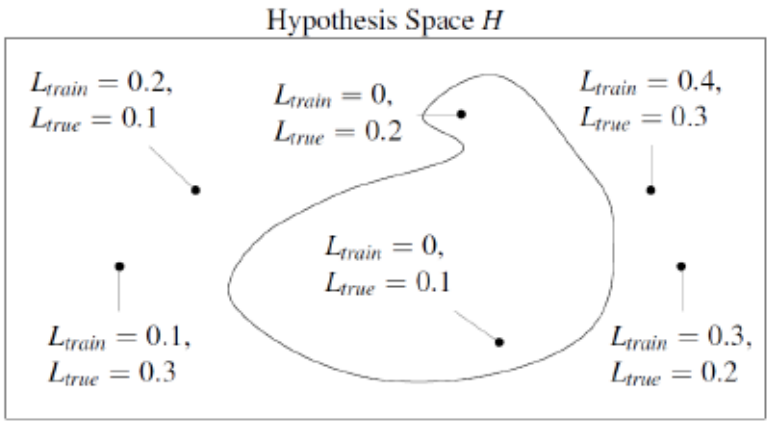


Figure 5.1: Graphical representation of the version space

5.2.2 Bound on true loss and needed samples

From the previous definition in the version space we can found the number of samples N such that the probability that $L_{true} \geq \varepsilon$ is at most δ :

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

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

Putting a bound to the true loss means that I don't want the learner to be perfect in the real context, but I want it to be **approximatively correct**. Considering that this is expressed with probabilities, we have the concept of **Probably Approximatively Correct**, i.e. (**PAC**) **Learning**.

TO SUMMARIZE: Given a classifier **consistent with the training data**, in order to be **approximatively correct**(the true error is $\geq \varepsilon$ with probability $\leq \delta$), the minimum number of samples needed is $\frac{1}{\varepsilon}\left(\ln(|H|) + \ln\left(\frac{1}{\delta}\right)\right)$.

1. **Advantage**: the probability is exponential in N
2. **Disadvantage**: the number of needed data depends on $\ln|H|$, where $|H|$ is usually high, so normally need a lot of data.

5.2.3 When to apply PAC-Learning

How can I know if a concept class is PAC-Learnable, i.e. there exist a classifier $h()$ for which we can bound the probability of having a big error and define the minimum number of data to do so?

An algorithm **L** is **PAC-Learnable** (i.e. allows to find a classifier $h()$ for which to bound the true error with a minimum number of samples) if:

- $\forall \varepsilon \in [0, 0.5]$
- $\forall \delta \in [0, 1]$
- $\forall$ possible distribution **P** of the data
- $\forall$ classifiers $c() \in \mathcal{C}$

the number of samples needed to have a classifier $h()$ for which $P(L_{true} \geq \varepsilon) \leq \delta$ is **polynomial in** $\frac{1}{\delta}, \frac{1}{\varepsilon}$

5.2.4 Efficient PAC-learning

An algorithm **L** is **Efficiently PAC-Learnable** (i.e. is PAC learnable fast) if:

- $\forall \varepsilon \in (0, 0.5)$
- $\forall \delta \in (0, 0.5)$
- $\forall$ possible distribution **P** of the data
- $\forall$ classifiers $c() \in \mathcal{C}$

the **time** needed to have a classifier $h()$ for which $P(L_{true} \geq \varepsilon) \leq \delta$ is **polynomial in** $\frac{1}{\delta}, \frac{1}{\varepsilon}, M, size(c)$

5.3 Finite H: generic case

5.3.1 Pre-requisite: Hoeffding Bound

Given N i.i.d coin flips $x_1, x_2 \ldots x_N$ (i.e. events that can be 0 or 1) and the random variable $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} x_i$ the probability that $\bar{X}$ is distant from its expected value $E[\bar{X}]$ more than ε is limited:

$$P(E[\bar{X}] - \bar{X} > \varepsilon) \leq e^{-2N\varepsilon^2} \qquad \forall \varepsilon \in (0, 1)$$

(Eg. for coin flips we expect that the average value $E[\bar{X}] = 0.5$)

5.3.2 Application of the Hoeffding Bound

In the generic case we talk about **Agnostic Learning**: $L_{train} \neq 0$ and so $VS = \emptyset$. In this case to estimate the true error we want to express the value of ε in

$$L_{true} \leq L_{train} + \varepsilon$$

In order to do this, we use the **Hoeffding Bound**:

$$P(L_{true} - L_{train} > \varepsilon) \leq e^{-2N\varepsilon^2} \leq \delta$$

We can compute ε

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

Knowing that $L_{true} - L_{train} \leq \varepsilon$ we find that

$$L_{true} \leq L_{train} + \sqrt{\frac{\ln|H| + \ln\left(\frac{1}{\delta}\right)}{2N}}$$

The latter formula represents the **Bias-Variance** trade-off for the generic case. The desired output is $L_{train} = 0$ but this does not happen so L_{train} represents the **Bias** (i.e. the loss of accuracy) while ε represents the gap between the train and the true error. The higher the gap, the less precise the model: in this sense, ε is the **variance.**

- $\uparrow |H|$: bigger variance, lower bias because we have the possibility of being more accurate and finding $h()$ with smaller L_{train}.
- $\downarrow |H|$: smaller variance, bigger bias.

5.3.3 Minimum number of samples

The minimum number of samples can be expressed by the graphical example below. Given some data that are inside the rectangle, we want to estimate R using the **tightest** rectangle R' that comprehends all the points:

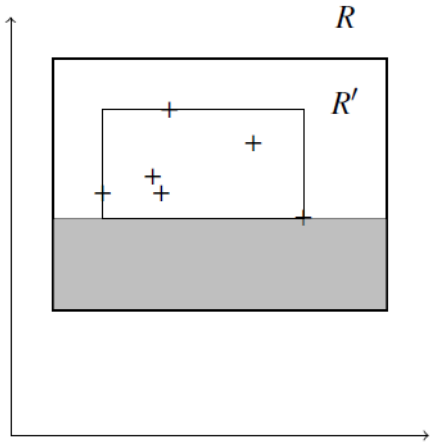


Figure 5.2: Graphical representation of the problem: R is the rectangle to find, R' is the tightest rectangle compatible with the data

The area between R and R' is the $L_{true} - L_{train}$ and we want it to be $area < \varepsilon$. This area can be divided into 4 regions (one of which is represented in the figure) and so each region has to be $region < \frac{\varepsilon}{4}$. The probability that, taking a random data, it is out of that region is $1 - \frac{\varepsilon}{4}$. We do NOT want that the error in each region is more than $\frac{\varepsilon}{4}$: if this happened, the area outside the region would be smaller than $1 - \frac{\varepsilon}{4}$. Extracting N data we have

$$(1 - \frac{\varepsilon}{4})^N$$

And this has to be true for all the 4 regions at the same time. The union probability should be smaller than δ

$$4(1 - \frac{\varepsilon}{4})^N \leq \delta$$

From this we derive

$$N \geq \frac{4}{\varepsilon} \ln \frac{4}{\delta}$$

5.4 Infinite H

5.4.1 VC-dimension

When $|H| = \infty$ the above considerations cannot be done. In this case, we have to use the concept of **VC dimension** of the hypothesis space H, i.e. the maximum number of data that can be correctly classified by any function $h() \in H$ in any possible case.
Example:

Considering the hypothesis space of the **linear functions** $y = ax + b$, $\forall a, b \in R$. $|H| = \infty$ because there are infinite possible values for a, b. The line has got $M = 2$ parameters because it uses only a linear function of x as only feature. It can correctly classify at most 2 points so $VC(H) = 2$:

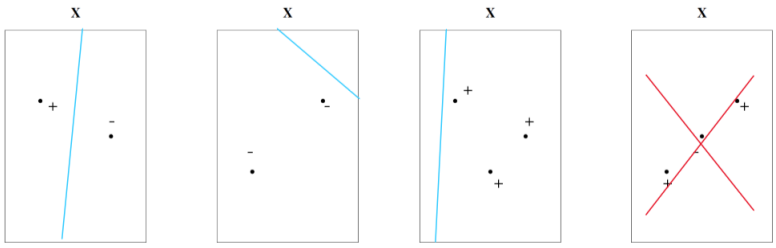


Figure 5.3: Shattering of a set of instances

Similarly, the space of quadratic functions has $M = 3$ parameters as the generic function is $y = ax^2 + bx + c$ and it can correctly classify at most 3 points, so $VC(H) = 3$.

In the general case , as **rule of thumb** an hypothesis space H in which the functions use M parameters can correctly classify at most M points, and so its VC-dimension is $VC(H) = M$ (e.g. Linear Classifier or Neural Network).However, there can be cases in which the VC dimension is infinite even with only 1 parameters (e.f. sin(ax)).

5.4.2 Mathematical definition of VC-dimension

Knowing that:

1. A **dichotomy** is a partition of a set into two disjoint subsets (in binary classifications, the positive and the negative points)

2. A set of instances $\mathcal{S}$ is **shattered by H** if, for every dichotomy in $\mathcal{S}$, there is a function $f() \in H$ that separates all the positive points on one side and all the negative points on the other (i.e. consistent with the dichotomy).

The **VC-dimension** of an Hypothesis Space H is the maximum number of elements that can be shattered by any hypothesis $h() \in H$. If this number can be indefinitely large, $VC(H) = \infty$.

Whenever $VC(H) = \infty$, the algorithm is **NOT PAC-Learnable** (e.g. 1 Nearest Neighbour, SVM with Gaussian Kernel). Moreover,

$$VC(H) <= \log_2 |H|$$

So if $|H|$ is finite, VC(H) is finite (and thus the learner is PAC-Learnable). If $|H|$ is infinite, though, the learner **may** be PAC learnable (if VC(H) is finite) or not (if VC(H) is infinite).

5.4.3 PAC-Learning and VC-dimension

$VC(H)$ can be used to express the bound on $P(L_{true} \geq \varepsilon)$ and the minimum number of data N to guarantee that:

$$N >= \frac{1}{\varepsilon}\left(4\log_2\left(\frac{2}{\delta}\right) + 8VC(H)\log\left(\frac{13}{\varepsilon}\right)\right)$$

From this, we can derive ϵ and find that:

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

Chapter 6

Kernel Methods

6.1 Definition

Main characteristics of the kernel methods:

- Used to rewrite linear parametric (**non-probabilistic**) models using **kernel functions**
- Non-Parametric (like KNN); require a **similarity metric**
- Re-use part (or all) the training points for the prediction phase
- Fast to train, slow to predict

Consider a vector of **M** features

$$\phi = \begin{bmatrix} \phi_1(\mathbf{x}) & \phi_2(\mathbf{x}) & ... & \phi_M(\mathbf{x}) \end{bmatrix}$$

A **kernel function** is expressed as a scalar product of ϕ applied at two vector inputs $\mathbf{x}, \mathbf{x}'$:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') = \sum_{j=1}^{M} \phi_j(\mathbf{x}) \phi_j(\mathbf{x}')$$

If $\mathbf{x} = \mathbf{x}'$, then $k(\mathbf{x}, \mathbf{x}') = ||\phi(\mathbf{x})||^2$. The more $k(\mathbf{x}, \mathbf{x}')$ is different from $||\phi(\mathbf{x})||^2$, the more $\mathbf{x}, \mathbf{x}'$ can be considered "different". In this sense, the kernel function allows to express a **similarity relation**.

The most basic kernel is the **identity mapping** (also called **linear kernel**), i.e.

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$$

Kernel Trick: Any function of $f(\mathbf{x}, \mathbf{x}')$ that can be rewritten s.t. the inputs appear only in scalar product, can be expressed with some sort of kernel function.

6.2 Properties of Kernel Functions

Given valid kernels k_1, k_2, k_3, the following properties hold s.t. k is a valid kernel too:

1. Symmetry
$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$$
2. Linearity
$$k = \alpha k_1 + \beta k_2 \ \forall \alpha, \beta \in R$$
3. Product
$$k = k_1 k_2$$
4. Any matrix (Symmetric Positive Semi-Definite) $\mathbf{A}$ can be used to build a kernel
$$k = \mathbf{x^T A x'}$$
5. Given a function $g(\mathbf{x}) \in R^M$
$$k = k_3(g(\mathbf{x}), g(\mathbf{x}'))$$
6. Given a polynomial function $p()$ with non-negative coefficients
$$k = p(k1)$$
7. Composition with the exponential
$$k = e^{k_1}$$

6.3 Special kind of kernels

1. **Stationary kernels**: depend only on $\mathbf{x} - \mathbf{x}'$
$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$$
2. **Homogeneous kernels**: depend only on $||\mathbf{x} - \mathbf{x}'||$. Also known as **radial basis functions**:
$$k(\mathbf{x}, \mathbf{x}') = k(||\mathbf{x} - \mathbf{x}'||)$$

Gaussian Kernel (homogeneous) is very used:

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||^2}{2\sigma^2}}$$

6.4 Ridge Regression expressed with kernels

Objective: to express the loss only with kernels.

Proof

Consider the Ridge regression loss function

$$L = \frac{1}{2}\sum_{i=1}^{N}(\phi(\mathbf{x}_i)\mathbf{w} - t_i)^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

where ϕ is the vector (dimension $1 \times M$) of the features expressed as before, $\mathbf{w}$ is the parameter vector (dimension $M \times 1$) and $\mathbf{x}_i$ is one of the N samples. The gradient of the loss wrt $\mathbf{w}$ is

$$\nabla L = \sum_{i=1}^{N}(\phi(\mathbf{x}_i)\mathbf{w} - t_i)\phi(\mathbf{x}_i)^T + \lambda\mathbf{w}$$

Putting the gradient $= 0$ we find $\mathbf{w}$ as

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{i=1}^{N}(\phi(\mathbf{x}_i)\mathbf{w} - t_i)\phi(\mathbf{x}_i)^T$$

Calling $a_n = -\frac{1}{\lambda}(\phi(\mathbf{x}_i)\mathbf{w} - t_i)$ the elements of a vector $\mathbf{a}$ (dimension $1 \times N$) and matrix $\mathbf{\Phi}$ (dimension $N \times M$)

$$\mathbf{w} = \sum_{i=1}^{N} a_i\phi(\mathbf{x}_i)^T = (\mathbf{a\Phi})^\mathbf{T}$$

We can substitute now the value of $\mathbf{w}$ in the Loss:

$$L = \frac{1}{2}(\mathbf{\Phi w} - \mathbf{t})^T(\mathbf{\Phi w} - \mathbf{t}) + \frac{\lambda}{2}\mathbf{w^T w}$$

$$L = \frac{1}{2}(\mathbf{\Phi}(\mathbf{a\Phi})^\mathbf{T} - \mathbf{t})^T(\mathbf{\Phi}(\mathbf{a\Phi})^\mathbf{T} - \mathbf{t}) + \frac{\lambda}{2}\mathbf{a\Phi\Phi^T a^T}$$

$$L = \frac{1}{2}(\mathbf{\Phi\Phi^T a^T} - \mathbf{t})^T(\mathbf{\Phi\Phi^T a^T} - \mathbf{t}) + \frac{\lambda}{2}\mathbf{a\Phi\Phi^T a^T}$$

Calling $\mathbf{K} = \mathbf{\Phi\Phi^T}$ the **Gram Matrix**:

$$L = \frac{1}{2}(\mathbf{aK^T} - \mathbf{t^T})(\mathbf{Ka^T} - \mathbf{t}) + \frac{\lambda}{2}\mathbf{aKa^T}$$

$$L = \frac{1}{2}(\mathbf{aKK^T a^T} - \mathbf{tKa^T} - \mathbf{aK^T t^T} + \mathbf{t^T t}) + \frac{\lambda}{2}\mathbf{aKa^T}$$

$$L = \left(\frac{1}{2}\mathbf{aKK^T a^T}\right) + (-\mathbf{tKa^T}) + \left(\frac{1}{2}\mathbf{t^T t}\right) + \frac{\lambda}{2}\mathbf{aKa^T} = k_1(a,a) + k_2(t,a) + k_3(t,t) - \lambda k_2(a,a)$$

The loss has been written as a sum of three valid kernel (by definition, $\mathbf{K}$ is symm) and thus it is a valid kernel.
Putting $\nabla L = 0$ we can find the only unknown $\mathbf{a}$ as

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

Observations:

1. The **Gram Matrix K** have as element (i, j) the value of $k(xi, xj) = \phi(\mathbf{x_i})\phi(\mathbf{x_j})^T$, has dimension $N \times N$ and is spd (symmetric positive definite) by construction.

2. The value of **a** is a linear combination of the target values, and so becomes the estimated value $y(\mathbf{x}) = \phi(\mathbf{x})\mathbf{\Phi^T a^T}$

Chapter 7

Gaussian Processes and SVM

7.1 Gaussian Processes

7.1.1 Definition

A **random process** is a distribution over the function space $y(\mathbf{x})$ described with first and second order momentum (mean, covariance). In practical terms, given an input x, $f(x)$is a random variable and thus the outcome is a "random function" whose **mean** is still a function and whose value varies in a "set of possible functions", depending on the values assumed by the inputs. The statistics are:

1. **Mean**: $y(\mathbf{x}) = 0$
2. **Covariance**: matrix that describes the correlation between two inputs $\mathbf{x_i}, \mathbf{x_j}$ (*similarity*) and thus its elements are **kernel functions** $k(\mathbf{x_i}, \mathbf{x_j})$. This implies that the covariance matrix is the **Gram Matrix K**.

A **gaussian process** is a random process in which the inputs are **jointly gaussian**. The elements of **K** are gaussian kernels:

$$\mathbf{y} \sim N(\mathbf{0}, \mathbf{K}) \qquad \mathbf{K_{ij}} = k(\mathbf{x_i}, \mathbf{x_j}) = e^{-\frac{||\mathbf{x_i} - \mathbf{x'_j}||^2}{2\sigma^2}}$$

The gaussian processes are the **kernel version** of the **probabilistic** approaches for **regression**.

7.1.2 Application to regression

Consider the following hypothesis:

1. The target is given by model + gaussian noise, both gaussians:

$$\mathbf{t} = \mathbf{y} + \varepsilon = \mathbf{\Phi w} + \varepsilon$$

$$\varepsilon \sim N(\mathbf{0}, \sigma^2 \mathbf{I})$$

$$\mathbf{y} = \mathbf{\Phi w} \sim N(0, \mathbf{K})$$

2. The noise on any two inputs is independent.

3. The noise and the model are independent.

The targets are thus sum of independent gaussian distribution and because of this the mean and the covariance simply add:

$$\mathbf{t} \sim N(0, \mathbf{C}) = N(0, \mathbf{K} + \sigma^{\mathbf{2}}\mathbf{I})$$

This implies that the elements of the covariance are

$$C(\mathbf{x_i}, \mathbf{x_j}) = e^{-\frac{||\mathbf{x_i} - \mathbf{x'_j}||^2}{2\theta^2}} + \sigma^2 \delta_{ij}$$

The values of θ and σ influence the appearance of the possible output. In particular, considering a generic

$$\theta_0 e^{-\frac{\theta_2 ||\mathbf{x_i} - \mathbf{x'_j}||^2}{2}} + \theta_2^2$$

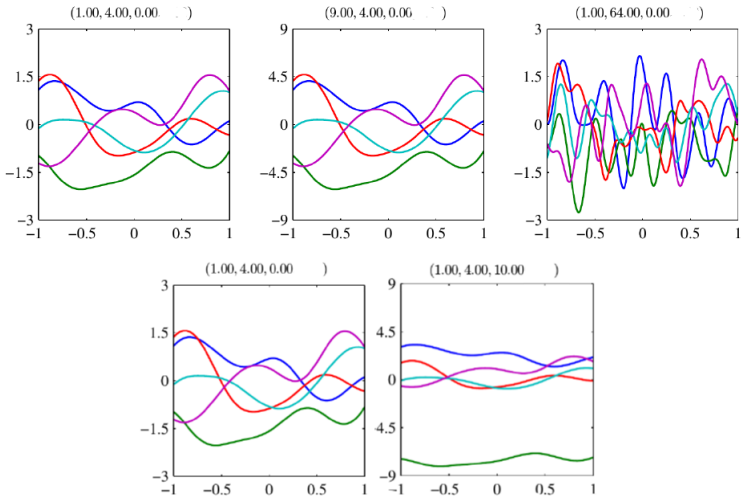


Figure 7.1: What happens changing the parameters $\theta_1, \theta_2, \theta_3$

1. Increasing θ_0, the range increases.

2. Increasing θ_2, the output are less smooth (this means reducing the variance in the exponential)

3. Increasing θ_3, the output is globally "flatter"

Given a new input, the distribution changes in

$$\mathbf{t} \sim N(0, \mathbf{C_{new}})$$

where $\mathbf{C_{new}}$ is still symmetric and has a new line that represents the similarity between an the new input and all the other inputs:

$$\mathbf{C_{new}} = \begin{bmatrix} \mathbf{C} & \mathbf{k}(\mathbf{x_{new}}, \mathbf{x_{1:N}})' \\ \mathbf{k}(\mathbf{x_{new}}, \mathbf{x_{1:N}}) & k(\mathbf{x_{new}}, \mathbf{x_{new}}) + \sigma^2 \end{bmatrix}$$

7.2 Support Vector Machines

7.2.1 Introduction

SVM is the kernel-version method for **classification**. It is very complex, powerful instance-based method whose main ingredients are:

1. A subset of the training data (do not use them all: if so, it is useless to use SVM), i.e. the **support vectors**.
2. Weights (vector α).
3. A Kernel, as it is a measure of similarity.

The prediction is a function of the support vectors (in $\mathcal{S}$):

$$f(\mathbf{x_i}) = \sum_{j \in \mathcal{S}} \alpha_j t_j k(\mathbf{x_i}, \mathbf{x_j}) + b$$

It is built as the sign of a linear combination of the targets of the support vector; the importance of each target is tuned with the similarity between the input and the support vector: the more similar they are, the more important the contribution of the target of the SV. The next part will describe how to obtain this result.

7.2.2 From perceptron to SVM

The SVM method derivation starts from the perceptron (which is **parametric**, not instance-based). The classifier of the perceptron is

$$f(\mathbf{x_i}) = sign\left(\sum_{k=1}^{M} w_k \phi_k(\mathbf{x_i})\right)$$

But if we fix the parameter $\mathbf{w}$ as

$$w_k = \sum_{j=1}^{N} \alpha_j t_j \phi_k(\mathbf{x_j})$$

we obtain

$$f(\mathbf{x_i}) = sign\left[\sum_{k=1}^{M}\left(\sum_{j=1}^{N} \alpha_j t_j \phi_k(\mathbf{x_j})\right)\phi_k(\mathbf{x_i})\right]$$

By moving inside the summation over k, we obtain the **kernel** function $k(\mathbf{x_i}, \mathbf{x_j}) = \sum_{k=1}^{M} \phi_k(\mathbf{x_i})\phi_k(\mathbf{x_j})$ and thus the result is

$$f(\mathbf{x_i}) = sign\left[\sum_{j=1}^{N} \alpha_j t_j k(\mathbf{x_i}, \mathbf{x_j})\right]$$

Observations

1. We have to be sure that the basis function exist, but we do not need to compute them as it is enough to compute the equivalent kernel. The difficulty has been now moved from the choice of the feature to the choice of the kernel.
2. We need to tune α_j: if it is different from 0, $\mathbf{x_j}$ is a support vector.
3. No closed-form solution. If the result is not good, the problem is in the choice of the kernel, not in the optimization method.
4. With the SVM (differently from kernel methods), the computation can be fast as only a subset of inputs is used for each prediction (the SV).

First the optimization will be done in the parametric case (**primal**), then it will be applied to the instance-based method (**dual**). Finally, we will consider the presence of noise in the data and how to tune the **bias-variance** trade-off. In order to use SVMs, need to choose:

1. The **kernel**: by experience (usually the gaussian kernel)
2. The subset of **samples**: it is done with the selection of the weights.
3. How to dimension the **weights**: maximize the margin, i.e. the points closest to the classifier have the maximum achievable distance (the classifier is more robust wrt noise)

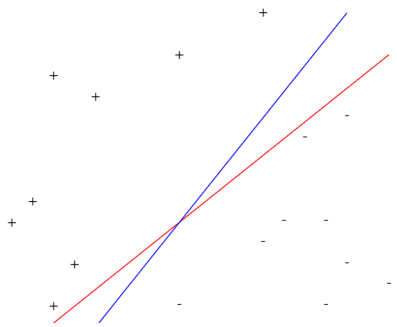


Figure 7.2: Two valid classifiers: the blue one is better because the margin (i.e. the minimum distance from a point of the dataset) is maximized.

7.3 Maximization of the margin

7.3.1 Preface: distance of a point from an hyperplane

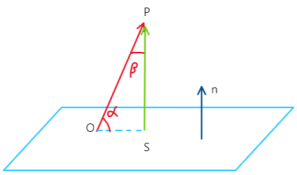


Figure 7.3: Distance between a point and a plane.

The distance **PS** of a point P from a plane is computed as

$$PS = OP\sin(\alpha) = OP\cos(\beta) = \vec{OP}\cdot\vec{n}$$

when $||\vec{n}|| = 1$ (i.e. $\vec{n}$ is the **versor** orthogonal to the plane). If $\vec{n}$ is not a versor, need do divide by its norm. The key idea is that the orientation of the plane does not change with the norm of n, as long as the versor remains the same. The norm of n is a degree of freedom that can be arbitrarily set with no loss of generality: because of this, versors are preferred (their norm is 1).

7.3.2 Parametric problem setting

The **margin** is the minimum distance between the points of the dataset and the boundary:

$$\textbf{margin} = \min_{n\in\mathcal{D}} t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b)$$

Notice that we assume that the boundary correctly classifies all the points, so that the factor t_n allows to have all positive distances; b instead gives the intercept of the boundary(the first element in ϕ is not 1). Given this, we want to find the maximum achievable margin:

$$\mathbf{w}^* = \arg\max_{\mathbf{w},b} \frac{1}{||\mathbf{w}||}\textbf{margin}$$

The norm of $\mathbf{w}$ has been added so that $\mathbf{w}$ is the **versor** orthogonal to the boundary hyperplane. Once $\mathbf{w}$ is found, the hyperplane is completely defined.

In order to simplify the problem, fix the minimum margin to 1 and transform the margin definition in a constraint.

$$\texttt{minimize} \qquad \frac{1}{2}||\mathbf{w}||^2$$

$$\texttt{subject to} \qquad t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b) \geq 1 \quad \forall n$$

We have M unknowns (one for each feature) and N constraints (one for each sample). The problem is well posed because the number of constraints is greater than the number of the unknowns. The problem is that the objective function is **not linear**. This is solved with the **Lagrange multiplier optimization**.

Example

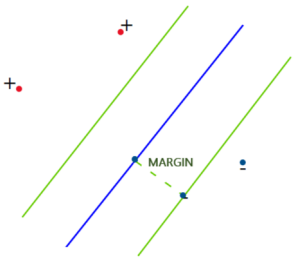


Figure 7.4: Margin representation.

Assume for example that the represented boundary is optimal with $||\mathbf{w}|| = 10$ and $\mathbf{w^T}\phi(\mathbf{x_n}) + b = -100$ in the nearest point; given $t_n = -1$, the margin will be $\frac{-1(-100)}{10} = 10$.Equivalently, we could obtain the same result by setting a priori $\min_{n\in\mathcal{D}} t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b) = 1$ and choosing $\mathbf{w}$ such that its norm is 0.1. This is just a scaling method that allows to avoid the problem of solving $\min_{n\in\mathcal{D}} t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b)$ and it does not change the solution since pre-setting the minimum value will change the parameter in order to have the same result.

7.3.3 The Lagrangian function

Given a generic minimization problem over a set of constraints

$$\texttt{minimize} \qquad f(\mathbf{w})$$
$$\texttt{subject to} \qquad h_i(\mathbf{w}) = 0 \forall i = 1, 2, ...$$

The optimal solution lies on the frontier of the constraints, i.e. on the space generated by $\nabla h_i() \forall i$. To solve this kind of problem, need to consider the **Lagrange function**

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

And solve $\nabla L = 0$. The number of equations is equal to the number of parameters + the number of constraints.

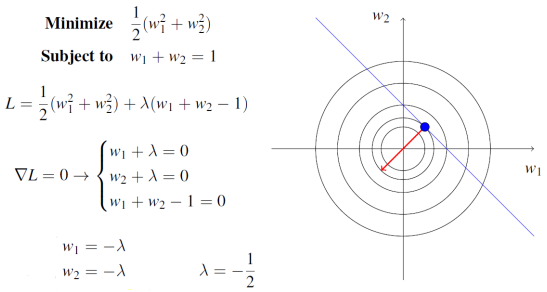


Figure 7.5: Example in 2D space; $f(\mathbf{w}) = \frac{1}{2}(w_1^2 + w_2^2)$ and there is only one constraint, i.e. $w_1 + w_2 - 1 = 0$. In this problem we want to find a point that belongs to a circle and to the constraint (i.e. the tangent point).

If some constraints are **inequalities**, we need to modify the Lagrange function as:

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

where the h constraints are equalities, while the g constraints are inequalities. It can be proved that the optimal solution verifies the(**KKT necessary conditions**:

1. The solution satisfies all the constraints (obvious, otherwise it would not be admissible).
2. The solution is a stationary point for the Lagrange function.
3. All the weight of the inequalities are either positive or zero.
4. For any inequality constraint, it is either active (the multiplier is positive, $g_i(\mathbf{w}) = 0$) or inactive ($\alpha_i = 0$, $g_i(\mathbf{w}) > 0$ or < 0).

The formulation and the conditions are expressed below:

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

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

Figure 7.6: Formulation with inequalities and KKT necessary conditions

If a constraint is not active (i.e. its weight is zero), the constraint is not important: it can be removed without changing the solution. This concept will be applied in order to define the support vector.

7.3.4 Primal representation

The **primal problem** for SVM is expressed by

$$\begin{aligned} \texttt{minimize} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \texttt{subject to} \quad & t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b) \geq 1 \quad \forall n \end{aligned}$$

and the unknown is $\mathbf{w}$. This is equivalent to

$$\begin{aligned} \texttt{minimize} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \texttt{subject to} \quad & -(t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b) - 1) \leq 0 \quad \forall n \end{aligned}$$

But this form allows to apply the Lagrangian multipliers when there are inequalities constraint (as described before). The Lagrangian function becomes

$$L(\mathbf{w}, \alpha, b) = \frac{1}{2}||\mathbf{w}||^2 - \sum_n \alpha_n(t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b) - 1)$$

The SVM quadratic problem is solved with the Lagrange multipliers in which each sample is associated to a constraint. The SVM builds the classifier basing only on samples related to active constraints, i.e. the ones with **positive weight**. These samples are called **support vectors**.

7.3.5 Dual Representation

From the Lagrangian function, we can write the **dual problem** wrt the **weights**, which is easier to solve. Once that the weights have been identified, it is not necessary to compute $\mathbf{w}$: this implies that the SVM (in its dual formulation) is a **non-parametric method**. In order to use the dual, we need to:

1. Compute ∇L wrt b and $\mathbf{w}$ and put it to 0 :

$$\frac{dL}{d\mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N} \alpha_n t_n \phi(\mathbf{x_n}) \quad \Longrightarrow \quad \mathbf{w} = \sum_{n=1}^{N} \alpha_n t_n \phi(\mathbf{x_n})$$

$$\frac{dL}{db} = -\sum_{n=1}^{N} \alpha_n t_n \quad \Longrightarrow \quad \sum_{n=1}^{N} \alpha_n t_n = 0$$

2. Substitute the found values in the Lagrangian function and making the problem become a **maximization** over $L(\alpha)$. The condition obtained from the computation of b becomes a constraint:

$$\begin{aligned} \texttt{maximize} \quad & \sum_{n=1}^{N} \alpha_n + \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x_n}, \mathbf{x_m}) \\ \texttt{subject to} \quad & \sum_{n=1}^{N} \alpha_n t_n = 0 \quad \alpha_n \geq 0 \qquad \forall n = 1...N \end{aligned}$$

3. Compute the values of α solving the quadratic problem

Once that α has been defined, the prediction is done with

$$y(\mathbf{x}) = sign\left[\sum_{j=1}^{N} \alpha_j t_j k(\mathbf{x}, \mathbf{x_j})\right]$$

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

where N_s is the number of support vectors (usually $N_s << N$). If the problem lies in high-dimension space, the number of support vectors that are needed becomes high.

7.3.6 Solution of the dual problem and noise robustness

The simplest technique used to solve the dual problem is the **SMO**, aka the **Sequential Minimal Optimization**: update one α_i at time, so that the problem becomes linear. The problem is that this approach violates the constraints and so we need to consider a **pair** of α_i, α_j at time:

1. Select the sample $\mathbf{x_i}$ that violates the most the KKT coditions
2. Select another random sample $\mathbf{x_j}$
3. Optimize jointly α_i, α_j
4. Repeat 1-2-3 until convergence

$$\begin{cases} \alpha_n = 0 & \texttt{the sample is not a SV, is correctly classified} \\ 0 < \alpha_n < C & \texttt{the sample is a SV and lies ON themargin} \\ \alpha_n = C & \texttt{the sample is a SV and lies INSIDE the margin} \end{cases}$$

Even with this algorithm, though, convergence is not ensured as a linear hyperplane that separates the classes may not exist. In order to make the model more flexible, we introduce some **slack variables** ξ_i: the constraint can now be violated by some samples, but in this case some penalty is added to the objective function. The primal formulation thus becomes:

$$\begin{aligned} \texttt{minimize} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N} \xi_i \\ \texttt{subject to} \quad & t_n(\mathbf{w^T}\phi(\mathbf{x_n}) + b) \geq 1 - \xi_i \quad \forall n \\ & \xi_i \geq 0 \quad \forall i = 1, 2, ... \end{aligned}$$

while the dual formulation becomes

$$\texttt{maximize} \quad \sum_{n=1}^{N} \alpha_n + \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m t_n t_m k(\mathbf{x_n}, \mathbf{x_m})$$

$$\texttt{subject to} \qquad \sum_{n=1}^{N} \alpha_n t_n = 0 \quad 0 \leq \alpha_n \leq C \qquad \forall n = 1...N$$

The parameter C is a constant used to tune the **bias-variance** trade-off:

1. If its value is high, the penalization is important to allow to handle samples that are far from their correct region (the model is more flexible): this choice increases the bias and reduces the variance.

2. If C is low, the penalization is not very important and thus the classifier should be as accurate as possible. This reduces the bias but increases the variance.

Substantially, if we want to decrease the variance we have to use a big penalty factor C. In case of good data and high precision, we can worsen the variance a little to reduce the bias by reducing C.

Part II

Reinforcement Learning

Chapter 8

What is RL

8.1 Setting

This part will describe the **Reinforcement Learning**, i.e. the study of how to maximize a cumulative reward on a decision problem (i.e. chess game, reach a goal with the minimum number of steps...). RL is used when the dynamics of the environment are unknown or when the model of the environment is too complex.

The setting of RL is made by an **agent** (learner) that observes the **environment** and, according to observation, he takes some **actions** on it, passing from one state to another. According to the action, the environment will give to the learner a certain **reward**. The agent must decide which is the optimal **policy** (i.e. strategy) to apply in order to maximize the overall reward.

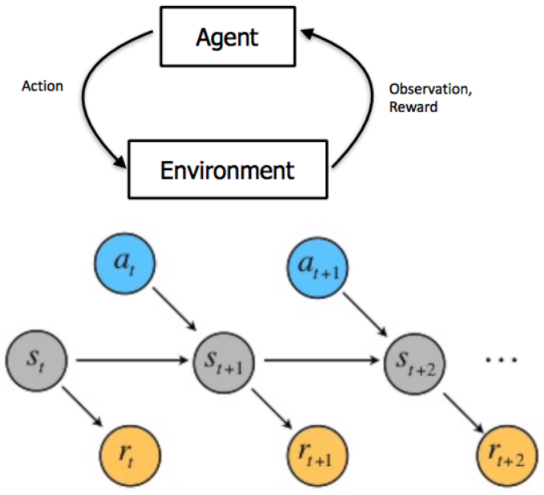


Figure 8.1: Interaction between agent and environment

8.2 Definitions

History, Exploration and Exploitation

According to the model of the interactions, at each time t:

- Learner receives some **observations** o_t
- Learner makes an **action** a_t
- Learner receives a **reward** r_t

The sequence of observations - action - reward in time builds the **history** h_t, which is used to decide the next action to perform. Differently from SL, the RL does not use training samples as it learns on the fly by taking different actions and examining the result. The **exploration** refers to the attempt of doing different actions in order to gain experience, while the **exploitation** refers to the choice of a particular strategy using the gained experience. RL must handle the trade-off between the exploration and exploitation.

States

Both the agent and the environment are characterised by a **state**:

- The **agent's state** is the set of all information needed to make a decision (besides the observation) and is a function of the history:

  $$s_t^a = f(h_t)$$

  The state of the agent can be **fully observable** (the state information is deterministic) or **partially observable** (the state information is only a *belief*, e.g. a robot position).

- the **environment's state** is the internal representation of the environment. Only in case of **fully observable** problems(e.g.chess game) $s_t^e = o_t$, otherwise the problem is**partially observable** (e.g. card game).

An **episode** allows to follow a path between the starting state and the terminal state.
A **terminal state** is a state for which the episode ends.

Discount Factor γ

The **discount factor** $\gamma \in [0, 1)$ represents both the probability that the game will go on after the current step and how much the reward for the next step is reduced. An high value of the the discount factor means that the decision is less critical and the time horizon is long; a small value indicates that the game will probably end and thus that the decision to take is more "critical". If $\gamma = 1$ the problem is **undiscounted**.

Reward

The **marginal reward** is the gain obtained at step t, while the **cumulative reward** is the reward cumulated from the beginning to the end; the latter can be expressed as:

- **Total reward**:
$$\sum_{t=1}^{\infty} r_t$$
- **Averaged reward**:
$$\lim_{n\to\infty} \frac{\sum_{t=1}^{\infty} r_t}{n}$$
- **Discounted reward**:
$$\sum_{t=1}^{\infty} \gamma^{t-1} r_t \ \ , \gamma \in [0, 1)$$

Time Horizon TH

The **time horizon** represents the expected duration of the problem until an ending state is reached. It can be:

- **Finite**: the number of steps is finite
- **Infinite**: the total reward cannot be used as the series would not converge. The discounted cumulative reward, instead, can be used as long as the value of $|r_t| < R$ are limited (allow the series to converge to $\frac{R}{1-\gamma}$)
- **Indefinite**: either finite or infinite, depends on the execution.

Return

The **return** v_t is the total discounted reward from time t on:

$$v_t = \sum_{k=0}^{\infty} \gamma^k r_{(t+1)+k}$$

Policy

The **policy** represents the *strategy* that the agent follows; in mathematical terms, it specify how much probable is to take each action a given a certain state s:

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

The policy can be:

- **Markovian** $\subseteq$ **History-Dependent** (i.e. depend on the present VS depend on past too)
- **Deterministic** $\subseteq$ **Stochastic** (i.e. $(s, a) \to s'$ VS $(s, a) \to P(s')$). With Stochastic policies we represent the state transition as $P(s'|s, a)$

- **Stationary** $\subseteq$ **Non-Stationary** (i.e. policy does not change VS the policy changes in time)

The objective of the agent is to **optimize the policy** in order to maximize the reward. There are two approaches:

1. **Model-based**: use a *model* of the environment to decide the probability to win choosing a certain sequence of states. Given the model, can use it to decide which is the most convenient policy.
2. **Model-free**: use a **trial-and-error** approach without knowing the environment (trial and error).

8.3 Classification of RL problems

The classification of the RL problems is done with:

- The number of agents (single-agent or multi-agent)
- The dimension of the state space
- The number of actions that can be performed at each instant t
- The type of transitions:
  - **stochastic**: given an action a that leads to s', there is a probability p that the next state will be s', while there is a probability *1-p* that some other state will be reached. In any case, any action can lead to different states with different probabilities.
  - **deterministic**: given an action a, it will be performed in the 100% of the cases.
- The representation of the **marginal reward** and of the **cumulative reward**.
- The **time horizon**
- The discount factor

Chapter 9

Markov Decision Process (MDP)

9.1 Definition

A **MDP** is a RL problem in which the **Markovian Assumption** is valid:

> *The future depends only on the present, not on the past. In other words, the effect of an action will depend only on the current state.*

In mathematical terms, given the state at time t X_t and the action performed at time t a_t

$$P(X_{t+1}|X_{1:t}, a_{1:t}) = P(X_{t+1}|X_t, a_t)$$

A MDP policy is **stationary**. For any MDP there is always an optimal policy which is Markovian, Stationary and Deterministic.

In mathematical terms, a MDP can be expressed as $< \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu >$:

1. $\mathcal{S}$ is the set of states
2. $\mathcal{A}$ is the set of actions
3. **P** is the state-transition matrix that specifies $P(s'|s, a)$. The matrix has $|\mathcal{S}|$ rows and, for each state, specify the probability of arriving in any other state *s'* given an action *a*. This means that the matrix has dimension $|\mathcal{S}| \times |\mathcal{A}||\mathcal{S}|$
4. **R** is the reward function $R(s, a)$
5. γ is the discount factor
6. μ specifies the initial probability for each state.

When an episode is played, given a MDP and a policy π:

- The sequence of taken states is called **Markov Process**
- The sequence of states-reward is called **Markov Reward Process** (or **Markov Chain**).

A MRP can be seen as $< \mathcal{S}, P^\pi, R^\pi, \gamma, \mu >$ where:

- P^π represents the state transition probability following the policy π; is built like a weighted average over $P(s'|s,a)$ in which the weights are given by the $\pi = P(a|s)$. The result is a matrix $|S| \times |S|$:

$$P^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) P(s'|a,s)$$

- R^π represents the average reward gained following policy π:

$$R^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) R(a,s)$$

In general, following policy π means making the average with $\pi(a|s)$.

9.2 Bellman Equations

Given a policy, there should be a method to evaluate the its goodness. The **policy evaluation** can be done by measuring the **value of a state s** with:

1. **State-Value function**: describes the expected *return* if, from time t on, the policy π is followed

$$\begin{aligned} V^\pi(s) &= E_\pi[v_t | s_t = s] \\ &= E_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{(t+2)+k} | s_t = s] \\ &= E_\pi[r_{t+1} | s_t = s] + \gamma E_\pi[v_{t+1} | s_t = s] \\ &= E_\pi[r_{t+1} | s_t = s] + \gamma E_\pi[\sum_{s' \in S} P(s'|s,a) V^\pi(s')] \end{aligned}$$

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

The first term represents the expected reward gained from taking the action a, while the second term represents the average discounted value of the next state s', considering the state transition probability. In the end, policy π is applied.

In other words:

(a) Choose one action a between all the possible action.

(b) Compute the expected value gained by taking it as

$$V' = \sum_{s' \in S} P(s'|s,a)V^{\pi}(s')$$

(c) Compute the reward $R(s,a)$ obtained by doing the action a

(d) Sum the two terms $R(s,a) + \gamma V'$ and weight it with $\pi(a|s)$, which represents the probability of taking the action a from the current state s.

(e) Repeat $a - d$ for all possible actions and sum everything.

Given the policy π it is **exploited** since the first action from time t on.

2. **State-Action function**: describes the expected *return* if, **after** taking an action a, the agent follows the policy π:

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

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

By definition, $V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s)Q^{\pi}(s,a)$ because it means that also the action a follows the policy π.

$$Q^{\pi}(s,a) = R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s')Q^{\pi}(s',a')$$

Given the policy π, the next action is used for **exploration**, then **exploitation** is applied until the end.

The State-Action function represents the **quality Q** of an action, given the starting state, and it can be implemented as a matrix.

State \ Action	0	1	2	3	4	5
0	−1	−1	−1	−1	0	−1
1	−1	−1	−1	0	−1	100
2	−1	−1	−1	0	−1	−1
3	−1	0	0	−1	0	−1
4	0	−1	−1	0	−1	100
5	−1	0	−1	−1	0	100

Figure 9.1: An example of a matrix implementation of Q

The state-value function is called also **Bellman Expectation Equation**, in which the unknown is V^π, i.e. the values of the states in a certain instant t. The matrix form of the Bellman equation is:

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$

whose solution is

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

The solution exists unique if $\gamma \in [0, 1)$. If $\gamma = 1$, the inversion is not possible as at least one eigenvalue of $I - \gamma P^\pi$ will be 0 and so its determinant will be 0.

9.2.1 Bellman Operator T^π

The state-value functions and state-action functions can be rewritten by renaming the right-hand side of the equation as a new function, called **Bellman operator**:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \right) \iff T_V^\pi(V^\pi) = V^\pi$$

$$Q^\pi(s,a) = R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s',a') \iff T_Q^\pi(Q^\pi) = Q^\pi$$

This form says that V^π and Q^π are **fixed points** for the Bellman Operator: thanks to this, starting from a state **s** and applying the operator iteratively, the fixed point is reached.

9.3 Optimality

For any state **s**, the **optimal value** is the maximum value of V^π (or Q^π):

$$V^*(s) = max_\pi V^\pi(s) \quad Q^*(s,a) = max_\pi Q^\pi(s,a)$$

The **optimal policy** is the one that **maximizes the value of all the states** π^*.

9.3.1 Theorem

For any MDP:

- At least one optimal policy $\pi^* \geq \pi \, \forall \pi$ always exists
- All optimal policy go through V^*, Q^* , i.e. $V^{(\pi^*)} = V^*$ and $Q^{(\pi^*)} = Q^*$
- At least one between all the optimal policies is **deterministic** and can be found by taking, at each step, the action a that maximises Q^*:

$$\pi^*(a|s) = \begin{cases} 1 & a = arg\, max_a Q^* \\ 0 & otherwise \end{cases}$$

9.3.2 Bellman Optimality Equation

The optimal policies can be found as **solutions of the Bellman Optimality Equation**:

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right\} \iff T_V^*(V^*) = V^*$$

$$Q^*(s,a) = R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a') \iff T_Q^*(Q^*) = Q^*$$

No policy is applied here because the optimality equation simply maximizes the values of all the states. When this has been done, need to find the **optimal policy**,i.e. the one that builds a path between the states in order to maximize the overall reward.

These equations are **non-linear** because of the **max** function and thus no closed-form solution exists. To solve these equations there are different methods:

- **Model-Based methods** $\implies$ Policy Iteration, Dynamic Programming, Linear Programming
- **Model-Free methods** $\implies$ MonteCarlo methods, Temporal Difference

9.3.3 Insight on Model-Free approaches

In **model-free learning**, the state-value function for a **greedy policy** can be rewritten as

$$V(s,a) = r + \gamma \max_{s'} V(s')$$

while the Q function for the greedy policy becomes

$$Q(s,a) = r + \gamma \max_{a} Q(s',a)$$

With these approaches, it is often used the ε**-greedy policy** explores new actions with probability ε and exploits the best strategy known with probability $1-\varepsilon$. If $\varepsilon = 0$, the policy becomes *pure exploitation* (**Greedy policy**), in which the agent follows the states with maximum Q values; if $\varepsilon = 1$ it becomes *pure exploration*. Usually, $\varepsilon = 0.1$ is used.

Chapter 10

Model-based methods to solve MDP

10.1 Policy Search for TH $< \infty$

The most basic algorithm that evaluates all the policies and then picks the best one. Unfortunately, the complexity is $|\mathcal{A}|^{|\mathcal{S}|}$.

10.2 Dynamic Programming for TH $\leq \infty$

10.2.1 Introduction

This method can be applied in lots of contexts (beside RL), typically with recursion, and is based on:

- Splitting the problem into smaller subproblems
- Caching the result of subproblems in order to speed the performances

This allows to transform the complexity from **exponential** to **polynomial**. As regards RL, Dynamic Programming requires a **full knowledge** of the MDP (i.e. states, actions, state transitions and reward). The DP allows to solve two problems:

- **Prediction**: from MDP and a policy $\pi \implies V^{\pi}$
- **Control**: from MDP $\implies V^{*}, \pi^{*}$

The dynamic programming approach depends on the time horizon:

- **Finite** time horizon $\implies$ **Backward Recursion**: start from the terminating state (with $V*_N = 0$) and backtrace choosing the action that leads to the state with the highest value.
- **Infinite** time horizon $\implies$ **Policy Iteration** (made by evaluation + improvement done alternatively), **Value Iteration**:
  1. **Prediction problem**: iteratively apply the policy evaluation
  2. **Control problem**: apply value iteration or policy iteration.

10.2.2 Backward Recursion

The Backward recursion solves the **control problem** starting from a MDP with finite TH (N steps). To **find** $V^*(s)$the idea is to start from the last step (which has $V^*_{N-1}(s) = 0$) and iterate the update of $V^*_k(s)$ for N times (up to $V^*_0(s)$) assigning at each step the result of

$$V^*_k(s) = \max_{a \in \mathcal{A}} \left\{ R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) V^*_{k+1}(s') \right\}$$

In order to **find the optimal policy** we take the actions that allow to maximize $V^*(s)$ in each state, i.e.

$$\pi^*(s) = \max_{a \in \mathcal{A}} \left\{ R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right\}$$

The cost of this approach is $|N||\mathcal{A}||\mathcal{S}|$.

Example

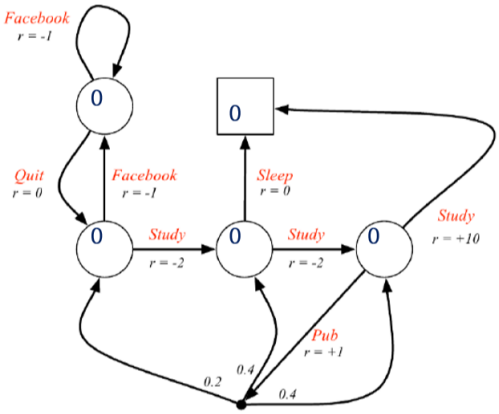


Figure 10.1: Initial Setting : all the states have value = 0 , the square is the terminating state, whose value will not be updated. The dot represents an initial state and specifies the probability with which the agent can go to state B,C,D. Its value is considered as 0 for all the process.

In the first iteration, the values are updated as follows in order D-C-B-A starting from the terminal state. As regards $P(s'|a,s)$, it will be considered as 1 for all the actions (i.e. if an action a that links s to s' is taken, with 100% probability s' will be reached). At each state, will be considered only the outgoing actions and $\gamma = 1$:

- $V^*(D) = \max\{10 + 1 \cdot 0; 1 + 1 \cdot 0\} = 10$
- $V^*(C) = \max\{0 + 1 \cdot 0; -2 + 1 \cdot 0\} = 0$
- $V^*(B) = \max\{-2 + 1 \cdot 0; -1 + 1 \cdot 0\} = -1$

- $V^*(A) = \max\{0 + 1 \cdot 0; -1 + 1 \cdot 0\} = 0$

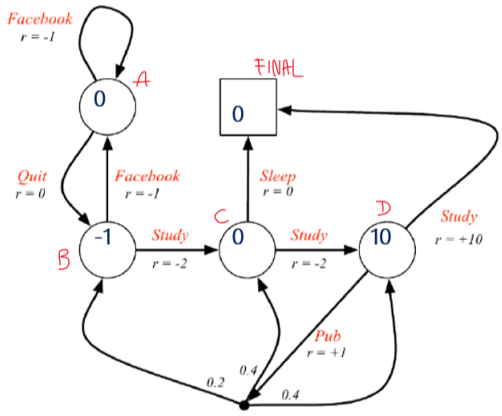


Figure 10.2: States after the first iteration

In the second iteration:

- $V^*(D) = \max\{10 + 1 \cdot 0; 1 + 1 \cdot 0\} = 10$
- $V^*(C) = \max\{0 + 1 \cdot 0; -2 + 1 \cdot 10\} = 8$
- $V^*(B) = \max\{-2 + 1 \cdot 0; -1 + 1 \cdot 0\} = -1$
- $V^*(A) = \max\{0 + 1 \cdot (-1); -1 + 1 \cdot 0\} = 0$

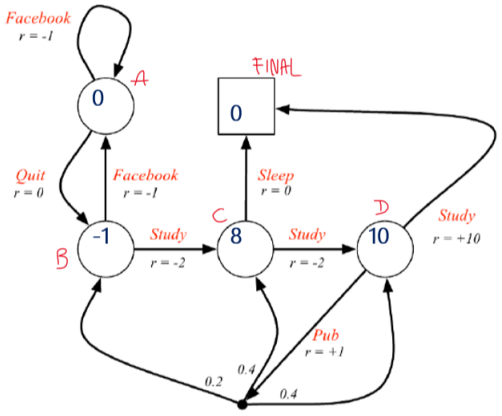


Figure 10.3: States after the second iteration

In the third iteration:

- $V^*(D) = \max\{10 + 1 \cdot 0; 1 + 1 \cdot 0\} = 10$

- $V^*(C) = \max\{0 + 1 \cdot 0; -2 + 1 \cdot 10\} = 8$
- $V^*(B) = \max\{-2 + 1 \cdot 8; -1 + 1 \cdot 0\} = 6$
- $V^*(A) = \max\{0 + 1 \cdot (-1); -1 + 1 \cdot 0\} = 0$

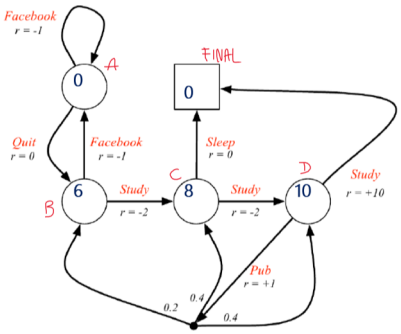


Figure 10.4: States after the third iteration

In the fourth iteration:

- $V^*(D) = \max\{10 + 1 \cdot 0; 1 + 1 \cdot 0\} = 10$
- $V^*(C) = \max\{0 + 1 \cdot 0; -2 + 1 \cdot 10\} = 8$
- $V^*(B) = \max\{-2 + 1 \cdot 8; -1 + 1 \cdot 0\} = 6$
- $V^*(A) = \max\{0 + 1 \cdot 6; -1 + 1 \cdot 0\} = 6$

This configuration is the optimal one because the value of the states will not change any more. The **optimal policy** is the set of arcs that have been followed in order to set the maximum values of the states. In the example the policy has been shown only in the last case, but it can be computed after each iteration in the same way.

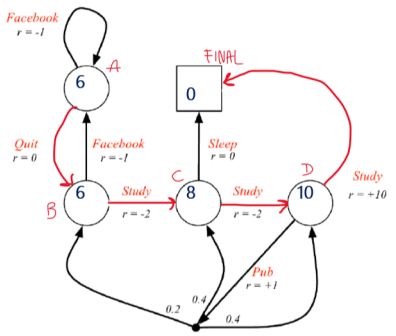


Figure 10.5: States after the fourth iteration (optimality). The optimal policy is shown in red.

10.2.3 Policy Iteration

Policy iteration is an approach that can be split in :

1. **Evaluation**: Initially assign a value for $V(s)$. Then apply several times the **Bellman operator** $T_V^\pi()$. Given that $V^\pi(s)$ is its only fixed point, if T_V^π is applied several times, in the end $V(s) \to V^\pi(s)$. The convergence verifies only for ∞ iterations, thus a **stopping criterion** is needed: when

$$\max_{s \in \mathcal{S}} |V^{(k+1)}(s) - V^{(k)}(s)| \leq \varepsilon$$

   the convergence is considered to be reached.

2. **Improvement**: After an evaluation, in all states check if some other action can perform better than the one advised by the policy (through higher probability). If so, the policy is updated. It can be proved that this method always improves the policy, until the **optimal** policy is reached

The **evaluation phase** can be used as stand-alone to find, given a policy π, the values $V^\pi(s)$ (prediction problem). If alternated with the **improvement** (i.e. one iteration in evaluation, one iteration for the improvement and so on), it can be used to find the **optimal values** $V^*(s)$ and the optimal policy $\pi^*(a|s)$.

Example

Consider a small gridworld with 14 cells and one terminal state (doubled in the grid, represented with grey colour). The discount factor is $\gamma = 1$ and in each cell there are 4 possible actions $(\uparrow, \downarrow, \leftarrow, \rightarrow)$. At the beginning,

$$V(s) = 0 \quad \forall s \in \mathcal{S}$$

$$\pi(\uparrow |s) = \pi(\downarrow |s) = \pi(\leftarrow |s) = \pi(\rightarrow |s) = 0.25 \quad \forall s \in \mathcal{S}$$

π is thus the **random policy**. The goal is to reach the terminal states (starting from 1-14) in the minimum number of steps (each one has a reward $= -1$):

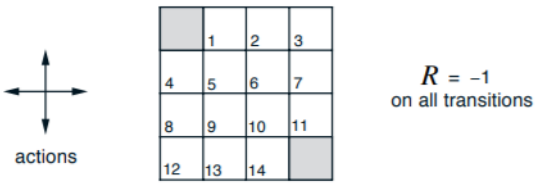


Figure 10.6: 4×4 gridworld setting

As previously said, at the beginning all the values are 0 and all the actions are equally probable (first row in the figure below). After the first iteration, all the values are equal to -1 because, from each cell, only one step is needed to reach a state with null value. In the improvement phase, for cells $1 - 4 - 11 - 14$ the policy has been updated: for each of this cell, the remaining arrow indicates the adjacent cell with the lowest value. The other cells' policy remain unchanged as

all the adjacent cells have values -1. After the second evaluation, the policy of all cells except for $3-6-9-12$ have been updated. After the third evaluation the optimal policy is reached (as it won't change any more), while the values $V(s)$ are not optimal yet. From this example we can se that the **policy converges faster than the values**.

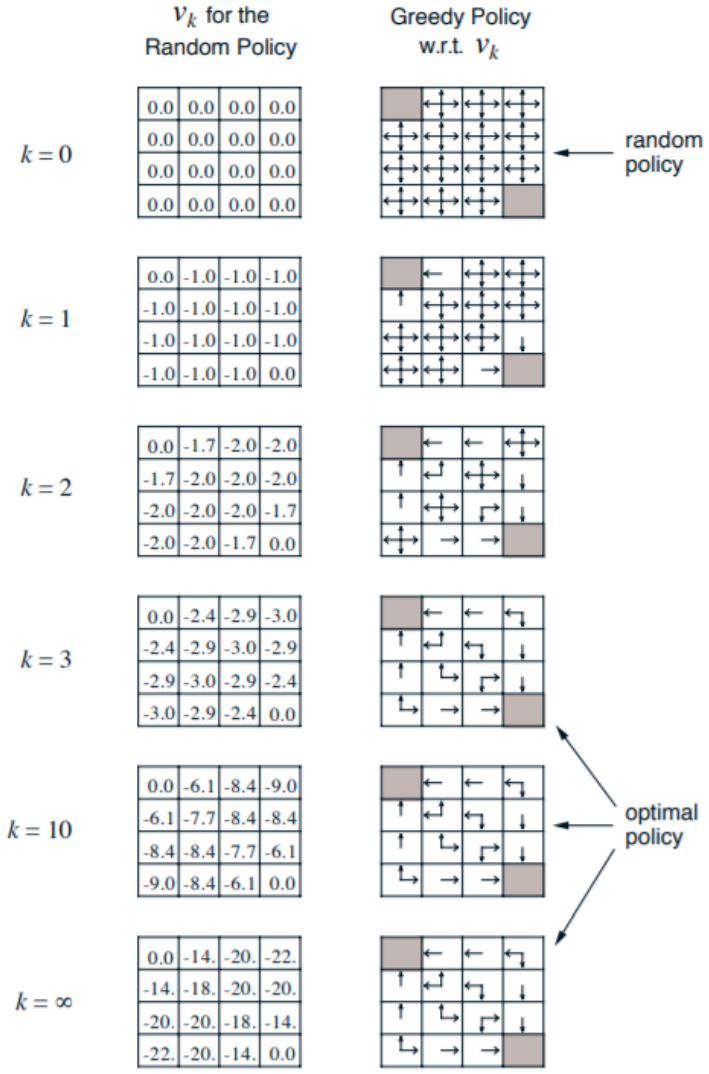


Figure 10.7: **Policy Iteration**: looking only at the left column, there is the **policy evaluation** and in the last row there are the values for $V^{\pi}(s)$ of the random policy. Looking at the right column only, there is the **policy improvement**. The whole algorithm can be read from top to bottom, one row at time

10.2.4 Value Iteration

Similar to policy evaluation, but here the Bellman **optimality** operator is used, i.e. $T_V^*()$, for which $V^*(s)$ is the only fixed point. Normally, the policy iteration is computationally more demanding wrt the value iteration, but it converges in less iterations.

10.3 Linear Programming for TH = ∞

10.3.1 Primal

Linear programming can be used with problems with **infinite time horizon**. In the control problem, the optimal value $V^*(s)$ is the solution of the Bellman optimality equation, i.e.

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ R(a,s) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right\}$$

This objective function, however, is **non-linear** wrt the action. The same thing can be obtained by reformulating the problem in the following way:

$$min_V \sum_{s \in \mathcal{S}} \mu(s) V(s)$$

$$\texttt{s.t.} \quad V(s) \geq R(a,s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \qquad \forall s \in \mathcal{S}, \forall a in \mathcal{A}$$

What is minimized here is the **expected utility**, i.e. the average **V(s)** weighted by the initial probability $\mu(s)$. This formulation thus states that the **optimal** value function is the smallest one that satisfies all the constraints and the formulation is **linear** wrt the value function. The problem is well-posed as the number of unknowns is $|\mathcal{S}|$ and there is a constraint for each state-action pair (so the number of constraints is greater than the number of unknowns). As regards the efficiency, the linear-programming **worst case** is faster than the DP worst case but, on **average**, **DP is more efficient.**

10.3.2 Dual

The primal problem can be in turn reformulated as

$$max_\lambda \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \lambda(s,a) R(s,a)$$

$$\texttt{s.t.} \quad \sum_{a' \in \mathcal{A}} \lambda(s',a') = \mu(s) + \gamma \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \lambda(s,a) P(s'|s,a) \quad \forall s' \in \mathcal{S}$$

$$\lambda(s,a) \geq 0 \quad \forall s \in \mathcal{S}, \forall a in \mathcal{A}$$

where λ represents the (discounted) number of times that we expect to visit the state. The result is the maximization of $\texttt{immediate}_\texttt{r}\texttt{eward} \cdot \texttt{number}_\texttt{o}\texttt{f}_\texttt{v}\texttt{isits}$ and the optimal value of λ represents, for each state, the number of necessary visits in order to maximize the performances. In this formulation, the value of $\lambda(s,a) = 0$ for the actions that are sub-optimal in the state s, while $\lambda(s,a) > 0$ for optimal actions. Given this, thanks to λ we find the optimal policy.

Chapter 11

Model-free methods to solve MDP

11.1 Outline

This methods are applied when the MDP is unknown:

1. **Model-free Prediction**: estimate $V^{\pi}(s) \implies$ MonteCarlo , Temporal Difference, n-step TD
2. **Model-free Control**: find $V^{*}and\pi^{*} \implies$ SARSA, Q-learning

Model-free learning is very useful when the system is too complex to be modelled (e.g. bot that learns a strategy to win in online battle games or to play chess should take into account too many states and the model would be too big). This means that the transition probabilities are not known.

Model-free learning means **to learn from experience**, literally: in DP problems, instead, there was not necessary to do attempts to estimate the best strategy, as the model was completely defined and the optimal policy could be precomputed. In model-free learning e.g. for chess, may need to play some games and get a positive/negative reward for winning/losing and use the experience to elaborate the best policy. If the learner needs to use complete episodes (e.g. full plays of chess), we consider **Monte Carlo**, whereas if the learner can use incomplete episodes, we consider **Temporal Difference**.

Examples on TD, Q-learning, SARSA
https://www.analyticsvidhya.com/blog/2019/03/reinforcement-learning-temporal-difference-learning/

Examples on Monte Carlo
https://www.analyticsvidhya.com/blog/2018/11/reinforcement-learning-introduction-monte-carlo-learning-openai-gym/

11.2 Prediction

11.2.1 Monte-Carlo Methods

The main characteristics are:

1. Need **complete** episodes of experience (i.e. in which terminal state is reached)
2. Update Values estimates and policy only at the end of the episode.
3. Estimates of every episodes are **independent** $\rightarrow$ no bootstrapping

The MC methods exploit the **empirical mean** of the values which is updated incrementally:

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

The Value can be updated on:

1. **First Visit** on s : unbiased estimator
   (a) Initialize the policy, state-value function
   (b) Generate an episode according to the policy, keeping track of the states encountered through that episode
   (c) For every state:
      i. If it's its first occurrence in the episode, remember the return
      ii. If not, do not remember the return
   (d) For every state, update the value $V(s)$ using the **empirical mean** of the returns.
   (e) Repeat 2-4 for every episode, until satisfied

   $V(s)$ is estimated as the empirical mean of the returns gained the first time the state was encountered in each episode (e.g. 4 episodes $\implies$ 4 returns if the state was encountered at least once in every episode).
2. **Every-visit** on s : biased estimator, but consistent. Same as before, but keep track of the return **any time** the state is visited(e.g. 4 episodes $\implies$ 8 returns if the state was encountered twice in every episode).

Example

Consider a problem with 2 states, A and B and $\gamma = 1$. Two episodes are watched:

$$A \rightarrow^{3} A \rightarrow^{2} B \rightarrow^{-4} A \rightarrow^{4} B \rightarrow^{-3} \rightarrow end$$
$$B \rightarrow^{-2} A \rightarrow^{3} B \rightarrow^{-3} \rightarrow end$$

where $A \rightarrow^{3} A$ means that state goes from A to A with reward = 3. The **return** is $v_t = \sum_{k=0}^{\infty} r_{(t+1)+k}$. Given this:

1. **First visit**: take into account the return for A and B only the first time they are encountered in the two episodes (for A at step 1 and 2, for B at step 3 and 1).

$$v_A(1) = 3 + 2 - 4 + 4 - 3 = 2 \qquad v_A(2) = 3 + 3 = 0$$

$$v_B(3) = -4 + 4 - 3 = -3 \qquad v_B(1) = -2 + 3 - 3 = -2$$

   Given this returns, the values of the states are:

$$V(A) = \frac{1}{2}(2 + 0) = 1 \qquad V(B) = \frac{1}{2}(-3 - 2) = -2.5$$

2. **Every visit**: compute the return for A,B as above every time they are encountered in both episodes, then make the average to find the values of the states:

$$V(A) = \frac{1}{4}(2 - 1 + 1 + 0) = 0.5 \qquad V(B) = \frac{1}{4}(-3 - 3 - 2 - 3) = -\frac{11}{4}$$

In case of **non-stationary problems**, instead of the empirical mean can use the **running mean**:

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t))$$

where α represents the weight of the correction factor. May not converge!

11.2.2 Temporal Difference

The main characteristics are:

1. Use **incomplete** episodes
2. No transition probability required

The TD update rule is the following:

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

where α is the learning rate and the term between ( ) represents the **TD error** δ_t (seen as an error because it's the difference between the actual reward and the expected reward). In other words, the value is updated "on the fly" after the transition to the next state is completed. The steps involved in TD prediction are:

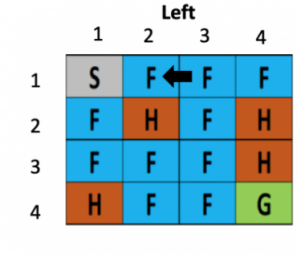
1. Initialize V(s) (e.g. all to 0) and begin an episode.
2. At every step in which
   (a) the learner is in state s and performs an action a, arriving in s' and receiving a reward r
   (b) After arriving in s', update s with the rule:

$$V(s)_{new} = V(s)_{old} + \alpha(r + \gamma V(s')_{old} - V(s)_{old})$$

3. Repeat 2 until terminal state is reached.

Example

Given the frozen lake problem, $\gamma = 0.5$,$\alpha = 0.1$, with 16 states initially initialized to 0; suppose to have (after 2 iterations) the following situation, being in state (1,3):



State	Value
(1,1)	-0.04
(1,2)	-0.04
(1,3)	0
...	0
(4,4)	0

Figure 11.1: Situation after 2 iterations

The learner decides to go left, as shown in the figure. Data:

1. $s = (1, 3)$
2. s'$= (1, 2)$
3. $a =$ go left , with reward $r = -0.4$

After the action is performed, $V(s = (1, 3))$ is updated as follows:

$$V(s) = 0 + 0.1(-0.4 + 0.5(-0.04) - 0) = -0.042$$

n-step Temporal Difference

The return v_t takes into account the following n time instants, instead of just the immediate successive one:

$$TD_n = \begin{cases} n = 1(TD) & v_t^{(1)} = r_{t+1} + \gamma V(s_{t+1}) \\ n = 2 & v_t^{(2)} = r_{t+1} + \gamma V(s_{t+1}) + \gamma^2 V(s_{t+2}) \\ ... & ... \\ n = \infty(MC) & v_t^{(\infty)} = r_{t+1} + \gamma V(s_{t+1}) + ... + \gamma^{T-1} V(s_T) \end{cases}$$

The *n-th* TD step is

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

and the return $v_t^{(n)}$ looks forward by n time instants. The $\lambda - return$ gives a weighted sum of all n-step returns, i.e.

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

but episodes need to be complete in this case (otherwise it cannot be computed). To overcome this need, the **Backward TD(λ)** is introduced.

Backward TD(λ)

Advantages and characteristics:

1. Applicable also on incomplete episodes
2. Computation distributed in time instead of at the end of the episode
3. Uses a short-term memory vector called **eligibility trace**:

$$\mathbf{e_t} = \begin{bmatrix} e_t(s_1) & e_t(s_2) & ... & t_t(s_n) \end{bmatrix}$$

The eligibility trace vector has one entry for each state and refers to one time instant. For each state s and time t, $e_t(s)$ describes if the state was reached at that time instant or not with a function

$$e_0(s) = 0 \quad \forall s$$
$$e_t(s) = \gamma\lambda e_{t-1}(s) + 1_{s=s_t}$$

Initially, all the traces are set to 0. When a state is reached, its corresponding trace is bumped up by 1, otherwise is value is reduced by a factor $\lambda\gamma$, where λ is the **trace-decay** parameter:

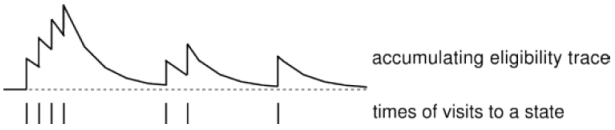


Figure 11.2: Evolution in time of $e_t(s)$ for a given state s

Given the learning rate α, the Value of the state is updated as

$$V(s) \leftarrow V(s) + \alpha\delta_t e_t(s)$$

Particular cases:

1. $\lambda = 0$: $e_t(s)$ is a square wave between 0 and 1, without period (as states are not traversed periodically). If state is not selected, $V(s)$ remains unchanged, otherwise it is incremented by $\alpha\delta_t$. This corresponds to the basic TD algorithm.
2. $\lambda = 1$: roughly equivalent to every-visit MC as, with steps going over, the sum of the errors become:

$$\delta_t + \gamma\delta_{t+1} + ... + \gamma^{T-t}\delta_{T-1} =$$
$$(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) +$$
$$(\gamma r_{t+2} + \gamma^2 V(s_{t+2}) - \gamma V(s_{t+1})) +$$
$$...$$
$$(\gamma^{T-1} r_{t+T} + \gamma^T V(s_{t+T}) - \gamma^{T-1} V(s_{t+T})) =$$

$$r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-1} r_{t+t} - V(s_t) = v_t - V(s_t)$$

With frequently accessed states, the eligibility trace can be greater than 1. To solve this, the replacing trace is used:

$$e_t(s) = \begin{cases} 1 & s_t = s \\ \gamma\lambda e_{t-1}(s) & otherwise \end{cases}$$

11.3 Control

11.3.1 Monte Carlo

MC control is an **on-policy** algorithm which substantially adapts policy iteration using $Q(s,a)$ instead of $V(s)$ as the transition probabilities are not known. Thus, the policy is updated with

$$\pi'(s) = \texttt{argmax}_a Q(s,a)$$

In order to avoid having unexplored actions, exploration must be taken with some probability. This means that, given m action and an $\varepsilon - greedy$ policy in which all the action have the same probability $\frac{\varepsilon}{m}$, except for the one with highest Q value which has a much higher probability (complementary event)

$$\pi(s,a) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{m} & ona* = \texttt{argmax}_a Q(s,a) \\ \frac{\varepsilon}{m} & onallotheractions \end{cases}$$

MC control algorithm is a **GLIE** algorithm (Greedy in the Limit of Infinite Exploration), i.e. satisfies the following properties:

1. For every state, every action is explored an infinite number of times
2. The policy converges to a greedy policy

This method works only on **complete episodes**. The steps of the algorithm are:

1. Choose the k-th episode
2. For every (s_t, a_t) in the episode (policy evaluation):
   (a) $N(s_t, a_t) += 1$
   (b) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)}(v_t - Q(s_t, a_t))$
3. Update the policy $\pi \leftarrow \varepsilon - greedy(Q)$ (policy improvement)
4. Update $\varepsilon = \frac{1}{k}$

11.3.2 Temporal difference

The TD control algorithms can be classified in 2 categories:

1. **Off-policy**: learn a policy with experience of another one (Q-Learning)
2. **On-policy**: learn a policy with experience of that policy (SARSA)

Q-Learning

The **Q-Learning** control algorithm has the following characteristics:

- **Off-policy**
- No transition probability model
- Uses state-action function $Q(s, a)$ in the update rule, updating Q^π with the experience from the **greedy policy** (i.e. choose the action with the action with the highest Q at each step).

The algorithm is the following:

1. The Q function is initialized randomly.
2. Start an episode.
3. For each step in the episode:
   (a) Set the value of ε and extract a random number:
      - **Explore**: $\mathbf{r} > \varepsilon$. One of the actions a is selected.
      - **Exploit**: $\mathbf{r} \leq \varepsilon$. The agent is in state s, he looks at the actions (row of state S in matrix Q) and selects the action a that gives the maximum value of Q(S,a).

   (b) The agent performs the action.
   (c) The agent updates the Q values using the update rule:

$$Q(s, a)_{new} = Q(s, a)_{old} + \alpha(r + \gamma \max_{a'} Q(s_{new}, a') - Q(s, a)_{old})$$

4. Repeat 3 until a termination state is reached. If necessary, start a new episode.

Observations:

- Q is updated with a variation weighted by the learning rate α. The variation includes the difference between (immediate reward + discounted Q value) and the previous value of Q.
- The learning rate is in $[0, 1]$: the higher, the more quickly the more the Q values will change at each iteration.
- At the first episodes, the learner must explore so $\varepsilon = 1$. As steps goes on, more information are gathered, so ε becomes lower (e.g. $\varepsilon = 0.1$).
- During **exploration**, the lizard in state S firstly chooses the action A to explore, secondly updates the value of Q(S,A) in the Q table. During **exploitation**, instead, the action is chosen with the max value in the Q matrix (looking only at the row of the current state).

Example of Q-learning

Setting:
Consider a grid where a lizard moves. If it meets a bird, the reward is -10 and the episode; if it meets 5 crickets, the reward is $+10$ and the episode ends; if it meets 1 cricket, the reward is $+1$, while on empty cells the reward is -1. At the beginning, the Q values are all equal to 0. The learning rate is $\alpha = 0.7$, while $\gamma = 0.99$.

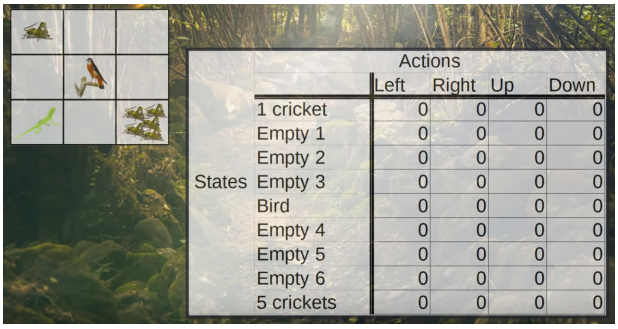


Figure 11.3: Q-Learning algorithm, 1st episode

Consider a **first episode:**

1. Initially, the lizard must explore as all q values are 0, thus $\varepsilon = 1$. Suppose the lizard goes right (to cell Empty 6), which gives a reward of -1. The Q-value of the starting cell (Empty 5) will now be

$$Q(\texttt{e5}, \texttt{right}) = 0 + 0.7(-1 + \gamma \max_a Q(\texttt{e6}, a) - 0) = 0.7(-1 + \gamma 0 - 0) = -0.7$$

This is because the max function selects the highest Q value on the line corresponding to state **e6**: given that all the values are 0, the function returns 0.

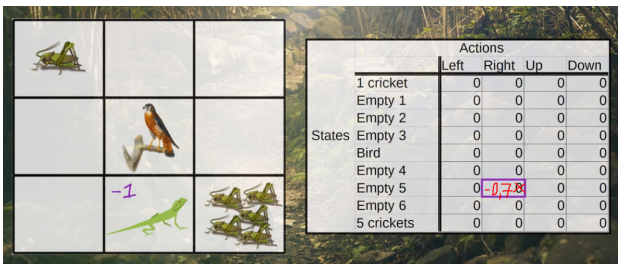


Figure 11.4: Q-Learning algorithm, 1st episode: situation after the first move

2. Suppose that at the next iteration the lizard explores going right again. The action gives a reward of $+10$. The Q value of **e6** is now updated to

$$Q(\texttt{e6}, \texttt{right}) = 0+0.7(+10+\gamma \max_{a'} Q(\texttt{5crick}, a')-0) = 0+0.7(+10+\gamma 0-0) = 7$$

3. The lizard has reached a terminal state and thus the episode terminates.

Consider a **second episode:**

1. The lizard starts exploring again going right into **e6** with the Q function obtained after the first episode. The Q value of **e5** becomes

$$Q(\texttt{e5}, \texttt{right}) = -0.7 + 0.7(-1 + 0.99 \cdot 7 - (-0.7)) = 3.941$$

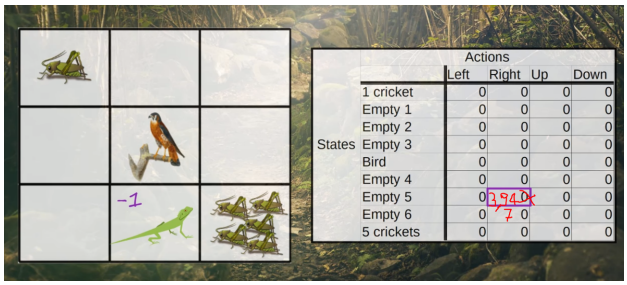


Figure 11.5: Q-Learning algorithm, 2nd episode: situation after the exploration

2. The lizard chooses to exploit. At this point, it can go either to the bird, to **e5** or to **e6**. The lizard in **e6** analyses the actions it can take (looking at the row of e6): the one with the highest Q value is the action `right`, thus the lizard chooses it. The update of Q value is :

$$Q(\texttt{e6}, \texttt{right}) = 7 + 0.7(+10 + 0.99 \cdot 0 - 7) = 9.1$$

3. The lizard has reached a terminal state and thus the episode terminates.

11.3.3 SARSA

The **SARSA** control algorithm has the following characteristics:

- **On-policy**
- No transition probability model
- Uses state-action function $Q(s, a)$ in the update rule, updating Q^π with the experience from the **greedy policy** (i.e. choose the action with the action with the highest Q at each step)

which are the same of Q-Learning. However, the update rule does not consider the $\max_{a'} Q(s', a')$: the action a' will be chosen again with the $\varepsilon - greedy$ policy (actually, SARSA stays for State Action Reward State Action):

- In case of **exploitation**, choose in s' the action that maximizes the Q value.

- In case of **exploration**, choose in s' the action that is needed to be explored.

Summarizing, the algorithm becomes:

1. The Q function is initialized randomly.
2. Start an episode.
3. For each step in the episode:
   (a) Set the value of ε and extract a random number:
      - **Explore**: $\mathbf{r} > \varepsilon$. One of the actions a is selected.
      - **Exploit**: $\mathbf{r} \leq \varepsilon$. The agent is in state s, he looks at the actions (row of state S in matrix Q) and selects the action a that gives the maximum value of Q(S,a).

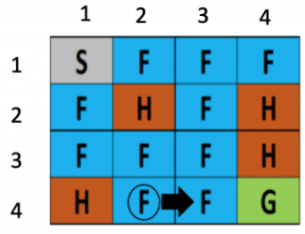
   (b) The agent performs the action. and observes the return r and the new state s'.
   (c) The agent updates the Q values using the update rule:

$$Q(s,a)_{new} = Q(s,a)_{old} + \alpha(r + \gamma Q(s',a') - Q(s,a)_{old})$$

   where a' is choosing again with an $\varepsilon-$ greedy policy.
   (d) In the next iteration, update Q(s',a').
4. Repeat 3 until a termination state is reached. If necessary, start a new episode.

Example of SARSA algorithm

Suppose being in the grid-world 4×4 of the frozen lake, in state (4,2). Consider $\gamma = 1, \alpha = 0.1$:



State	Action	Value
(4,2)	Up	0.2
(4,2)	Down	0.4
(4,2)	Right	0.6
(4,3)	Up	0.2
(4,3)	Down	0.4

Figure 11.6: SARSA Algorithm: setting

1. In the first iteration, choose with the $\varepsilon-$greedy policy to exploit the best strategy and thus to go right into (4,3). The observed reward is 0.4. This implies to update $Q((4,2), \texttt{right})$ with the rule:

$$Q((4,2), \texttt{right}) = 0.6 + 0.1(0.4 + 1 * Q((4,3), a') - 0.6)$$

2. In order to choose *a'*, apply the $\varepsilon-$greedy policy and choose to explore (e.g. the **right** action). This will lead to a terminal state and thus the latter we can apply the formula for the update without the need of choosing another action with the $\varepsilon-$greedy policy. Suppose to obtain $Q((4,3),\texttt{right}) = 0.7$. Then , the value of $Q((4,2),\texttt{right})$ becomes:

$$Q((4,2),\texttt{right}) = 0.6 + 0.1(0.4 + 1 * 0.7 - 0.6) = 0.65$$

Chapter 12

Multi Armed Bandit

12.1 Introduction

RL problem with only 1 state in which the learner needs to repeatedly choose an action over a finite set $\mathcal{A} = \{a_1, a_2, ..., a_n\}$. Each action (here called **arm**)will give a reward (which can be known or not) and the objective is to decide which action to take in order to maximize the long-term reward (finite TH). The problem is undiscounted ($\gamma = 1$) as the game can continue for sure after each action. The typical MAB problem is the slot machine that has e.g. 3 arms (each one with a different reward) and the player must choose which arm to pull. The classification of MAB problems is :

1. **Deterministic**: once that the reward is discovered for each arm it will be always the same. The problem is trivial as it suffices to try all the action once to discover the reward and then always pull the arm with the highest reward.
2. **Stochastic**: the reward is unknown and, any time the arm a_i is pulled, the reward will be sample from an **unknown** distribution.
3. **Adversarial**: the reward is unknown and, at each time step, it is chosen by adversary. The key idea here is to behave in an **unpredictable** way.

The MAB is an **online problem** in which data is collected on the way and, for any new data, we would like to improve the performances (as fast as we can, i.e. with the minimum number of steps).A MAB problem is thus characterized by the trade-off **exploration-exploitation**: it uses the $\varepsilon-$greedy policy to decide what to do at each step. In order to represent this dilemma, the policy is:

$$\pi(a_i, s) = \begin{cases} 1 - \varepsilon & \texttt{if a}_\texttt{i}\texttt{ maximized }\hat{\texttt{Q}} \\ \frac{\varepsilon}{|\mathcal{A}|-1} & \texttt{on all other actions} \end{cases}$$

The policy can be represented also with the **Boltzmann distribution** that converges to the optimal distribution when the parameter τ decreases in time:

$$\pi(a_i, s) = \frac{e^{\frac{\hat{Q}(a_i|s)}{\tau}}}{\sum_a e^{\frac{\hat{Q}(a|s)}{\tau}}}$$

With both representation, however, we do not know how fast we converge.

12.2 Stochastic MAB

12.2.1 Definitions

Setting of a stochastic MAB:

1. **States**: only one state s available
2. **Actions:** $\mathcal{A}$ = set of possible arms
3. **State-transition probability**: $P(s|a, s) = 1 \ \forall a$
4. **Rewards:** $\mathcal{R}$ = set of distributions of rewards (one for each arm). The reward obtained by pulling the arm a_i is a random sample from an unknown distribution $\sim \mathbf{R}(a_i)$. This is not function of the states as there is only one state. The expected reward for action a_i is the reward function $E[\mathbf{R}(a_i)] = R(a_i)$. So with 5 different actions we have 5 unknown reward distribution, each one with a certain expected value.
5. **Discount factor**: $\gamma = 1$
6. **Initial probability**: $\mu(s) = 1$ as there is only one state

We cannot use the DP techniques as we do not know the rewards! **Evolution of an episode of MAB:**
For every t in $1...T$:

1. The agent chooses an arm $a_{i,t}$ and pull it.
2. The environment samples a reward from the reward distribution $\mathbf{R}(a_i)$ of the chosen arm.
3. The agent updates the information about the arm.

Objective:

$$\max\{\texttt{cumulative reward}\} = \min\{\texttt{exptected pseudo} - \texttt{regret}\}$$

where the **cumulative reward** is

$$\sum_{i=1}^{T} r_t$$

(r_t is the reward obtained by pulling an arm at time-step y)and the **expected pseudo-reward** is the difference between the optimal cumulative reward (i.e. TR^*, where R^* is given by the best action $a*$) and the average total reward(i.e.):

$$L_T = \sum_{t=1}^{T} R^* - E\left[R(a_i)\right] = TR^* - E\left[\sum_{t=1}^{T} R(a_i)\right]$$

Calling $\Delta_i = R^* - R(a_i)$ the difference between the maximum reward and the reward obtained by pulling a_i (both are average values) and $N(a_i)$ the number of times that and action a_i has been pulled:

$$L_T = E\left[\sum_{t=1}^{T} R^* - R(a_i)\right] = \sum_{a_i \in A} N(a_i)\Delta_i$$

With this definition, the regret is the sum of what we loose by pulling the arm a_i multiplied by the number of times we choose that arm, for all possible arms. This allows to minimize the regret by **minimizing the number of times** with which we pull a non-optimal arm. According to this, we can set a **lower** and an **upper** bound to L (i.e. L can never be zero). The total regret L_T has a **lower bound** that is proportional to $\log T$:

$$\lim_{T\to\infty} L_T \geq K \cdot \log T \quad \forall \texttt{algorithms for MAB}$$

The objective is to find an algorithm that ensures the regret of a time-step will converge to zero as the time passes by.

12.2.2 Pure Exploitation

In order to maximize the cumulated reward choose **at each time** t the action that gives $\max \hat{R}(a_i)$, where

$$\hat{R}_t(a_i) = \frac{1}{N_t(a_i)} \sum_{j=1}^{t} r_i y_j$$

i.e. add to the sum the reward obtained at step j iff at that time the action a_i has been chosen. This condition is expressed through the boolean variable y_j. This technique **only exploits the best action**: due to the missing of exploration, there is no guarantee that in infinite time the estimate converges to $R(a_i)$. This happens because we do not consider uncertainty on the estimation (done empirically with the above formula) and thus, if we always sample a small rebound from the arm which is actually optimal, we may discard it thinking that it cannot give higher rewards. To solve this problem, we need to keep exploring sub-optimal arms. This approach can be applied in two ways:

1. **Frequentist**: the mean values $R(a_i)$ for each arm are unknown **parameters**; at each time step the selection of the arm is based on history only.
2. **Bayesian:** the mean values $R(a_i)$ for each arm are **random variables** with a **prior distribution**; at each time step the selection of the arm is based both on history and on prior.

12.2.3 Frequentist: Upper Confidence Bound (UCB)

Instead of using $\hat{R}_t(a_i)$ alone, we build an **upper bound** of $R(a_i)$ as

$$U_t(a_i) = \hat{R}_t(a_i) + B_(a_i) \geq R(a_i)$$

The higher t, the more the bound gets near to $R(a_i)$. In order to do this, we use the **Hoeffding Bound** computing the probability that the real value exceeds the bound:

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

Then we put a value p for this probability and compute $B_t(a_i)$:

$$e^{-N_t(a_i)B_t(a_i)^2} \leq p$$

$$B_t(a_i) \geq \sqrt{\frac{\ln\left(\frac{1}{p}\right)}{2N_t(a_i)}}$$

Decreasing p over time, we put a tighter bound and we have guarantee to converge to $R(a_i)$ (consistent estimator). From this, the **UCB1 algorithm** is: For every time step t:

1. Compute $\hat{R}_t(a_i) \forall a_i$
2. Compute $B_t(a_i) \forall a_i$ and thus $U_t(a_i) = \hat{R}_t(a_i) + B_t(a_i)$
3. Pull the arm a such that $a = \texttt{argmax}_{actions}\left(U_t(a_i)\right)$

At finite time T, the expected total regret is upper bounded by

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

12.2.4 Bayesian: Thompson sampling

Consider to have a **Bernoulli** distribution for each arm a_i in which we obtain success (i.e. reward > 0) with probability p_i and failure (i.e. reward $= 0$) with probability $1 - p_i$. We do not know the value of p_i for each arm and they will be estimated during the process. At the beginning, each value in $[0, 1]$ is equally probable. The Thompson sampling algorithm is the following:

1. Consider a starting uniform **prior** $Beta(\alpha_0, \beta_0) = Beta(1, 1)$ for the value of p_i each arm.
2. Sample a reward $\hat{r_i}$ from each one of the Bernoulli distribution for each arm (for each arm, take as p_i the value that maximized the Beta). At each sample we can obtain either 0 (failure) or a positive reward (success)
3. Effectively pull the arm with highest $\hat{r_i}$.
4. Update the priors :
   (a) If the pulling gave a positive reward, the distribution of the pulled arm will now be $Beta(\alpha_t + 1, \beta_t)$
   (b) If the pulling gave a null reward, the distribution of the pulled arm will now be $Beta(\alpha_t, \beta_t + 1)$

   In this way the distribution of p_i changes for each arm.
5. Repeat 2-4 until satisfied

The Thompson sampling upper bound is

$$L_T \leq O(K \log T + \log \log T)$$

12.3 Adversarial MAB

The Adversarial MAB can be formulated as a problem in which the player (called **forecaster**) chooses one of N possible action at each step. The reward is chosen by an adversarial (the **nature**) and the game is played for **n rounds**. The behaviour thus is:

1. The nature selects the reward
2. The forecaster pulls an arm
3. The forecaster observes the reward

Given that there is an opponent that chooses the reward, we cannot use bounds described before. In this case, we use a **weak regret**, i.e.

$$L_T = \max_i \sum_{t=1}^{T} r_{i,t} - \sum_{t=1}^{T} r_{i_t,t}$$

The above quantity is the difference between the total reward given by best constant action(i.e. provided by an algorithm that chooses always the same action) that can be chosen and the cumulative reward effectively obtained. The lower bound that is valid for any algorithm applied in adversarial MAB is **higher** wrt the stochastic one, i.e. the problem is **harder** as we cannot reduce the regret as much as we can in stochastic MAB (we have to pay more).

12.3.1 EXP3 algorithm

The idea is to base on **randomness** of choice in order to be unpredictable. As in **softmax**, we keep a distribution of the possible arm and we choose sampling by it. The arms are selected randomly and the probability depends on weights w. For each step t:

1. Set $\pi_t(a_i)$ for each arm a_i
2. Randomly draw an action
3. Observe the reward
4. Update the values of the weight $w_t(a_i)$ of the arm that has been pulled, leaving unchanged the other weights.

The formulas are shown below:

The probability of choosing an arm is

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

where

$$w_{t+1}(a_i) = \begin{cases} w_t(a) e^{-\eta \frac{r_{t,t}}{\pi_t(a_t)}} & \text{if } a_i \text{has been pulled} \\ w_t(a_i) & \text{otherwise} \end{cases}$$

The upper bound of the average performance is $O(\sqrt{TNlogN})$ where N is the number of actions.

12.4 Generalized MAB

There are other types of MAB problems:

- Arm identification problem: we just want to identify the optimal arm with a given confidence, without caring about regret
- 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 $\mathcal{A}$ which is not
- Unimodal bandits: the arms are ordered in such a way that their expected value increases monotonically until some unknown point and than decease monotonically
- Expert setting: we are allowed also to see the reward which would have given the not pulled arm each turn (online learning problem)

Chapter 13

Appendix A - Evaluation

13.1 Evaluation of linear regression model

In order to choose the model, we should consider that the approximation error is a good estimator of the estimation error only with few features. Moreover, there are some common indices used to evaluate the goodness of the model, as shown below:

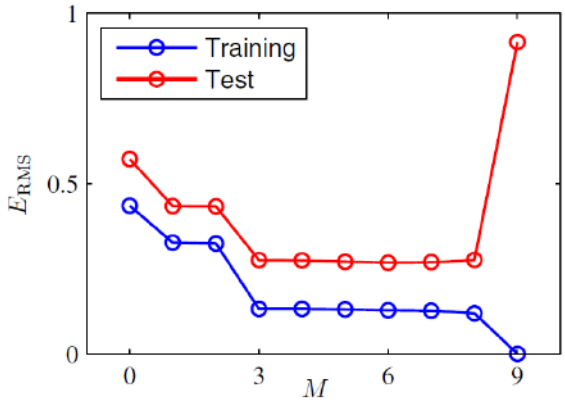


Figure 13.1: Approximation and Estimation error

Name	Expression	Comment
RSS(w)	$\sum_{i=1}^{N}(\hat{t_i} - t_i)^2$	How much the prediction differs from the target (globally)
R^2	$1 - \frac{RSS(\mathbf{w})}{\sum_{i=1}^{N}(\bar{t}-t_i)^2}$	How much better the model is wrt using just the sampled average value of the targets. **The closer to 1, the better**. Percentage of variance explained by the model.
Degrees of Freedom (dfe)	N-M	Flexibility of the model. Should be equal to $10^{number\ of\ parameters}$
Adjusted R^2	$R^2 \frac{N}{dfe}$	R^2 corrected with flexibility of the model
Root Mean Square Error (RMSE)	$\sqrt{\frac{RSS(\mathbf{w})}{N}}$	RSS normalized. Should be close to 0 and represents the expected error on new data

Table 13.1: Common evaluation indices

13.2 Evaluation of linear classification models

The evaluation of the classification models s done with the **confusion matrix** which allows to establish the percentages of:

1. **True positives (tp)**: values correctly classified in the positive class
2. **True negatives (tn)**: values correctly classified in the negative class
3. **False positives and negatives (fp, fn)**: misclassified values

	Effective = 1	**Effective = 0**
Predicted = 1	tp	fp
Predicted = 0	fn	tn

Table 13.2: Confusion matrix for 2 classes (binary classification problem)

The confusion matrix can be extended for K classes too:

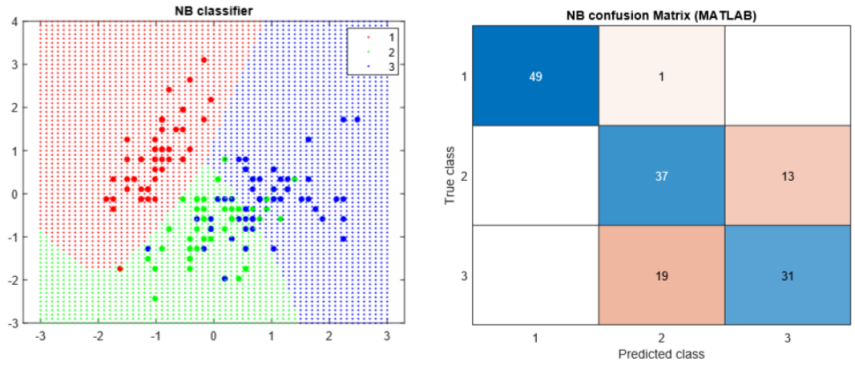


Figure 13.2: Confusion matrix for 3 classes

Based on this, the indices that can be used to evaluate the performances are :

1. **Accuracy**: the percentage of overall correctly classified points

$$Accuracy = \frac{tp + tn}{N}$$

2. **Precision**: the percentage of true positives wrt all the positive-classified point (i.e. fp and tp).

$$Precision = \frac{tp}{tp + fp}$$

3. **Recall**: the percentage of true positive wrt all the effectively positive points (i.e. fn and tp).

$$Recall = \frac{tp}{tp + fn}$$

4. **F1 score**: harmonic mean of precision and recall

$$F1 = \frac{2 \cdot Prec \cdot Recall}{Prec + Recall}$$

Given that the objective is to find the elements belonging to the positive class, the **selected elements** are the points classified as belonging to it (i.e. tp + fp), while the **relevant elements** are the points effectively belonging to it (i.e. fn + tp), The **precision** thus represents how many of the selected elements are effectively relevant, while the **recall** represents how many of the relevant elements have been selected.

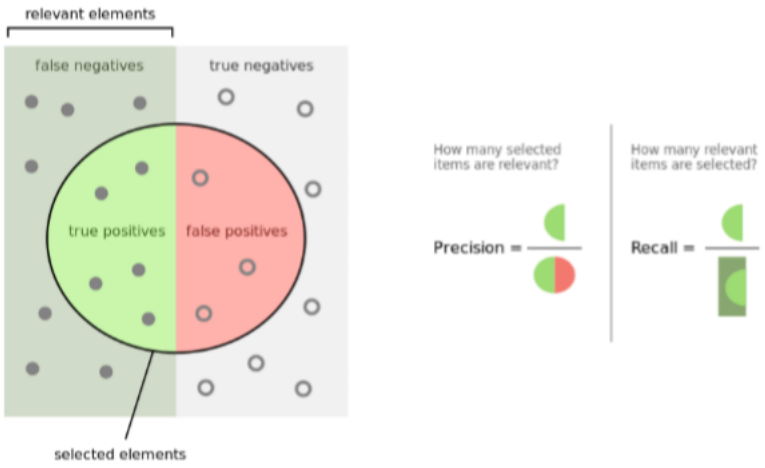


Figure 13.3: Graphical representation of precision and recall

13.3 Evaluation of Best Model for Feature Selection

13.3.1 Cross Validation

One cannot use the same dataset for multiple trainings, so a solution can be the **LOO** (Leave One Out) approach:

1. Divide the dataset into train and test. From the train data, remove one element (which will be the validation data)
2. Train the algorithm on the training set and compute the training error.
3. Use the validation data to compute the validation error.
4. Repeat train-validation phase removing one different data at each iteration. At the end, the model chosen after the validation will be the one that in all the iterations has had the smallest validation error. The general validation error, though, is an average of all the validation errors.
5. Perform the test applying the selected model on the test data.

The LOO approach guarantees an **unbiased model**, but if the dataset is big, the process can be very slow. To fasten it, the **k-fold Cross-Validation** is used. The training dataset is divided into k chunks: only one is used as validation set, the others are used for the training. The process is the same of LOO and at the end of validation phase the model with the best performance is chosen. The k-fold Cross-Validation method is **faster**, but also **biased**.

13.3.2 Adjustment Techniques

The indices used for the selection of the optimal model (between the ones that minimize the training error) have a correction term that considers the model complexity:

1. **C_p**: takes into account the number of parameters d and the estimation of the noise variance
$$C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma}^2)$$

2. **AIC**: takes into account the log likelyhood and the number of parameters d
$$AIC = -2\log L + 2d$$

3. **BIC** similar to C_p, but the correction terms depends on the number of data. If N > 7, BIC$< C_p$
$$BIC = \frac{1}{N}(RSS + \log N d\tilde{\sigma}^2)$$

4. **Adjusted R^2**: takes into account the estimation of the variance of the model $S_{model} = \frac{1}{N-1}\sum_{i=1}^{N} y_n - \bar{y}$. The larger the index, the better:
$$Adjusted\, R^2 = 1 - \frac{RSS}{S_{model}(N-d-1)}$$

Chapter 14

Appendix B - Practical Application on MATLAB

14.1 Linear regression - LS

14.1.1 Preprocessing

Before learning the model, data must be pre-processed with:

1. shuffling
2. removal of outliers and inconsistent data
3. filling of missing data
4. normalization of data (function `zscore`()), which moves the data around zero and divides by the standard deviation. This is necessary to interpret the F-statistics.

How to identify inputs to use? Use **gplotmatrix(inputs)** and select correlated inputs (i.e. there is positive or negative slope).

14.1.2 Interpretation of the results

There are various MATLAB functions for the LS method:

- **fit(x,t)**: only for 2 scalar inputs + 1 scalar output. Returns some indices like SE (standard error), R^2, dfe, Adjusted R^2, RMSE.
- **fitlm(x,t)**: also for multiple inputs. Returns also:
  - **F-statistics** VS constant null model: how well performs the model wrt the null model. A low *p-value* means that the model performs well wrt the null model.
  - Estimate, t-statistic, SE, p-Value for all parameters.

How to interpret the result:

- Which features are important: look at *p-value* for the parameters; if $p >> 0.05$, the feature is not relevant.
- RSS can be computed by RMSE as $RMSE^2 \cdot N$
- The average error on new data is RMSE.
- The percentage of variance explained by the model is given by R^2. E.g. $R^2 = 0.48$ means that the model explains 48% of the variance.
- High values of the parameters mean that the model is overfitting and that the inverted matrix is almost singular. Regularization should be applied.
- SE is the standard error: the lower the better
- The t-statistic $= \frac{Estimate}{SE}$: the higher the better

See also
https://it.mathworks.com/help/stats/understanding-linear-regression-outputs.html

Regularization

Consider the following MATLABcommands:

```
1 [lasso_coeff, lasso_fit] = lasso(x, t);
2 ridge_coeff = ridge(t, x, 0.0001);
```

The two commands are used to perform lasso and ridge regularization in order to make the model as simple as possible. The **ridge** uses $\lambda = 0.0001$ and discards the parameter w_0 as it is not meaningful. The **lasso** instead puts the non interesting features to 0 and returns the coefficients of the model and some statistics on it. The ridge model returns only the coefficients. The **lasso** returns different cases, computed with different values of lambda. The more λ gets near 1, the more the coefficients tend to zero as this means that the model becomes more and more simple. If $\lambda \to 0$, no regularization is performed. This implies that with higher λ, there is a higher MSE.

Confidence Interval

We run a linear regression and the slope estimate is $\hat{w}_k = 0.5$ with estimated standard error of $\hat{\sigma}v_k = 0.2$. What is the largest value of w for which we would NOT reject the null hypothesis that $\hat{w}_1 = w$? (assume normal approximation to t distribution, and that we are using the $\alpha = 5\%$ significance level for a two-sided test.

This is asking for the confidence interval, which is

$$[\hat{w} - \bar{z} \cdot \hat{\sigma}; \hat{w} + \bar{z} \cdot \hat{\sigma}]$$

Thus the superior limit of the interval is

$$0.5 + 1.96 \cdot 0.2 = 0.892$$

14.2 Linear Classification

14.2.1 KNN

1. Is the simplest method.
2. For each new datum , evaluate distance from the samples and reorder them in ascending order.
3. For the first K neighbours (with the smallest distance), take the class that appears most frequently.
4. The higher K, the smoother the boundary but also the more probable to have unknown values.

14.2.2 Perceptron

1. Uses the sign function on the linear model
2. No close-form solution, need to use the gradient. If do not use the `perceptron` network on MATLAB, do several iteration (e.g. 10) on all the data to update the parameters with the gradient.

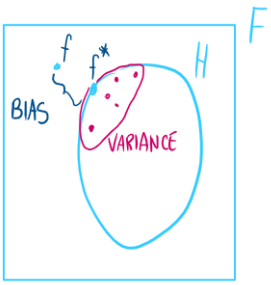
14.2.3 Logistic Regression

1. Uses the maximum likelihood built as a product of Bernoulli's distribution, where p = sigmoid of the linear model.
2. Uses the MATLAB methods
   (a) $[\texttt{B}, \texttt{dev}, \texttt{stat}] = \texttt{mnrfit}(\texttt{X}, \texttt{t})$
   (b) $\texttt{pihat} = \texttt{mnrval}(\texttt{B}, \texttt{X})$;

   Apply firt mnrfit and then mnrval. (`pihat`) is the probability of belonging to each class, for any sample.To have the prediction, take the maximum in each row with $\texttt{max}(\texttt{pihat}, [], 2)$, where the 2 refers to the second dimension (rows).

14.3 Bias-Variance dilemma

Used to choose the best model. Consider the below figure:



1. **Bias**: how much we are far from the real function inside the hypothesis space
2. **Variance** of the **parameter of the model**: how far we arrive from the optimal model inside the hypothesis space. We obtain f^* only with infinite number of data. If the dataset is finite, we have an approximation of f^* (variance is different from zero).

To compute the exact variance and bias, we need the real function and infinite data, but we don't normally have it.

14.3.1 Ideal example: f(x) is known

Assume to have a **regression problem** : for easiness that the true model is known:

$$t(x) = f(x) + \varepsilon = 1 + \frac{1}{2}x + \frac{1}{10}x^2 + \varepsilon$$

We generate some random data from this model. We want to estimate *f(x)* and we consider two hypothesis space:

1. $\mathcal{H}_\infty$ =**linear model**: $y = \mathrm{b}x + \mathrm{a}$
2. $\mathcal{H}_\in$ =**Quadratic model:** $y = \mathrm{c}x^2 + \mathrm{b}x + \mathrm{a}$

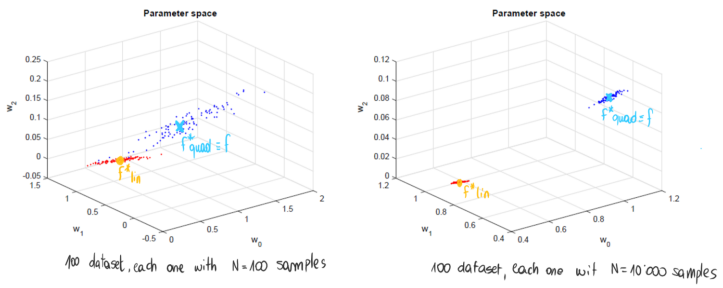
In this case, assuming to have infinite data, we could compute exactly the bias and the variance. With the linear model, we have a simpler model wrt the real process, so the function f^* that we obtain is not equal to f. With the quadratic model, instead, we obtain exactly f:

1. $\mathcal{H}_\infty$ =**optimal linear model**: $f^* = \frac{7}{12} + x$
2. $\mathcal{H}_\in$ =**optimal quadratic model:** $f^* = 1 + \frac{1}{2}x + \frac{1}{10}x^2$

What happens with a finite number of data?

1. $\mathcal{H}_\infty$ =**linear model**: $y = \mathrm{b}x + \mathrm{a}$
2. $\mathcal{H}_\in$ =**Quadratic model:** $y = \mathrm{c}x^2 + \mathrm{b}x + \mathrm{a}$

Consider to have 100 different datasets in two cases:



1. Each dataset has N = 100 samples : the linear models are nearer to the optimal linear model wrt the case of the quadratic model.
2. Each dataset has N = 10000 samples: in both cases hypothesis space, the models are concentrated near their optimal f^*, but the linear models will never be near to the true one as the quadratic term is not present.

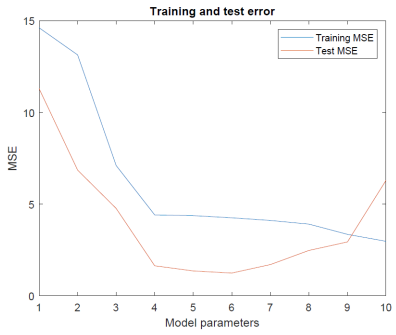
Thus empirically we conclude that:

1. The **linear model** has less variance, but more bias.
2. The **quadratic model** has more variance but less bias.
3. Adding samples, we reduce the variance, but the computation is slower. Moreover, we do not always have the possibility to have lots of data.
4. With lots of data, is better to have complex models as the quantity of data already reduces the variance.
5. With few data, we should use simpler model and accept some bias to have lower bias (with more precise model, the estimation is more "controllable" and predictable).

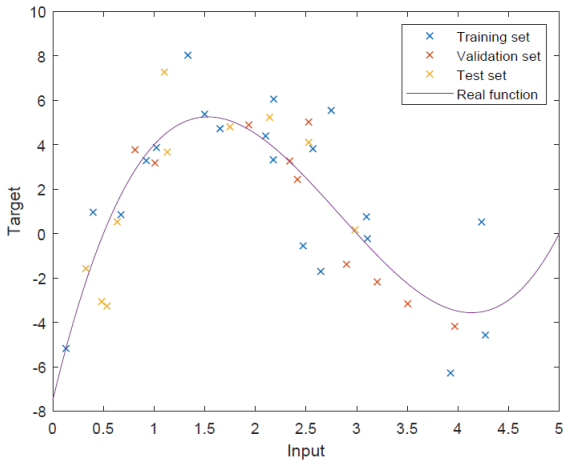
In general, choose the model with the overall smaller error. Remember that these considerations are done in **average**, not for a single realization.

14.3.2 Realistic example: f(x) is not known

When $f(x)$ is not known, what we can check to evaluate the model is the **RSS** of the model. This, however, is not a good estimation of the error on new data. To check the performances, we use some data for the test and we check the MSE for both train and test.



To have a generalization of the error on new data, we should use the **validation**. Remember that the validation error is not a good estimation as it is biased. After train and validation, the test error will be a good estimation on new data as it is independent from the learning phase.



If is important to have test, train and validation data **shuffled together**, otherwise I would train only a part of the function and validate on another one and test on another one else. Thus, shuffle data before separating the train-validation-test set!!! Remember: **we cannot use ML unless the distribution of the data is invariant wrt time**. Remember: **without cross validation or leave one out, the validation error is biased wrt the test error**. With cross-validation, we have the validation error that is the average between all the validation error obtained. Once the best model is found, we retrain it with the whole training set and then test it.

14.3.3 Exercises from Trovò

Exercise 5.1

Cp, AIC, BIC and adjusted R2 indexes take into account both the error and the complexity of the model so the division between training and validation is not useful. For example, with very few data the validation / cross-validation / LOO are not useful and thus these indices should be used. Rule of thumb for the number of samples wrt the number of parameters in regression: we should have 5-10 data for each parameter. With 2 parameters, we should have at least 10-20 samples. With fewer samples, do not use validation.

Exercise 5.5

1. NO, because it is related to the noise on data and it is independent from the complexity of the model.

2. YES, but only on average: I cannot be sure that this will happen on a single realization. We could have the same bias if the true model is linear and thus increasing the complexity will not change the bias (if the bias is already zero, it cannot decrease anymore). Moreover, if the true model is even more complex than quadratic, it may happen that increasing the hypothesis space does not decrease the bias.

3. NO, on average the more complex model have bigger variance.

4. IT DEPENDS: since it depends on how much the bias and variance change.

Exercise 5.6

1. FALSE, as the average accuracy is 0.5
2. TRUE, because for any algorithm there will be one region in which they perform well and other regions in which they will perform bad.
3. FALSE, because the training data will change the probability
4. FALSE, because it would be over-fitting the training data and with a small noise the result would be wrong. USUALLY WE ARE GIVING PAC PROPERTIES.

Exercise 5.7

1. TRUE, we have less parameters and the selected ones are the most significant.
2. TRUE, it puts to zero the parameters which are not useful.
3. FALSE, because we obtain a simpler model.
4. TRUE, as a simpler model will have smaller variance on average.

The choice of λ in LASSO is done with validation, cross-validation and LOO. $\lambda \geq 0$ up to ∞. RIDGE instead does not provide sparsity and if a parameter is small it does not imply that it is not significant.

Exercise 5.12

Two linear models on the same dataset of N = 100 samples. They use the same validation set.

1. Model 1: 2 inputs (x_1, x_2), uses linear features, RSS = 0.5 on the validation set.
2. Model 2: 8 inputs $(x_1, ..., x_8)$, uses quadratic features (all possible quadratic combination of the inputs of only degree 2), RSS = 0.3 on the validation set.

We'd choose the model with the smallest RSS (thus the second one). But how do we know that the RSS is significantly smaller and that this is not only due to randomness of the data? We use the F-statistic, distributed as a Fisher distribution to decide if we have:

1. $RSS_1 \leq RSS_2$
2. $RSS_1 \geq RSS_2$

The second one would be preferred because this means that the second model is better. The F-statistics says if a group of variables are jointly statistically significant wrt the null hypothesis. In order to compute the F-statistics we need:

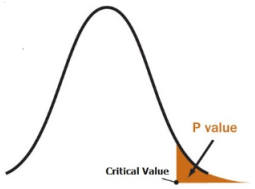
1. Number of samples: N = 100

2. The number of parameters of the 2 models: $p1 = 1 + 2 = 3$, while $p2 = 1 + 8 + 28 = 37$. In the second model we have the constant term, all the quadratic features with one input, all the quadratic features made by the combination of two inputs $\binom{8}{2}$

Substituting in the F statistics we have:

$$F = \frac{N - p_2}{p_2 - p_1} \frac{RSS_1 - RSS_2}{RSS_2} = 1.23$$

Knowing that the F statistics behaves like a Fisher Distribution, we should check the **p-Value**, i.e. 1 - cumulated Fisher Distribution computed at the found value.



The pValue is $0.23 >> 0.05$. Thus we do not have statistical significance that the second model is better than the first one.

In order to improve the situation, we could increase the number of data to understand if there is statistical significance that the second model is better. Up to now, we can say that with confidence of 77% we cannot reject the hypothesis that the first model is better than the second one.

Exercise 5.13

1. LOO because the computation is not highly costly on small dataset.
2. Train on whole dataset and use Lasso to select the important features, or consider the criteria AIC, BIC as they penalize the complexity.
3. k-fold validation because we can separate the validation and train set with big datasets; better than simple validation as it is less biased.
4. k-fold validation because we can train different models at the same time. LOO can be very good too because we have a lot of computation power.

Simple model means that the train is very fast. The exercise considers parametric method as there is a train phase. In non-parametric method no training is required, as only the prediction is performed (e.g. KNN).

14.4 Feature selection and Kernel methods

14.4.1 Outline

What if the model we chose does not meet the performance requirements? We could simplify / complicate the model. To simplify the model:

1. Regularization
2. Feature extraction (PCA): the most statistically significant features are selected through eigenvalues.
3. Feature selection: select a subset and run only on them. The problem is the computation.

To generate a more complex model:

1. Gaussian Processes
2. Kernels (SVM)
3. Basis Functions

14.4.2 PCA

PCA is an **unsupervised method** used to extract new features from the dataset (the extracted features are less than the original set of features). If the problem is in D-dimensional input space, the maximum allowed principal components are D. The last dimensions have a lot of noise and thus there is almost always only noise. The algorithm is:

1. Remove the empirical mean from the data (find the new origin).
2. Compute the covariance matrix as $X^T X$
3. The eigenvectors are the principal components, ordered by the value of eigenvalues.

Given an eigenvalue λ_i, the variance explained by the corresponding eigenvector is $\frac{\lambda_i}{\sum_i \lambda_i}$ as all the eigenvalues represent the variance in the direction of the eigenvector. On matlab:

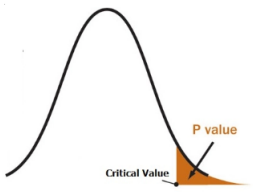
$$[\texttt{loadings}, \texttt{scores}, \texttt{variance}] = \texttt{pca}(\texttt{X});$$

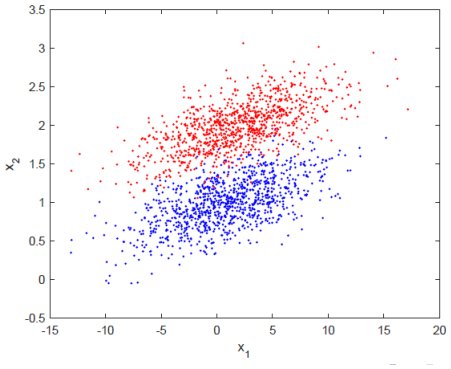
where:

1. `loadings` = matrix **W** with the principal components
2. `scores` = the data transformed in the new space (i.e. projected on the principal components)
3. `scores` = the eigenvalues

How many components to keep? There are various ways:

1. Keep only the ones whose cumulated variance is the 90-95% of the total variance.
2. Keep only the component whose single contribution is higher than 5% of all the variance.
3. Find the elbow (is quite subjective)





PCA can be used also for visualization purposes to see if there is a structure in the components. PCA does not always work: in the following example, is better to project over the second component (**Simpson's paradox**): it we project in the first dimension, we cannot distinguish the data:

Keep in mind that PCA is used assuming generally guassian data, not for strange distributions.

14.5 Gaussian Processes

Instead of choosing the H on polynomials, we consider the inputs in $\mathbf{x}$ to be **jointly gaussian**:

$$\begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix} \sim N(\mu, \mathbf{C})$$

All the newly added points are jointly gaussians with the other values. μ is a vector with the average value for each input, while $\mathbf{C}$ is the covariance matrix. In each input x_i the mean and σ_i are determined by all the other points. Depending on the **covariance**, I can plot a function of the jointly gaussian inputs.The Covariance is defined with a kernel and specifies how much two inputs are similar (in particular, we use a **gaussian kernel**):

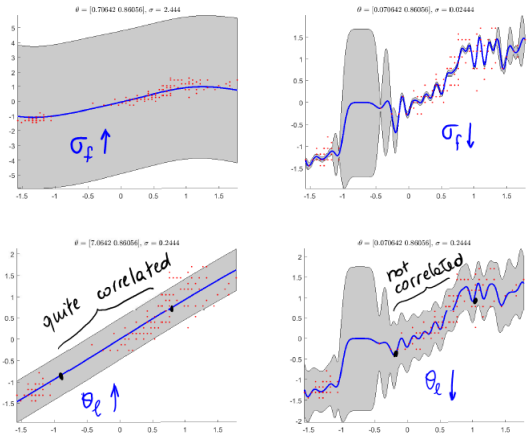
1. If two inputs are similar, the covariance entry is near 1 (very correlated). the value of the function in two near points is similar.

2. If the two inputs are very far, the covariance entry is near zero. The points are allowed to have a huge distance in the value.

The idea is to have the gaussian distribution of new target given the dataset. For each point, we are able to describe a gaussian distribution computing its variance and mean. Differently from linear regression, in which for each input the variance is the same, with gaussian proocess the variance changes at each point (i.e. we have different confidence in different points in the input space.) Considering the squared gaussian kernel:

$$\mathbf{K}_{ij} = \sigma_f^2 e^{-\frac{||x_i - x_j||}{2\theta_l^2}}$$

1. σ_f^2 represents the variance, as with i=j the exponential is 1. Is the largest noise that I expect on output when I do not have many data
2. θ_l is the bandwidth: how much the data should be close before the covariance goes to 0. The higher the value, the more regular the result is because even far points are related to each other.

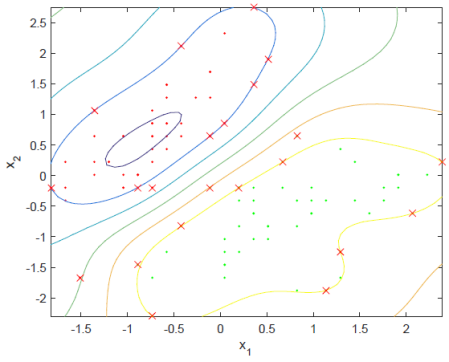


14.5.1 SVM

To use SVM in MATLAB:

1. **Train:** $\texttt{svm}_\texttt{m}\texttt{odel} = \texttt{fitcsvm}(\texttt{irisInputs}, \texttt{irisTargets})$; the parameter C is 1 (the weight of slack variables).If C is very large, even a small mistake will count a lot and this will change the boundary (the model will have a small margin). If C is **small**, the model will not change much (increase the margin)
2. **Draw the boundary**: $\mathbf{w^T x_n} + b = 0$, where $\mathbf{w}$= $\texttt{svm}_\texttt{m}\texttt{odel.Beta}$ and b= $\texttt{svm}_\texttt{m}\texttt{odel.Bias}$
3. **Draw the margins:** $\mathbf{w^T x_n} + b = \pm 1$

The $\mathbf{w}$ and b are written only as functions of the support vectors. The kernel can be added with $\texttt{svm}_\texttt{k}\texttt{ernel}_\texttt{m}\texttt{odel} = \texttt{fitcsvm}(\texttt{irisInputs}, \texttt{irisTargets}, '\texttt{KernelFunction}', '\texttt{gaussian}')$;, but do not have a specific formula for the boundary in the input space.



Differently from KNN, we do not need to look at all the samples to make a prediction, it is enough to check the Support Vector. The SVM method thus is non-parametric and is very sparse.

14.5.2 Exercises from Trovò

Exercise 6.9

1. TRUE, because the new mean will be the origin of the space generated by the PC. If not, we have vector from the origin to the center of the point cloud, which is not a principal component.

2. FALSE, if we keep the eigenvectors that have the 90-95% of the variance. If we keep all the eigenvectors, moreover, the reconstruction is exact. We just multiply be **W'** as the matrix is orthonormal (all column have norm= 1, are orthogonal, and $W^{-1} = W'$)

$$X = \tilde{X}W'$$

3. TRUE because we have at most d principal components.(Actually, even $k < d$. For visualization purposes, k is 2 or 3).

4. FALSE because it considers only the variance. The PCA is a deterministic method!!

Exercise 6.13

Which dataset would you use with Kernel:

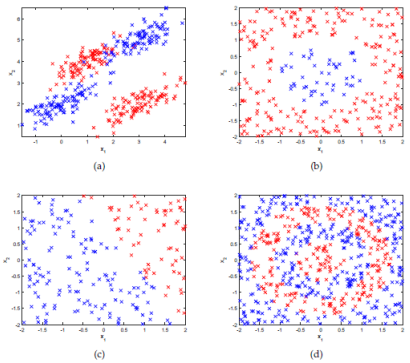
1. Yes because there is no linear separation (use **gaussian kernel**)

2. No because can use the features and transform into linear space

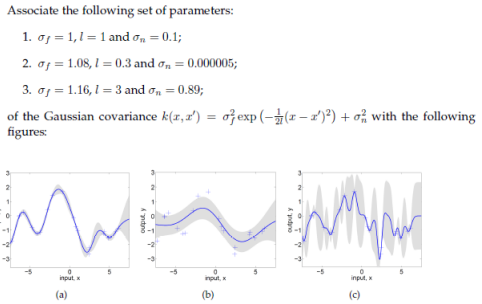
3. No sense because it has already a linear boundary

4. Yes because there is no linear separation (use **gaussian kernel**)

How to know if two classes are linearly separable: if you run the method and the train error is zero, they are separable; another way is to use PCA for visualization. Kernel should be our last option when the other techniques are not usable



Exercise 6.19



For each configuration, the figure is:

1. Figure **a**
2. Figure **c**
3. Figure **b** because it has the smaller l

Meaning of the parameters:

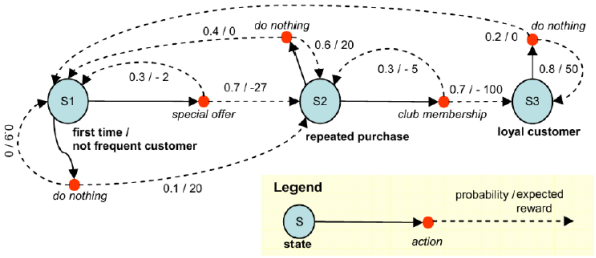
1. σ_f: Variance (maximum achievable)
2. σ_n: acts like noise
3. l :scale (how far 2 points have to be so that the function change; if high, the function is smooth, if small, the function is nervous.1)

14.6 Markov Decision Processes

14.6.1 Definition and Bellman Equation

Consider this problem:
The user can be in 3 states:

1. Not frequent user (S1)



2. Repeated purchase (S2)
3. Loyal customer (S3)

To move the customer between states, we can perform some actions:

1. Do nothing, on all the states
2. Give a special offer (with probability 0.7 the customer goes from S1 to S2)
3. Give a club membership (with probability 0.7 the customer goes from S2 to S3)

Each action will lead to the next state with a certain probability. This will be model with the **state-transition probability matrix**. Each action has also a **reward**, which in this case represents the cost/gain that an action will provide. The action "do nothing" will give a lot of reward in state S3, but nothing in state S2: its advantage changes along the states. Suppose that you always choose to DO NOTHING: what is the value of the states?
Let's model the problem with an **MDP**.

1. **States** : S1, S2, S3
2. **Actions** : Do nothing, Special offer (S1) Do nothing, club membership (S2) do nothing (S3). Remember to specify all the actions for each state, even if repeated. Following the above order and deciding to do nothing at each state, the matrix that describes the policy would become

$$\pi(a|s) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

   The dimension of the matrix is $|\mathcal{S}| \times |\mathcal{A}|$

3. **Reward**: the reward is a vector $|\mathcal{S}||\mathcal{A}| \times \infty$ that specifies, for each couple of state-action, the expected reward gained by taking that action. For each action, do an average of the rewards it can bring weighted by the probability of obtaining that reward. For example, in state S1 taking the action SPECIAL OFFER gives an expected reward equal to

   $0.3 \cdot (-2) + 0.7 \cdot (-27) = -19.5$

   Given that we have fixed the reward, we consider only the action DO NOTHING and thus we disregard the other actions. The result is

$$R^\pi = \begin{bmatrix} 0.9 \cdot (0) + 0.1 \cdot (20) \\ 0.4 \cdot (0) + 0.6 \cdot (20) \\ 0.2 \cdot (0) + 0.8 \cdot (50) \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \\ 40 \end{bmatrix}$$

4. **Transition**: Considering the policy, the probabilities are the one given by the action DO NOTHING in each state. The result is a matrix $|\mathcal{S}| \times |\mathcal{S}|$:

$$P^\pi = \begin{bmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.6 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix}$$

5. **Discount factor**: given by the problem. Is not 1 because otherwise we have infinite reward, nor 0 because the problem may go on for a long time. With small γ (myopic), we do not evaluate a lot the future, with big γ we evaluate a lot the future (far-sighted). Consider $\gamma = 0.9$

6. **Initial probabilities**: gives the chance that the user will start in each one of the states.

14.6.2 Prediction

To solve the prediction problem we use the **Bellman Expectation Equation**

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$
$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

1. The recursive solution will take more step

2. The close-form solution will take time for the inversion. Not so good if we have a lot of states.

The same problem can be solved with the **Q function**, i.e. the value function before the action that I am doing. The Q function gives a more general view:

1. The **Reward** considers all possible actions in all possible states. Given the formulation that we used before (in which the same action is repeated if it can be done in several states), the reward is a vector $|\mathcal{A}| \times \infty$:

$$R = \begin{bmatrix} 0.9 \cdot (0) + 0.1 \cdot (20) \\ 0.3 \cdot (-2) + 0.7 \cdot (-27) \\ 0.4 \cdot (0) + 0.6 \cdot (20) \\ 0.3 \cdot (-5) + 0.17 \cdot (-100) \\ 0.2 \cdot (0) + 0.8 \cdot (50) \end{bmatrix} = \begin{bmatrix} 2 \\ -19.5 \\ 12 \\ -18.5 \\ 40 \end{bmatrix}$$

2. The **State-Transition** matrix consider all possible states and actions:

$$P = \begin{bmatrix} 0.9 & 0.1 & 0 \\ 0.3 & 0.7 & 0 \\ 0.4 & 0.6 & 0 \\ 0 & 0.3 & 0.7 \\ 0.2 & 0 & 0.8 \end{bmatrix}$$

To compute the solution with the **Bellman equation** we have:

$$Q^\pi = R + \gamma P \pi Q^\pi$$
$$Q^\pi = (I - \gamma P \pi)^{-1} R$$

Depending on the value of γ a policy π_1 may perform better or worse than another policy π_2.

14.6.3 Control

For any MDP, at least one optimal policy is **deterministic, markovian and stationary**. To find the optimal policy:

1. **Policy search** (Brute Force)
2. **Policy iteration**: use Bellman to estimate value and then to improve the policy.
3. **Value iteration**: use Bellman to improve the value

Consider the advertisement problem:

1. **Brute Force**: search all the deterministic policies, comput the values V^π and choose the highest. Remember that is one policy is better than another on one state, it will be better also on all other states. The policies to be checked are $|\mathcal{A}|^{|\mathcal{S}|}$ so with complex process it is very slow.
2. **Policy Iteration**: use the Bellman optimality equation to estimate the value (**evaluation**) then try to improve the policy (**improvement**) with the greedy action (the best one, i.e. that leads to the highest Q).
3. **Value Iteration**: apply iteratively the Bellman optimality equation

The difference between policy iteration and value iteration is that in the first we have to solve a prediction problem at every iteration, while in the second one we just cycle on the same equation. We do not know which is the best approach because the number of steps for convergence depends on the problem.

14.6.4 Exercises from Trovò

Exercise 7.9

Observation is a state; the action is a prediction; the reward is 1 if the prediction is correct, 0 otherwise. It does not make sense to model a classification problem with a reinforcement learning problem, as we cannot model how we make the prediction. The MDP models the temporal influence between 2 states, while here the data are not drawn by a ddp which is not influenced by time. Is like using a bazooka to shoot a fly.

Exercise 7.11

1. TRUE because it means that we are not counting too much on future rewards
2. TRUE because the cumulative reward may diverge
3. FALSE it cannot be learned
4. TRUE its the meaning of γ