Machine Learning

Niccoló Didoni

February 2022

Contents

I Introduction **1**

1 Introduction to machine learning **2**
1.1 Inductive and deductive process . . . 2
1.1.1 Inductive reasoning . . . 2
1.1.2 Deductive reasoning . . . 2
1.1.3 A paradigm change . . . 3
1.1.4 Usage of machine learning . . . 3
1.2 Classification of machine learning . . . 3
1.2.1 Supervised learning . . . 3
1.2.2 Unsupervised learning . . . 5
1.2.3 Reinforcement learning . . . 5
1.3 Dichotomies . . . 6
1.3.1 Parametric and non-parametric approaches . . . 6
1.3.2 Frequentist vs Bayesian . . . 7
1.3.3 Generative vs discriminative . . . 7
1.3.4 Empirical risk minimization vs structural risk minimization . . . 7

II Supervised learning **8**

2 General description **9**
2.1 Graphical description . . . 9
2.1.1 Approximating the loss function . . . 11
2.1.2 Problem representation . . . 11
2.1.3 Model evaluation . . . 12
2.1.4 Optimisation . . . 12

3 Linear models for regression **13**
3.1 Linear regression . . . 13
3.1.1 Model . . . 13
3.1.2 Loss function . . . 15
3.1.3 Basis functions . . . 16
3.2 Minimising the loss function . . . 16
3.2.1 Approaches . . . 16
3.2.2 Mean square minimisation . . . 17
3.2.3 Stochastic gradient descend . . . 19

3.2.4 Maximum likelihood . . . 20
3.2.5 Multiple outputs . . . 23
3.2.6 Under-fitting and over-fitting . . . 23
3.3 Regularisation . . . 24
3.3.1 Ridge regression . . . 24
3.3.2 Lasso . . . 25
3.4 Bayesian linear regression . . . 25
3.4.1 Posterior distribution . . . 26
3.4.2 Bayesian linear regression and over-fitting . . . 26
3.4.3 Predictive distribution . . . 27
3.4.4 Modelling . . . 27
3.4.5 Computational complexity . . . 28
3.4.6 Advantages and disadvantages . . . 28

4 Linear models for classification **29**
4.1 Linear classification . . . 29
4.1.1 Model . . . 29
4.1.2 Approaches to classification . . . 31
4.2 Thresholds . . . 31
4.2.1 Double class . . . 31
4.2.2 One versus the rest . . . 31
4.2.3 One versus one . . . 32
4.2.4 Linear discriminant functions . . . 32
4.3 Discriminant functions . . . 33
4.3.1 Least squares . . . 33
4.3.2 Perceptron . . . 34
4.4 Probabilistic discriminative models . . . 37
4.4.1 Logistic regression . . . 37
4.4.2 Comparison between the perceptron and logistic regression . . . 40
4.5 Naive Bayes . . . 40
4.6 K-nearest-neighbours . . . 42
4.7 Result evaluation . . . 43

5 Theoretical concepts **45**
5.1 No free lunch theorems . . . 45
5.2 Bias-variance trade-off . . . 46
5.2.1 Bias . . . 47
5.2.2 Variance . . . 48

6 Testing and validation **51**
6.1 Measuring error and evaluating models . . . 51
6.1.1 Training error . . . 51
6.1.2 Prediction error . . . 52
6.1.3 Test error . . . 52
6.2 Validation . . . 53
6.2.1 Adjustment techniques . . . 55
6.2.2 Validation . . . 56

7 Managing the bias-variance trade-off **58**
7.1 Model selection . . . 58
7.1.1 Feature selection . . . 59
7.1.2 Regularisation . . . 60
7.1.3 Dimension reduction . . . 60
7.2 Model ensembles . . . 63
7.2.1 Bagging . . . 63
7.2.2 Boosting . . . 64
7.2.3 Comparison . . . 65

8 Balancing hypothesis space and data set **67**
8.1 PAC-Learning . . . 67
8.1.1 Version spaces . . . 68
8.1.2 Agnostic learner . . . 71
8.1.3 Continuous hypothesis space . . . 72
8.2 VC-Dimension . . . 72
8.2.1 Dichotomy and shattering . . . 72
8.2.2 VC-Dimension . . . 73
8.2.3 Estimating the data set size . . . 74
8.2.4 Properties . . . 75

9 Kernel methods **76**
9.1 Kernel . . . 76
9.1.1 Motivations . . . 76
9.1.2 Kernels . . . 77
9.1.3 Valid kernels . . . 79
9.1.4 Gaussian kernel . . . 81
9.2 Ridge regression . . . 81
9.2.1 Prediction . . . 83
9.2.2 Similarities . . . 84
9.2.3 Advantages of the kernel-representation . . . 84
9.3 Radial basis function networks . . . 84
9.3.1 Nadaraya-Watson model . . . 85
9.4 Gaussian processes . . . 87
9.4.1 Bayesian linear regression . . . 88
9.4.2 Predictions . . . 90
9.4.3 Kernel . . . 91
9.4.4 Uncertainty . . . 91

10 Support Vector Machines **93**
10.1 Support Vector Machines . . . 93
10.1.1 Support Vector Machines from pereceptrons . . . 94
10.2 Margin . . . 94
10.2.1 Prediction . . . 98
10.2.2 Dimensions . . . 99
10.3 Noisy data . . . 99

III Reinforcement learning 102

11 Markov Decision Processes 103
11.1 Introduction . . . 103
11.1.1 Agent-environment interface . . . 103
11.1.2 Usages of reinforcement learning . . . 104
11.1.3 Model definition . . . 104
11.2 Markov Decision Process . . . 105
11.2.1 Markov assumption . . . 105
11.2.2 Discrete-time finite Markov Decision Process . . . 105
11.3 Rewards and goals . . . 106
11.3.1 Time horizon . . . 106
11.3.2 Cumulative reward . . . 107
11.4 Policies . . . 108
11.4.1 Stationary stochastic Markovian policies . . . 109
11.5 Bellman Expectation Equation . . . 110
11.5.1 Policy evaluation . . . 110
11.5.2 Bellman Expectation Equation . . . 111
11.5.3 Bellman operators . . . 112
11.5.4 Optimal value function . . . 113
11.5.5 Optimal policies . . . 114

12 Dynamic programming 117
12.1 Introduction . . . 117
12.1.1 Prediction and control . . . 117
12.1.2 Finite-horizon dynamic programming . . . 118
12.2 Policy Iteration . . . 118
12.2.1 Policy evaluation . . . 119
12.2.2 Policy improvement . . . 120
12.3 Value Iteration . . . 120
12.4 Complexity and efficiency . . . 122
12.5 Linear programming . . . 123
12.5.1 Optimal value-function . . . 123
12.5.2 Optimal policy . . . 124
12.5.3 Complexity . . . 124

13 Reinforcement Learning techniques 125
13.1 Classification . . . 125
13.1.1 Techniques . . . 125
13.1.2 Problems . . . 126

14 Model-free prediction 127
14.1 Monte-Carlo reinforcement learning . . . 127
14.1.1 Mean estimation . . . 127
14.1.2 Monte-Carlo policy evaluation . . . 128
14.1.3 Incremental mean . . . 130
14.2 Temporal Difference . . . 131
14.2.1 Temporal Difference-0 . . . 131
14.2.2 Comparison with Monte-Carlo . . . 132

14.3 Temporal Difference-Lambda . . . 133
14.3.1 Lambda return . . . 133
14.3.2 Forward-view TD-Lambda . . . 134
14.3.3 Backward-view TD-Lambda . . . 134
14.3.4 TD(1) and Monte-Carlo . . . 135

15 Model-free control **136**
15.1 On-policy control . . . 136
15.1.1 Monte-Carlo control . . . 136
15.1.2 Temporal Difference control and SARSA . . . 139
15.2 Off-policy learning . . . 141
15.2.1 Importance sampling . . . 141
15.2.2 Importance sampling for off-policy Monte-Carlo . . . 142
15.2.3 Important sampling for off-policy SARSA . . . 143
15.2.4 Q-learning . . . 143

16 Multi Armed Bandit **144**
16.1 Introduction . . . 144
16.1.1 Reward . . . 144
16.2 Stochastic Multi Armed Bandit . . . 145
16.2.1 Formalisation . . . 145
16.2.2 Regret . . . 146
16.2.3 MAB lower bound . . . 148
16.2.4 Pure exploitation algorithm . . . 148
16.2.5 Upper Confidence Bound . . . 149
16.2.6 Thompson sampling . . . 150
16.3 Adversarial Multi Armed Bandit . . . 151
16.3.1 Pseudo regret . . . 151
16.3.2 EXP3 . . . 152

A Probability **153**
A.1 Bayes . . . 153
A.2 Joint distributions . . . 153
A.3 Conjugate prior . . . 154
A.4 Gaussian distribution . . . 154

B Algebra **155**
B.1 Notation . . . 155
B.2 Positive and semi-positive definite matrix . . . 155
B.2.1 Positive definite matrix . . . 155
B.2.2 Semi-positive definite matrix . . . 156
B.3 Eigenvectors and eigenvalues . . . 156
B.4 Matrix properties . . . 157

Part I

Introduction

Chapter 1

Introduction to machine learning

Machine Learning is the field of Computer Science that studies programs that can learn from experience. More precisely

> **Definition 1.1** (Machine learning)**.** *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance P at task t in T, as measured by P, improves with experience E.*

As we can notice from Definition 1.1, the focus of learning is on experience, in fact the more the experience, the better the program's performance is.

1.1 Inductive and deductive process

1.1.1 Inductive reasoning

Since the learning process is based on experience, we can say that learning programs use an **inductive** process to solve a problem. In particular the inductive way of reasoning imposes that one starts from evidence and tries to generalise. This approach is bottom-up, in fact we start from raw events upon which we can build knowledge (i.e. generalise the raw events). In other words, **inductive reasoning tries to find regularities that can be generalised**. This means that, this process isn't perfect, in fact a program can't learn if it has too few experience. Remember that, machine learning only extract knowledge from data.

1.1.2 Deductive reasoning

The opposite approach with respect to inductive reasoning is **deductive** reasoning. In this case, we start from some already known knowledge and generate new knowledge. This approach is top-down, in fact we start from general well-known concepts and specialise them.

It's important to underline that these approaches aren't exclusive, in fact they can be combined to solve problems very efficiently. An example is AI, that studies how to mix inductive (learning) and deductive (search and planning) to solve some tasks.

1.1.3 A paradigm change

Classical problem solving is based on a simple paradigm, we give some code and some data on which the code has to be executed to a machine and it produces some output (i.e., `in + code = out`). Machine learning is based on a different paradigm, in fact we give some input data and the respective desired input to a machine that has to figure out the code that maps the input in the outputs (i.e., `in + out = code`). The inputs and outputs (in some cases only the inputs) are called data-set D and we say that the data-set is used to train a program because it's used to obtain the `code`.

1.1.4 Usage of machine learning

Machine learning is used when

- We want to make **informed decisions on unseen data** (i.e. generalise on unseen samples).
- It's **hard to design and program rules by hand**.

A good example is image recognition; deciding if an image contains or not the face of a human is an hard to code problem and the program has to work with every image, hence even with images it has never seen.

1.2 Classification of machine learning

Machine learning algorithms can be divided in three main categories

- **Supervised learning** algorithms. When using supervised learning, a machine observes some inputs and the related outputs and tries to find the function (i.e. the model) that maps the inputs in the outputs.
- **Unsupervised learning**. When using unsupervised learning, a machine observes the input data and learns how to better represent them. In other words we are trying to find a structure in the input data.
- **Reinforcement learning**. When using reinforcement learning, a machine tries to learn how to act using a reward depending on the previous actions.

1.2.1 Supervised learning

Supervised learning allows a program to estimate a model, known the input and output data. This means that in supervised learning the data-set D can be written as a set of couples (x, t)

$$D = \{(x, t)\}$$

where

- x is an **input** value.
- t is the **desired output** (also target) when considering input x.

The program has to learn how to map x in t, i.e., it has to learn a function f that maps x in t for every couple in the data-set D

$$f(x) \to t$$

The function f is initially unknown but the program can learn it using the data-set. In other words the program learn to generalise the data-set.

Let's take a moment to understand better why we say that the program is generalising. After being trained with the data-set (i.e., after learning from the data-set), the program can take an unseen x and correctly map it to the corresponding target t (without knowing what the target is). As we can see, the program has generalised the data-set because it has learned from examples a general concept. Also note that, the function f that the program learns is only an approximation of the actual function that generated the data set D.

Example

Let us consider an example to better understand how supervised learning works. Say we want to create a program that recognises photos of apples and oranges (i.e., given a photo of an apple or an orange, say if it's one or the other). We can start training the program giving it some photos of apple and oranges and telling it if each photo is an apple or an orange. After many examples, if we give the program an unseen photo, it can tell if it's an apple or an orange.

Types of supervised learning

Supervised learning problems can be divided in

- **Classification**, if the target has a finite number of values (i.e. it's discrete). The apple-orange example is an instance of classification, in fact $t \in \{Apple, Orange\}$.
- **Regression**, if the target has continuous values. An example of a regression problem is telling the age of a person in a photo (one might argue that t has a limited number of values, but we can imagine that there is no limit on the age of a person). Another example could be, given a set of points in a 2D space, finding the function (which is continuous) that generated them.
- **Probability estimation**, if the target has continuous values. This case is very similar to regression, but the main difference is that f has to obey to the probability rules (i.e. the integral of f has to be 1).

Induction biases

Supervised learning leverages the induction biases that come from previous knowledge. Consider the apple-orange example. If the program were trained to recognise the colour of the background, it would have performed badly for our purposes, because it was biased by previous experience. Since we have trained our program to recognises fruits, we can leverage the fact that the program has restricted the features to search to actually recognise apple and oranges. To clarify this concept, say we didn't told the program what to look for in the input images. It could look for the background colour or the fruit represented. Learning in this way would be too expensive because we aren't helping the program in solving the problem we want to solve.

Representation of the data

In supervised learning, it's fundamental to represent the data correctly to extract knowledge, because we need to have information relevant for the domain of application (i.e. for the problem we want to solve). In other words we want to represent the data so that it can be efficiently used to solve the problem we care about.

1.2.2 Unsupervised learning

Unsupervised learning allows a program to learn a better representation (i.e., a structure) for the input data. The main difference with respect to supervised learning is that the data-set is made only of inputs

$$D = \{x\}$$

Types of unsupervised learning

Some of the most important types of unsupervised learning problems are

- **Domain reduction**. Domain reduction allows to reduce the domain of the input data without losing too much information. Consider for instance a data-set made of points in a three-dimensional space, i.e., each x is a three-dimensional point (x_1, x_2, x_3). The points might be disposed along a line (in one dimension), thus, through appropriate transformations, we could represent the same points as a line in 2D and characterise each point with just two coordinates.
- **Clustering**. Clustering allows to recognise if some data is closer to each other. In other words the program tries to find groups of inputs that are close one another. An example of clustering is customer classification, in fact customers with the same preferences are grouped because are close (i.e. have similar preferences in terms of products bought).

1.2.3 Reinforcement learning

Reinforcement learning allows a program to learn an optimal way of behaving, i.e., an optimal policy π. In this case the training set (i.e. the data-set) is much more complex, in fact we need

- The **state** x in which the program is.
- The **action** u the program executes.
- The **state** x' to which the program goes when applying action u in state x.
- The **reward** r for transitioning from x to x' using action u. The reward is used by the program to learn for each state x what is the best policy $\pi^*(x)$, that is the best action to execute from such state.

$$D = \{(x, u, x', r)\}$$

Learning

The most important thing to understand in supervised learning is that the program should learn what is the action that maximises the collection of future rewards. This means that the program should be able to sacrifice immediate rewards in favour of future positive rewards.

$$D \Rightarrow \pi^*(D) = \arg\max_u \{Q^*(x, u)\}$$

Notice that reinforcement learning gives empirical results, that is the policy might not always lead to the best possible outcome.

Domains of application

Reinforcement learning is used when the problem we are trying to solve isn't fully known.

1.3 Dichotomies

Machine learning let us choose among different approaches, each of which has its own advantages and disadvantages. In particular, we can have

- **Parametric vs non-parametric** approaches.
- **Frequentist vs Bayesian** approaches.
- **Generative vs discriminative** approaches.
- **Empirical risk minimization vs structural risk minimization**.

1.3.1 Parametric and non-parametric approaches

Machine learning algorithms can be

- **Parametric**.
- **Non-parametric**.

Parametric approaches

Parametric approaches use parameters to predict new values. More precisely,

> **Definition 1.2** (Parametric algorithm)**.** *Parametric algorithms use a fixed and finite number of parameters whose values are learned thanks to the data-set D. After learning the value of the parameters we don't need the data-set D anymore and the learned parameters can be used to predict the value of unseen data.*

Say, for instance, that we have a set of points generated by an unknown function $\bar{f}(x)$ that we want to approximate. We can decide to use a linear model $f(x) = ax^2 + bx$ and learn the values of the parameters a and b. After learning the values of the parameters we can discard the data-set and use $f(x)$ to predict the value $y = f(x)$ of a new input x.

Non-parametric approaches

Non-parametric approaches don't rely on parameters. More precisely,

> **Definition 1.3** (Non-parametric algorithm)**.** *Non-parametric algorithms use the data-set D every time they have to predict the value of unseen data.*

Put it in another way, the number of parameters in non-parametric algorithms depends on the training set.

Consider, for instance, that we have a set of classified points (i.e., our data-set), where each point either belongs to class C_0 or C_1. To classify a new point we can, for instance, consider the distance from the closest point of class C_0 and C_1. If the new point is closer to a point of C_0 than to one of C_1, then the new input is assigned to class C_0. As we can see that we have used the whole data set to classify a new point.

1.3.2 Frequentist vs Bayesian

Machine learning algorithm can follow a

- **Frequentist** approach. In this case, the algorithm uses probabilities to model the *sampling* process.
- **Bayesian** approach. In this case, the algorithm uses probability to model *uncertainty* about the estimate.

Bayesian approaches usually outperform their frequentist counterpart if we have only a few samples.

1.3.3 Generative vs discriminative

Machine learning algorithm can follow a

- **Generative** approach. In this case, the algorithm learns the *joint probability distribution*

$$p(x, t)$$

- **Discriminative** approach. In this case, the algorithm learns the *conditional probability distribution*

$$p(t|x)$$

1.3.4 Empirical risk minimization vs structural risk minimization

Machine learning algorithm can use

- **Structural risk minimization**. In this case, the algorithm minimises the error over the training set D.
- **Structural risk minimization**. In this case, the algorithm balances training error and model complexity.

Part II

Supervised learning

Chapter 2

General description

Supervised learning is the most mature field of machine learning. It aims at finding a good approximation $f(x)$ of a function $\bar{f}(x) \to t$ that takes as input the inputs of a data-set D and produces the respective output. The main concept of this definition is the fact that we are trying to find an approximation. This means that, our goal is finding a function that doesn't only works well on the data-set D but also on unseen inputs.

Data-set Before diving into supervised learning, let us specify the data-set D. In particular each couple (x, t) of the data-set is made of

- An input variable or **feature** x (also called predicator or attribute).
- An output variable or **target** t (also called response or label). Depending on the domain of t we can classify a problem as classification (t is discrete), regression (t is continuous) or probability estimation (t is a probability).

When to use supervised learning Machine learning offers a multitude of different approaches to solve a problem, thus it's useful to understand when to use supervised learning. In particular, supervised learning is useful when

- It doesn't exists a human expert. An example is DNA analysis.
- Humans perform efficiently in a task but we can't say how. An example is face recognition; we can easily tell when an image depicts a face but we can't put on paper the process we use to do so.
- The function to approximate changes frequently.
- There are many different functions f to approximate. An example is spam filters, in fact every person needs a personalised spam filter (i.e. a personalised function that tells if a mail is spam or not).

2.1 Graphical description

As for now we have given a rather abstract definition of supervised learning. Let's try to specify what does it mean to approximate a function f.

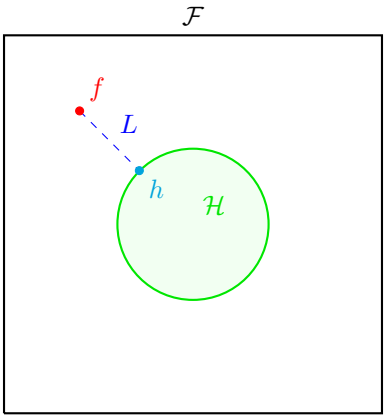


Figure 2.1: A graphical representation of the search space of a supervised learning algorithm.

Say we have a space $\mathcal{F}$ that contains all the possible function from the domain of x to the domain of t. The function f we are trying to approximate belongs to $\mathcal{F}$ but we don't know it, thus we have to approximate it using a data-set D. To reach this goal, every supervised learning algorithm needs

- A **loss function** $\bar{L}$ that return the distance between a tentative approximation h and f. Namely, this function measures how good a guess h (i.e. a function) is with respect to f.
- An **hypothesis space** $\mathcal{H}$. An hypothesis space is a subset of the function space $\mathcal{F}$ in which we search the best approximation of f. Supervised learning algorithms need $\mathcal{H}$ because $\mathcal{F}$ might be too big to explore, thus we have to reduce the search space. Notice that, f might be outside $\mathcal{H}$.
- An **algorithm** that minimises the loss function. In practice, the algorithm has to find a function $h \in \mathcal{H}$ with the smallest value of $\bar{L}$ (i.e. with the smallest distance from f). The function h **is the approximation of** f. Notice that h belongs to the search space $\mathcal{H}$.

A graphical representation of $\bar{L}$, $\mathcal{H}$ and h is shown in Figure 2.1. In other words, to solve a supervised learning problem we have to define

- The **representation** of the problem (i.e., $\mathcal{H}$).
- The **evaluation** of the model (i.e., L).
- The **optimization** of the evaluation.

Each of these characteristics can be implemented in different ways, in particular

- The loss function has to be easily optimised and can be, for instance, implemented with the mean square loss and the accuracy.
- The hierarchical space can be implemented, for instance, with neural networks.
- The optimisation depends on the loss-function.

2.1.1 Approximating the loss function

The main problem with the model defined above is that the loss function can't be computed. Namely, we don't know where f is in $\mathcal{F}$, thus we can't compute the distance between a guess h^* and f. This means that we should also try to approximate L using the data-set D. Namely, we should find an approximation L of the real loss function $\bar{L}$ that uses D to estimate the distance between a guess h^* and f.

Magnitude of the approximation

Since L is an approximation of the actual loss function, we should try to understand how good L is. In general, the smaller the data-set is, the noisier (i.e. the less precise) the approximation is, however we would like to precisely understand how wrong the approximation is. Intuitively, if we have few samples in D, the learning algorithm has less examples to improve its experience and approximate the loss function.

Enlarging the search space

One might think that enlarging the search space might lead to a better approximation h. This isn't true, in fact in some cases we could get an approximation h_2 found in a bigger space $\mathcal{H}_2$ that is further then $h_1 \in \mathcal{H}_1$. This seems counter-intuitive but we have to remember that the estimation h depends on D, hence we could get a function h which is really good at modelling the points in the data-set but not the actual function f. This phenomenon is called variance. The opposite of variance is bias, in fact the smaller the hypothesis space, the bigger the bias. In other words the bias is the distance between the hypothesis space and f. The main goal is to balance bias and variance to get the best hypothesis space possible. Some methods to balancing these two quantities will be explained later on.

2.1.2 Problem representation

Let us go back to the main components required to solve a supervised learning problem. The first thing we should do when approaching a supervised learning problem is defining how to represent the approximation of the function f to approximate. Basically, we have to decide a class of functions $\mathcal{H}$, i.e., a model, that contains the set of functions among which we can choose our approximation h. Some examples of representations are

- **Linear models**.
- **Instance-based models**.
- **Decision trees**.
- **Set of rules**.
- **Graphical models**.
- **Neural networks**.
- **Gaussian Processes**.
- **Support vector machines**.

- **Model ensembles**.

Let us consider liner models, which are easy to treat. A linear model could be $t = ax^2 + bx + c$, where a, b and c are the parameters to optimise through the optimisation of the model evaluation.

2.1.3 Model evaluation

After defining a model we have to evaluate it, hence we have to define an evaluation metric. Some examples are

- **Accuracy**.
- **Precision and recall**.
- **Squared error**.
- **Likelihood**.
- **Posterior probability**.
- **Cost or utility**.
- **Margin**.
- **Entropy**.
- **KL divergence**.

2.1.4 Optimisation

Now that we know which metric of the model should be optimised, we can define a way to optimise it. The three main approaches we can use are

- **Combinatorial optimisation**. An example of combinatorial optimisation is greedy search.
- **Convex optimisation**. Convex optimisation starts from one point and improves it at small steps until an optimum (local or global) is reached. An example of convex optimisation is gradient descent.
- **Constrained optimisation**. Constrained optimisation allows to express an optimisation problem as a function to optimise and a set of constraints to satisfy. An example of constrained optimisation is linear programming.

Chapter 3

Linear models for regression

3.1 Linear regression

Before describing in details how linear regression works, let us introduce the data-set D we are going to use. The data-set $D = \{(\mathbf{x}, t)\}$ is a set of samples each made of a feature $\mathbf{x}$ and a target t generated by a function f and some noise. This means that the outputs t we see are not the same generated by f, since the noise could have changed the data. In formulas we can write $t = f(x) + n$ where n is some noise.

Properties Linear regression models have some distinctive and important properties, in particular

- The **loss function is easy to optimise**.
- They **are the foundations of more advanced models**.
- They **are linear**. Notice that the linearity has to be enforced on the loss function, not on the input features, this means that L has to be linear but we can use a non linear feature (e.g. $\phi = x^2$).

3.1.1 Model

The linear regression model $y(\mathbf{x}, \mathbf{w})$ is the linear combination of the data-set's features and the parameters $\mathbf{w}$ of the model

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{D-1} w_j x_j \tag{3.1}$$

where

- $\mathbf{x} = \{1, x_1, \ldots, x_{D-1}\}$ is the vector of features. This means that each input in the data-set is described in D dimensions by $\mathbf{x}$. Considering a three dimensional space could make things clearer. If we call x_1 and x_2 the input axis and z the output axis, then an input point $\mathbf{x}_1$ can be represented as $(x_{1,1}, x_{1,2})$. Remember that $\mathbf{x}$ is a single data input.
- $\mathbf{w} = \{w_0, w_1, \ldots, w_{D-1}\}$ is the vector of parameters w_i. Note that, the parameter's vector has the same cardinality of the feature vector. The goal of the learning algorithm is to find the values of the parameters w that minimise the loss function. Notice that y isn't the loss

function, it represents the search space $\mathcal{H}$ in which we search. In particular $\mathcal{H}$ is the space of linear function (linear in the parameters) with parameters D parameters.

- D is the number of parameters of the model. $\mathbf{w}$ is the core of the model, in fact if we change $\mathbf{w}$ we change the model, too.
- w_0 is the open bias (but is not related to variance and bias). Since w_0 is used outside the sum, we start summing from $j = 1$ and reach $D - 1$ to consider all D parameters.

Let us clarify one important thing. The model y is a linear function that, given a point $\mathbf{x}$ (and the parameters $\mathbf{w}$) returns the predicted value for that point. Before going on, let us further clarify the notation:

- $\mathbf{x} = \{x_0, \ldots, x_{D-1}\}$ is a generic point.
- $\mathbf{x}_i = \{x_{i,0}, dots, x_{i,D-1}\}$ is an input point.
- t_i is an output point (i.e., a target). Note that t_i is a scalar.

This means that the data set can be written as $D = \{(\mathbf{x}_i, t_i)\}$.

Matrix notation

Since $\mathbf{x}$ and $\mathbf{w}$ are vectors, Equation 3.1 can be written in matrix form as

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} = \mathbf{x}^T\mathbf{w} \tag{3.2}$$

From this formulation we can notice that $\mathbf{x}$ must have the same number of elements of $\mathbf{w}$, in particular

$$\mathbf{x} = (1, x_1, x_2, \ldots, x_{D-1})^T$$

The first element is 1 because it's x_0 and is multiplied by w_0 to obtain the open bias, which is $1 \cdot w_0$.

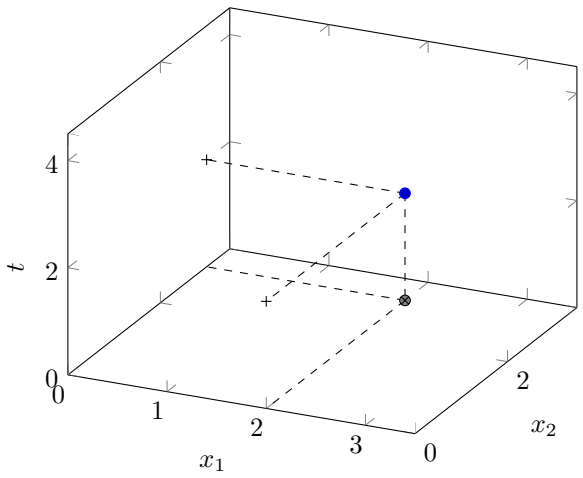


Figure 3.1: A representation of a sample in a 3-dimensional space (2 dimensions for the input, one for the target).

3.1.2 Loss function

Now that we have defined the search space, we have to define a loss function $L(t, y(\mathbf{x}))$ that defines how good a guess is (i.e. how far $y(\mathbf{x})$ is from f). Notice that the loss function takes as input the targets t of the data-set and the model $y(\mathbf{x})$ through which we have to search.

The loss function L can be used to compute the average distance of all samples $\mathbf{x_i}$ in the data-set; in particular we can use the expected value of L

$$\mathbb{E}[L] = \int\int L\big(t, y(\mathbf{x})\big)p(\mathbf{x}, t)\ d\mathbf{x}dt \tag{3.3}$$

where

- $L\big(t, y(\mathbf{x})\big)$ is the loss function.
- $p(\mathbf{x}, t)$ is the probability that a sample $(\mathbf{x}, t)$ is generated.

Mean squared loss function

As for now we have considered a general loss function L, let us now introduce a practical function. A widely used loss function is the mean square loss function that considers the squared distance between the targets and the model

$$L(t, y(\mathbf{x})) = \big(t - y(\mathbf{x})\big)^2 \tag{3.4}$$

This function can be used to compute the average as

$$\mathbb{E}[L] = \int\int \big(t - y(\mathbf{x})\big)^2 p(\mathbf{x}, t)\ d\mathbf{x}dt \tag{3.5}$$

The optimal solution of $y(\mathbf{x})$ that minimises $L(\mathbf{x})$ is called **conditional average**

$$y(\mathbf{x}) = \int t \cdot p(t|\mathbf{x})\ dt = \mathbb{E}[t|\mathbf{x}]$$

Minkowski loss function

The mean square loss function can be generalised, in particular we can define a class of functions

$$L(t, y(\mathbf{x})) = \big|t - y(\mathbf{x})\big|^q \tag{3.6}$$

where for each value of q we obtain a different function, for instance

- For $q = 2$ we obtain the **mean squared loss**.
- For $q = 1$ we obtain the **median error**.
- For $q \to 0$ we obtain the **conditional mode**.

3.1.3 Basis functions

When introducing linear models we said that the model (i.e. the parameters) has to be linear but the features can be non-linear. For this reason we can try to generalise the linear model and include non linear values of $\mathbf{x}$. To do so we can rewrite $y(\mathbf{x}, \mathbf{w})$ as

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \cdot \phi_j(\mathbf{x}) \tag{3.7}$$

where $\phi(\mathbf{x})$ is a function (linear or not) called **basis function**. As before we can rewrite the model in matrix form to obtain

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) \tag{3.8}$$

where

$$\boldsymbol{\phi}(\mathbf{x}) = \{1, \phi_1(\mathbf{x}), \ldots, \phi_{M-1}(\mathbf{x})\}$$

Thanks to this more general model we can improve the approximation, but we also have to remember that we are adding more points, thus we are increasing the variance of the model. Note that, in this case we are increasing the number of parameters $\mathbf{w}$ (i.e., we move from D to M parameters). Moreover, we have defined a set of functions $\phi_j(\mathbf{x})$, each of which takes the point $\mathbf{x} = \{1, x_0, \ldots, x_{D-1}\}$ and uses its coordinates to create a new feature. Let us consider a simple feature vector $\mathbf{x} = \{1, x\}$. Also note that we can write the model in Equation 3.1 as 3.7, if we consider a basis vector $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$. Some examples of basis functions are

- The **polynomial function**
$$\phi_i(\mathbf{x}) = x^i$$
- The **Gaussian function**
$$\phi_i(\mathbf{x}) = \exp\left(-\frac{(x - \mu_i)^2}{2\sigma^2}\right)$$
- The **sigmoidal function**
$$\phi_i(\mathbf{x}) = \frac{1}{1 + \exp\left(\frac{\mu_i - x}{\sigma}\right)}$$

3.2 Minimising the loss function

3.2.1 Approaches

There exists many approaches to minimise the loss function,

- **Generative**. The generative approach tries to learn the joint distribution (i.e., the probability of $\mathbf{x}$ and t), in particular we have to
  1. Model the joint density $p(\mathbf{x}, t) = p(\mathbf{x}|t)p(t)$.
  2. Use the joint density to infer the conditional density $p(t|\mathbf{x}) = \frac{p(\mathbf{x},t)}{p(\mathbf{x})}$.
  3. Marginalize to find the conditional mean $\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\ dt$.
- **Discriminative**. The discriminative approach tries to learn the probability of a target given an input. In particular we have to

1. Model conditional density $p(t|\mathbf{x})$.
2. Marginalize to find the conditional mean $\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\ dt$.

The discriminative method we are going to analyse is **maximum likelihood**.

- **Direct**. The direct approach straightforwardly finds f from the data-set. The direct method we are going to analyse is **mean square minimisation**.

3.2.2 Mean square minimisation

Before optimising the loss function, we have to rewrite it considering that the data-set is made of a finite number N of samples $\mathbf{x}_1, \dots, \mathbf{x}_N$. The loss function defined in Equation 3.5 can be rewritten as

$$L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(y(\mathbf{x}_n, \mathbf{w}) - t_n\Big)^2 \tag{3.9}$$

This function is called **Sum of Squared Errors** (SSE), in fact is sums the difference (i.e. the error) between our model (i.e., y) and the target (i.e., t). The SSE is the half of the Residual Sum of Squares (RSS). The loss function $L(\mathbf{w})$ can be also written in matrix form using Equation 3.2 as

$$L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(\mathbf{x}_n^T\mathbf{w} - t_n\Big)^2 \tag{3.10}$$

or, more in general, using basis functions as

$$L(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(\mathbf{w}^T\phi(\mathbf{x}_n) - t_n\Big)^2 \tag{3.11}$$

Another representation of the loss function $L(\mathbf{w})$ is

$$L(\mathbf{w}) = \frac{1}{2}\|\boldsymbol{\varepsilon}\|_2^2 = \frac{1}{2}\sum_{i=1}^{N}\varepsilon_i^2 \tag{3.12}$$

Where $\boldsymbol{\varepsilon} = \{\varepsilon_1, \dots, \varepsilon_N\}$ is the vector of residual errors $\varepsilon_i = \mathbf{w}^T\phi(\mathbf{x}_i) - t_i$.

This loss function can be very different from the actual loss function we are trying to estimate. Remember that the true loss function can't be computed since we don't know the actual function f to approximate, hence we are using the points in the data-set (i.e., $\mathbf{x}_i$ and t_i) to approximate the loss function. Finally, if we write all the targets in matrix form (i.e., $\mathbf{t} = (t_1, \dots, t_N)$), we can write $L(\mathbf{w})$ as

$$L(\mathbf{w}) = \frac{1}{2}RSS(\mathbf{w}) \tag{3.13}$$

$$= \frac{1}{2}\Big(\mathbf{t} - \Phi\mathbf{w}\Big)^T\Big(\mathbf{t} - \Phi\mathbf{w}\Big) \tag{3.14}$$

where

- $\mathbf{t} = (t_1, \dots, t_N)^T$ is the column vector of targets (i.e., all the targets in the data-set).
- $\boldsymbol{\phi} = (\phi_0(\mathbf{x}), \dots, \phi_{M-1}(\mathbf{x}))^T$ is the usual $M \times 1$ column vector of the basis functions.

- $\mathbf{w}$ is the $M \times 1$ column vector of the parameters.
- $\Phi = (\boldsymbol{\phi}(\mathbf{x_1}), \ldots, \boldsymbol{\phi}(\mathbf{x_N}))^T$ is the $N \times M$ matrix in which each of the N rows (one for each input sample), is a feature vector. In particular the matrix has as many rows as number of samples in the data-set and as many columns as the number of features of each sample. This matrix is like a database, in fact each row represents a sample and each column represents a feature.
- $\Phi\mathbf{w}$ is the prediction vector. The prediction vector contains N elements, in fact it's obtained as the multiplication of a $N \times M$ matrix and a $M \times 1$ matrix (a vector, actually). $\Phi\mathbf{w}$ is the matrix equivalent of y.
- $\mathbf{t} - \Phi\mathbf{w}$ is the vector of errors, i.e. the vector that specifies for each sample the error between the model (i.e., $\Phi\mathbf{w}$) and the target $\mathbf{t}$. As expected, this vector contains N elements.

Also notice that multiplying $\mathbf{t} - \Phi\mathbf{w}$ for its transpose is like elevating it to the power of two (like we did in Equation 3.9).

Given $L(\mathbf{w})$ we want to find the values of $\mathbf{w}$ that minimise the error (i.e. the loss function). This goal can be achieved using the gradient of L with respect to $\mathbf{w}$. Let's start by deriving L in $\mathbf{w}$ to obtain

$$\frac{\delta L(\mathbf{w})}{\delta \mathbf{w}} = \frac{\delta}{\delta \mathbf{w}} \frac{1}{2} \left[\left(\mathbf{t} - \Phi\mathbf{w} \right)^T \left(\mathbf{t} - \Phi\mathbf{w} \right) \right] \tag{3.15}$$

$$= \frac{1}{2} \left[\left(\frac{\delta}{\delta \mathbf{w}} (\mathbf{t} - \Phi\mathbf{w}) \right)^T \cdot (\mathbf{t} - \Phi\mathbf{w}) + \left(\frac{\delta}{\delta \mathbf{w}} (\mathbf{t} - \Phi\mathbf{w}) \right) \cdot (\mathbf{t} - \Phi\mathbf{w})^T \right] \tag{3.16}$$

$$= \frac{1}{2} \left[-\Phi^T \cdot (\mathbf{t} - \Phi\mathbf{w}) - \Phi \cdot (\mathbf{t} - \Phi\mathbf{w})^T \right] \tag{3.17}$$

$$= \frac{1}{2} \left[-\Phi^T \cdot (\mathbf{t} - \Phi\mathbf{w}) - \Phi^T (\mathbf{t} - \Phi\mathbf{w}) \right] \tag{3.18}$$

$$= -\Phi^T (\mathbf{t} - \Phi\mathbf{w}) \tag{3.19}$$

Now we have to remember that a function has a maximum or a minimum when the derivative is 0, thus we can set $\frac{\delta L(\mathbf{w})}{\mathbf{w}}$ to zero to find the values of $\mathbf{w}$ that minimise $L(\mathbf{w})$

$$\frac{\delta L(\mathbf{w})}{\delta \mathbf{w}} = 0 \tag{3.20}$$

$$-\Phi^T (\mathbf{t} - \Phi\mathbf{w}) = 0 \tag{3.21}$$

$$-\Phi^T \mathbf{t} + \Phi^T \Phi\mathbf{w} = 0 \tag{3.22}$$

$$\Phi^T \Phi\mathbf{w} = \Phi^T \mathbf{t} \tag{3.23}$$

$$\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{t} \tag{3.24}$$

This means that the optimal parameters $\mathbf{w}$ can be computed as

$$\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{t} \tag{3.25}$$

The points in which the derivative is null can be both max or min points, thus we should check when a $\mathbf{w}$ is a minimum. This involves computing the second derivative and checking if has non-zero

determinant (i.e., it's non-singular). The second derivative is

$$\frac{\delta^2 L(\mathbf{w})}{\delta \mathbf{w} \delta \mathbf{w}^T} = -\Phi^T \frac{\delta}{\delta \mathbf{w}}(\mathbf{t} - \Phi \mathbf{w}) \tag{3.26}$$

$$= -\Phi^T \cdot -\Phi \tag{3.27}$$

$$= \Phi^T \Phi \tag{3.28}$$

Computational complexity

The complexity of finding the optimal parameter vector is bounded by the complexity of inverting matrix $\Phi^T\Phi$ (NM^2) and multiplying $(\Phi^T\Phi)^T$ and Φ^T (M^3). Therefore, the complexity for minimising the loss function is

$$\mathcal{O}(NM^2 + M^3)$$

3.2.3 Stochastic gradient descend

Computing the parameter vector using Equation 3.25 can be very expensive for data-intensive application, thus we should find another way to optimise $L(\mathbf{w})$. Stochastic gradient descend comes in our help in such cases. In particular, this technique follows the direction of the gradient (i.e. moves in the space following the gradient) until a minimum is reached. The learning algorithm starts from a random value of $\mathbf{w}$ and iteratively improves it using (following) the gradient. In formulas, the value of $\mathbf{w}$ at time $k+1$ is the value at time k minus the gradient multiplied by a learning rate α (that depends on time k)

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha^{(k)} \nabla L(\mathbf{x}_n) \tag{3.29}$$

If we expand the gradient (using Equation 3.19) we obtain the complete form

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha^{(k)} \Big(\mathbf{w}^{(k)T} \boldsymbol{\phi}(\mathbf{x}_n) - t_n \Big) \boldsymbol{\phi}(\mathbf{x}_n) \tag{3.30}$$

Notice that stochastic gradient descend is an **online** optimisation method, in fact the parameter vector is updated using a single input sample $\mathbf{x}_n$, hence we can keep updating $\mathbf{w}$ with every new input point we get.

Convergence This technique can be applied only if the loss function can be expressed as a sum over samples, i.e.

$$L(\mathbf{x}) = \sum_n L(x_n)$$

Moreover, stochastic gradient descend converges to a minimum only if

- The learning rate α doesn't decrease too fast, otherwise we stop before reaching the minimum.

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

- The learning rate α doesn't decrease too slowly, otherwise we never stop.

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

3.2.4 Maximum likelihood

Maximum likelihood is a probabilistic technique that allows to find the model m (i.e. the function f, i.e. the parameters $\mathbf{w}$) that maximises the probability that the input data has been generated by model m.

Maximum likelihood has to take into consideration the noise ε, thus we have to write the target t as a function of the inputs x and the error ε (i.e. the noise)

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

Noise Noise can have many forms. We are going to consider white noise, i.e., a noise with Gaussian distribution, 0 mean and average σ^2.

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

Likelihood

Given an input $\mathbf{x}$ of a data-set and the respective output t, we can write the probability of t, given input model $\mathbf{x}$, parameters $\mathbf{w}$ and the variance of the noise σ^2 as

$$p(t|\mathbf{x}, \mathbf{w}, \sigma^2) = \mathcal{N}\big(t|y(\mathbf{x}, \mathbf{w}), \sigma^2\big) \tag{3.31}$$

$$= \mathcal{N}\big(t|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}), \sigma^2\big) \tag{3.32}$$

Basically, we are modelling each the probability of t as a Gaussian of mean $\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x})$ and variance σ^2.

Now we can consider a data-set made of N samples, with inputs $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$, and outputs $\mathbf{t} = \{t_1, ..., t_N\}^T$. Since each output t_i is independent from the others, the probability distribution of $\mathbf{t}$ can be computed as the product of the probability distributions of each t_i, hence the likelihood function p is

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(t_i|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i), \sigma^2) \tag{3.33}$$

This probability is the probability of observing all target values $\mathbf{t}$ given the inputs $\mathbf{X}$ and the model (i.e. $\mathbf{w}$). Notice that we can compute p multiplying the probabilities because their are independent, in fact each target doesn't depend on other inputs (i.e. t_i depends only on $\mathbf{x}_i$ and the model). In practice, we are assuming that the samples in the data set have a normal distribution and we want to find the Gaussian distribution $\mathcal{N}(y(\mathbf{x}, \mathbf{w}), \sigma^2)$ that better models the samples, namely, we want to find the mean (i.e., $y(\mathbf{x}, \mathbf{w})$) around which the points are centred that maximises. Basically, we want to find the parameters $\mathbf{w}$ for which the probability that the samples have been generated by $\mathcal{N}(y(\mathbf{x}, \mathbf{w}), \sigma^2)$ is maximum. Put it in another way we want to find the distribution $\mathcal{N}(y(\mathbf{x}, \mathbf{w}), \sigma^2)$ that better describes the probability that a new sample has an output t.

Logarithmic likelihood Usually it's convenient to consider the logarithmic likelihood instead of the likelihood because we can transform products in sums. In particular the logarithmic likelihood is

$$\ell(\mathbf{w}) = \ln\Big(p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2)\Big) = \sum_{i=1}^{N} \ln\Big(\mathcal{N}(t_i|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i), \sigma^2)\Big) \tag{3.34}$$

$$= -\frac{N}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}RSS(\mathbf{w}) \tag{3.35}$$

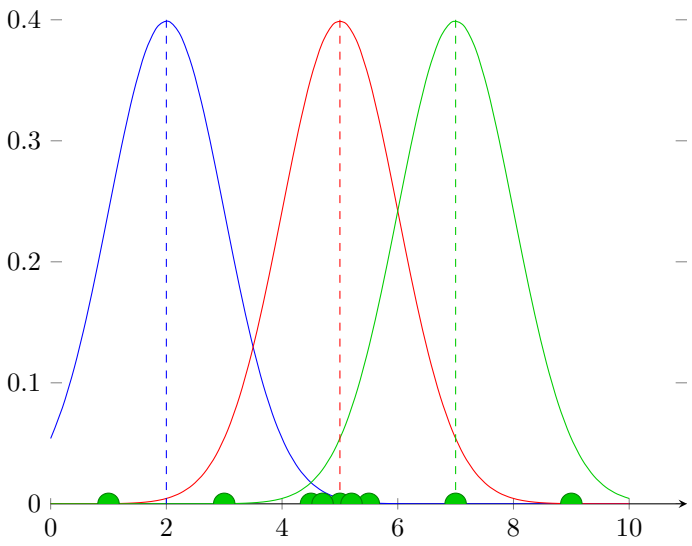


Figure 3.2: Three normal distribution that try to model a one-dimensional data set (e.g., the number of puppies of a dog). The middle distribution (in red) better models the data-set since the samples are centred around its mean, hence the probability that a sample is close to the mean is higher.

To obtain the result in Equation 3.34 we have simply replaced the probability distribution of a normal random variable (see Appendix A.4).

$$\ell(\mathbf{w}) = \ln\left(p(\mathbf{t}|\mathbf{X},\mathbf{w},\sigma^2)\right) = \sum_{i=1}^{N} \ln\left(\mathcal{N}(t_i|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i),\sigma^2)\right) \tag{3.36}$$

$$= \sum_{i=1}^{N} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_i-\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i))^2}{2\sigma^2}}\right) \tag{3.37}$$

$$= \sum_{i=1}^{N} \left(-\frac{(t_i-\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i))^2}{2\sigma^2} - \ln\sqrt{2\pi\sigma^2}\right) \tag{3.38}$$

$$= -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(t_i-\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i))^2 - \sum_{i=1}^{N}\ln\sqrt{2\pi\sigma^2} \tag{3.39}$$

$$= -\frac{1}{2\sigma^2}RSS(\mathbf{w}) - N\ln\sqrt{2\pi\sigma^2} \tag{3.40}$$

$$= -\frac{1}{2\sigma^2}RSS(\mathbf{w}) - N\frac{2}{2}\ln\sqrt{2\pi\sigma^2} \tag{3.41}$$

$$= -\frac{1}{2\sigma^2}RSS(\mathbf{w}) - \frac{N}{2}\ln(\sqrt{2\pi\sigma^2})^2 \tag{3.42}$$

$$= -\frac{N}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}RSS(\mathbf{w}) \tag{3.43}$$

Notice that using the logarithmic likelihood instead of the likelihood only shifts the value of the maximum but not its location. Since we are interested in the location (i.e. on the values of $\mathbf{w}$), we

can use $\ell(\mathbf{w})$. Now that we have the likelihood we can compute its gradient.

$$\nabla\ell(\mathbf{w}) = \frac{\delta\ell(\mathbf{w})}{\delta\mathbf{w}} \tag{3.44}$$

$$= -\frac{\delta}{\delta\mathbf{w}}\left[\frac{N}{2}\ln(2\pi\sigma^2) + \frac{1}{2\sigma^2}RSS(\mathbf{w})\right] \tag{3.45}$$

$$= -\frac{\delta}{\delta\mathbf{w}}\left[\frac{N}{2}\ln(2\pi\sigma^2)\right] - \frac{\delta}{\delta\mathbf{w}}\left[\frac{1}{2\sigma^2}RSS(\mathbf{w})\right] \tag{3.46}$$

$$= -\frac{1}{2\sigma^2}\frac{\delta}{\delta\mathbf{w}}\left[RSS(\mathbf{w})\right] \tag{3.47}$$

To wrap things up the gradient of the logarithmic likelihood is

$$\nabla\ell(\mathbf{w}) = \Phi^T(\mathbf{t} - \Phi\mathbf{w}) \tag{3.48}$$

$$= \Phi^T\mathbf{t} - \Phi^T\Phi\mathbf{w} \tag{3.49}$$

$$= \sum_{i=1}^{N} t_i\phi(\mathbf{x}_i)^T - \mathbf{w}^T\left(\sum_{i=1}^{N}\phi(\mathbf{x}_i)\phi(\mathbf{x}_i)^T\right) \tag{3.50}$$

The maximum of ℓ can finally be computed putting the gradient to 0

$$\Phi^T\mathbf{t} - \Phi^T\Phi\mathbf{w} = 0 \tag{3.51}$$

$$\Phi^T\Phi\mathbf{w} = \Phi^T\mathbf{t} \tag{3.52}$$

$$\mathbf{w} = \left(\Phi^T\Phi\right)^{-1}\Phi^T\mathbf{t} \tag{3.53}$$

As we can see Equation 3.34 is basically the same, apart from a constant $-\frac{N}{2}\ln(2\pi\sigma^2)$ and a factor σ^2, of the expression obtained from the mean square (Equation 3.14). This means that with a different approach (a discriminative one) we can obtain the same result of the direct approach. This is confirmed by the fact that the solution obtained with the maximum likelihood approach (Equation 3.53) is the same as the one obtained with the mean square minimisation (Equation 3.25).

Variance

After computing the values of the parameters $\mathbf{w}$, we should compute how certain are we about such values. Better said, we want to know how confident are we that such values minimise the loss function. To do this job we can compute the variance $Var(\mathbf{w})$ that measures, for each parameter, the confidence we have it minimises the loss function. The variance matrix of the least-squares estimate can be written as

$$Var(\hat{\mathbf{w}}) = \left(\Phi^T\Phi\right)^{-1}\sigma^2$$

where

- $\hat{\mathbf{w}}$ are the estimates of the parameters.
- σ^2 is the variance of the error and can be estimated by

$$\sigma^2 = \frac{1}{(N-M)}\sum_{i=1}^{N}\left(t_i - \hat{\mathbf{w}}^T\phi(\mathbf{x}_i)\right)^2$$

Notice that, to compute Var and σ^2 we have assumed

- The observations t_i are uncorrelated and have constant variance σ^2.
- The inputs $\mathbf{x}_i$ are fixed (non random).
- The model is linear in the features $\boldsymbol{\phi}(\mathbf{x}_i)$.
- The noise is additive and Gaussian. Furthermore if the error is Gaussian then the parameters are also distributed as a Gaussian.

The variance Var allows us to obtain an important result:

Theorem 3.1 (Gauss-Markov)**.** *The least squares estimate of* $\mathbf{w}$ *has the smallest variance among all linear unbiased estimates.*

This means that, the least squares estimator has the lowest Minimum Square Error of all linear estimator with no bias. However it might exist an estimator that is biased but has a smaller variance. Usually it's better to have biased estimators because they work better with small data-sets.

3.2.5 Multiple outputs

As for now we have considered data-sets in which each sample $(\mathbf{x}_i, t_i)$ had only one output t_i. If we allow a target to have multiple outputs, then the vector of targets $\mathbf{t} = (t_1, \ldots, t_N)^T$ becomes a matrix $\mathbf{T} = \mathbf{t}_0, \ldots, \mathbf{t}_{M-1}$. This means that we obtain a matrix $\hat{\mathbf{W}}$

$$\hat{\mathbf{W}} = \left(\Phi^T\Phi\right)^{-1}\Phi^T\mathbf{T}$$

This is the same as computing $\hat{\mathbf{w}}$ for each column vector $\mathbf{t}_i$ in T

$$\hat{\mathbf{w}}_i = \left(\Phi^T\Phi\right)^{-1}\Phi^T\mathbf{t}_i$$

In other words the solution can be decoupled and computed for each output separately. Also notice that the matrix $\left(\Phi^T\Phi\right)^{-1}\Phi^T$ has to be computed only once because it doesn't depend on the particular output we are considering.

3.2.6 Under-fitting and over-fitting

Variance and over-fitting

One might think that, increasing the number of feature of an input $\mathbf{x}_i$ (consider, for instance, $\mathbf{x}_i = (x, x^2, x^3, x^4)^T$ instead of $\mathbf{x}_i = (x, x^2)^T$) might generate a better estimator. However, this is not true in fact we are building a model that is too close to the target but far from the actual model to estimate. This happens because we are considering an approximation of the loss function based on the targets. This problem is defined **over-fitting** and it verifies when we have a big variance. In particular **variance tells us how much a model changes if we train it with different data sets**. Basically, if we have an high variance, we are learning the data-set by heart and we aren't approximating the real model.

Bias and under-fitting

Since increasing the number of features increases variance, which leads to over-fitting, one might think to reduce them. Reducing the number of feature increases however bias, which leads to under-fitting. Under-fitting means estimating a model which is too general and approximate.

A trade-off between variance and bias

Reducing variance brings to an increase in bias and vice-versa. This means that we have to build a model considering a trade-off between these quantities. The only way to improve a model without increasing neither of the two is to increase the number of samples.

3.3 Regularisation

Usually, the value of the parameters increases with the number of features (i.e. the more the features, the highest the values). This is a common behaviour for over-fitting because the estimator is trying to do quick changes to fit the targets t_i in the data-set. To solve this problem and mitigate over-fitting we can penalise high values for the parameters to get a smoother function since, when parameters have high values, the function changes rapidly.

Regularisation allows to change the loss function to penalise big parameters. The general form of the new loss function is

$$L(\mathbf{w}) = L_D(\mathbf{w}) + \lambda L_W(\mathbf{w}) \tag{3.54}$$

where

- $L_D(\mathbf{w})$ is the error on data (e.g. RSS).
- $L_W(\mathbf{w})$ is the loss that takes into consideration the complexity of the model.
- λ is a factor that discriminates big values of $\mathbf{w}$.

3.3.1 Ridge regression

If we consider $L_D(\mathbf{w}) = RSS(\mathbf{w})$ and

$$L_W(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}\|\mathbf{w}\|_2^2 \tag{3.55}$$

we obtain **ridge regression**. The complete form of the loss function for ridge regression, obtained replacing Equation 3.55 in the general Equation 3.54, is

$$L(\mathbf{w}) = \sum_{i=1}^{N} \big(t_i - \mathbf{w}^T\phi(\mathbf{x}_i)\big)^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \tag{3.56}$$

$$= \big(\mathbf{t} - \Phi\mathbf{w}\big)^T\big(\mathbf{t} - \Phi\mathbf{w}\big) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \tag{3.57}$$

Now that we have found the matrix-form of the loss function we can compute the gradient and put it to 0 to obtain

$$\mathbf{w} = \big(\lambda\mathbf{I} + \Phi^T\Phi\big)^{-1}\Phi^T\mathbf{t} \tag{3.58}$$

We can immediately notice that the expression to compute the parameters is similar to the one obtained without regularisation apart from a term $\lambda\mathbf{I}$ (where $\mathbf{I}$ is the identity matrix, hence $\lambda\mathbf{I}$ has λ on the diagonal). Note that, with ridge regression, the eigenvalues (B.3) of the matrix $(\lambda\mathbf{I} + \Phi^T\Phi)$ are all greater or equal than λ.

3.3.2 Lasso

Another regularisation technique is lasso. In this case we use $L_D(\mathbf{w}) = RSS(\mathbf{w})$ and

$$L_W(\mathbf{w}) = \frac{1}{2}\sum_{j=1}^{N}\mathbf{w}_i = \frac{1}{2}\|\mathbf{w}\| \tag{3.59}$$

The loss function for lasso regularisation can therefore been written as

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

Unfortunately, there isn't a closed form solution to compute $\mathbf{w}$.

Advantages and disadvantages

The main advantage of lasso is that when λ increases some parameters go exactly to 0 (i.e. exists some value of λ so that a parameter goes to 0), thus we can use lasso for feature selection (i.e. to select only those features that are useful for the problem). A consequence of this property is that lasso generates sparse models (i.e. with only few non null features). Ridge regularisation, on the other hand, allows only to reduce the value of parameters without really putting them to 0.

The main advantage of ridge is that it's linear in t_i and it has a closed-form solution. On the other hand lasso in non-linear in t_i and no closed form-solution exists (i.e. it's a quadratic programming problem).

3.4 Bayesian linear regression

Bayesian linear regression allows to formulate the knowledge about the world in a probabilistic way. In particular we have to

1. Define the model that expresses our knowledge qualitatively. Our model will have some unknown parameters that we have to learn.
2. Capture our assumptions about unknown parameters by specifying the prior distribution over those parameters before seeing the data. Thinking about the Bayes' rule, we want to model the probability $p(\mathbf{w})$.
3. Observe the data. Thinking about the Bayes' rule we observe $p(\mathcal{D}|\mathbf{w})$.
4. Compute the posterior probability distribution for the parameters, given observed data. If we translate this in the Bayes' world, we obtain

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathbf{w}) \cdot p(\mathcal{D}|\mathbf{w})}{p(\mathcal{D})}$$

5. Use the posterior distribution $p(\mathbf{w}|\mathcal{D})$ to
   - Make predictions by averaging over the posterior distribution.
   - Examine or account for uncertainty in the parameter values.
   - Make decisions by minimizing expected posterior loss.

3.4.1 Posterior distribution

A fundamental part of Bayesian linear regression is the posterior distribution for the parameters. This distribution can be computed only after observing the data. In particular the distribution of the parameters given the data $p(parameters|data)$ can be found, using the Bayes' rule, by combining the prior distribution of the parameters $p(parameters)$ with the likelihood for the parameters given data $p(data|parameters)$. In formulas

$$p(parameters|data) = \frac{p(parameters) \cdot p(data|parameters)}{p(data)}$$

or more concisely

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathbf{w}) \cdot p(\mathcal{D}|\mathbf{w})}{p(\mathcal{D})}$$

where

- $p(\mathbf{w}|\mathcal{D})$ is the **posterior distribution** of the parameters known the data (i.e. the data-set $\mathcal{D}$).
- $p(\mathbf{w})$ is the **prior distribution** of the parameters (i.e. the distribution of the parameters before using the data-set to learn them).
- $p(\mathcal{D}|\mathbf{w})$ is the probability (**likelihood**) of observing a data-set $\mathcal{D}$ given the parameters $\mathbf{w}$.
- $p(\mathcal{D})$ is the **marginal likelihood** computed as

$$p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})\ d\mathbf{w}$$

3.4.2 Bayesian linear regression and over-fitting

Bayesian regression can be used to reduce over-fitting, in particular it allows us to say how much we are uncertain about some prediction. To so do, we can assume that the parameters (i.e., the prior) have a Gaussian distribution

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{w}_0, \mathbf{S}_0)$$

If the likelihood is also Gaussian, i.e., the prior is a conjugate prior (see Appendix A.3), then the posterior distribution is also Gaussian

$$p(\mathbf{w}|\mathbf{t}, \Phi, \sigma^2) \propto \mathcal{N}(\mathbf{w}|\mathbf{w}_0, \mathbf{S}_0)\mathcal{N}(\mathbf{t}|\Phi\mathbf{w}, \sigma^2\mathbf{I}_N) = \mathcal{N}(\mathbf{w}|\mathbf{w}_N, \mathbf{S}_N)$$

In the formula above:

- $\mathcal{N}(\mathbf{w}|\mathbf{w}_0, \mathbf{S}_0)$ is the prior distribution.
- $\mathcal{N}(\mathbf{t}|\Phi\mathbf{w}, \sigma^2\mathbf{I}_N)$ is the likelihood.

- $\mathcal{N}(\mathbf{w}|\mathbf{w}_N, \mathbf{S}_N)$ is the posterior distribution with
  - $\mathbf{w}_N = S_N\left(\mathbf{S}_0^{-1}\mathbf{w}_0 + \frac{\Phi^T\mathbf{t}}{\sigma^2}\right)$
  - $\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \frac{\Phi^T\Phi}{\sigma^2}$

Since the prior is a conjugate prior, the posterior has the same distribution of the prior. This means that after using a point in the data set to train a model, we can use the output (i.e. the posterior) as the input (i.e., the prior) of a new iteration that uses a different data-set point to further train the model and obtain a new posterior. This iteration can go on indefinitely since the prior is a conjugate prior. Also remember that, we are still using a linear model $y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x})$.

3.4.3 Predictive distribution

In practice, we aren't interested in the distribution of the parameters but in predicting a value t for an input $\mathbf{x}$. This means that we have to compute the probability $p(t|\mathbf{x}, \mathcal{D}, \sigma^2)$, called **posterior predictive distribution**

$$p(t|\mathbf{x}, \mathcal{D}, \sigma^2) = \int \mathcal{N}(t|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}), \sigma^2)\mathcal{N}(\mathbf{w}|\mathbf{w}_N, \mathbf{S}_N)\ d\mathbf{w} \tag{3.60}$$

$$= \mathcal{N}(t|\mathbf{w}_N^T\boldsymbol{\phi}(\mathbf{x}), \sigma_N^2(\mathbf{x})) \tag{3.61}$$

where

$$\sigma_N^2(\mathbf{x}) = \sigma^2 + \boldsymbol{\phi}(\mathbf{x})^T\mathbf{S}_N\boldsymbol{\phi}(\mathbf{x}) \tag{3.62}$$

In the formula for $\sigma_N^2(\mathbf{x})$ we can distinguish two components

- σ^2, which is the *noise in the targets* $\mathbf{t}$.
- $\boldsymbol{\phi}(\mathbf{x})^T\mathbf{S}_N\boldsymbol{\phi}(\mathbf{x})$, which is the *uncertainty associated with the parameter values.*

Note that, when N goes to infinity, the second term goes to 0

$$\boldsymbol{\phi}(\mathbf{x})^T\mathbf{S}_N\boldsymbol{\phi}(\mathbf{x}) \xrightarrow{N\to\infty} 0$$

3.4.4 Modelling

In general, when approaching a Bayesian regression problem, we have to define two important components:

- The **model** (as for now we have used a linear model $\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x})$). The model should be able to describe all possibilities that we think are likely.
- The **prior distribution**. A suitable prior should avoid giving zero or very small probabilities to possible events, but should also avoid spreading out the probability over all possibilities.

To design appropriate models and priors we can model dependencies between parameters for instance, introducing latent variables into the model and hyperparameters into the prior.

3.4.5 Computational complexity

The core of Bayesian regression is computing the posterior distribution. Some approaches for computing it are

- **Analytical integration**. If we use conjugate priors, the posterior distribution can be computed analytically. This approach only works for simple models.
- **Gaussian (Laplace) approximation**. Approximate the posterior distribution with a Gaussian. This approach works well when there a lot of data compared to the model complexity.
- **Monte Carlo integration**. Once we have a sample from the posterior distribution, we can do many things. Currently, the common approach is Markov Chain Monte Carlo (MCMC), that consists in simulating a Markov chain that converges to the posterior distribution.
- **Variational approximation**. A cleverer way of approximating the posterior. It is usually faster than MCMC, but it is less general.

3.4.6 Advantages and disadvantages

The main advantages of Bayesian regression are

- It has a **closed-form solution**.
- It allows to handle **arbitrary non-linearity with the proper basis functions**.

This algorithm also has its drawbacks, in fact

- Basis functions are chosen independently from the training set.
- Curse of dimensionality.

Chapter 4

Linear models for classification

After focusing on continuous problem, we can shift our attention to discrete ones. Basically, in linear classification we have to assign each input $\mathbf{x}_i$ to one of k discrete labels or classes C_k. Usually each input is assigned to one class only. The learning algorithm has to divide the input space in k sub-spaces so that each input is correctly assigned to its class. The boundaries of each regions are called **surface boundaries** or **decision boundaries** or **decision surfaces**.

Notice that, from our definition, each point deterministically belongs to one of the classes, thus there is no noise in the data-set. This isn't however always true, in particular there exist problems in which we might have noise in the labels (i.e. in the classes).

A practical example Let us consider a practical example to understand how classification works. Say we want, given an image, to recognise if it's a cat (class C_0) or a dog (class C_1). Assume that all images are images of dogs or cats (i.e. there is no noise). If we consider a sample i, the input $\mathbf{x}_i$ is the vector of pixels of the image and the target is the class to which the image belongs to (i.e. C_0 or C_1).

4.1 Linear classification

4.1.1 Model

In linear classification we use a linear model that is the same as the one used for linear regression

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D-1} w_i x_i = \mathbf{x}^T \mathbf{w} + w_0$$

Always remember that the model has to be linear in the parameters, thus we can have an arbitrary non linear function ϕ of $\mathbf{x}$ instead of $\mathbf{x}$. The updated model is therefore

$$y(\mathbf{x}, \mathbf{w}) = \boldsymbol{\phi}(\mathbf{x})^T \mathbf{w} + w_0$$

Notice that when we work with the basis function ϕ we want to transform the input space and reach a representation that makes classification easier, i.e. that makes it easier to find boundaries between regions. For instance one might use polar coordinates instead of Cartesian's. Note that, after applying the basis functions, decision boundaries will be linear in the feature space, but would

correspond to nonlinear boundaries in the original input space, since we have used non linear basis functions.

Compressing function

The linear model we have just described returns a continuous result but we need to have discrete labels to assign an input to a class. The first step for obtaining this result is to use a function f of the linear model $\boldsymbol{\phi}(\mathbf{x})^T + w_0$ that can compress the continuous model in a smaller range (i.e. $(0, 1)$ or $(-1, 1)$). The new model is therefore

$$y(\mathbf{x}, \mathbf{w}) = f\big(\boldsymbol{\phi}(\mathbf{x})\mathbf{w} + w_0\big) \tag{4.1}$$

Notice that, f can be **non-linear**, thus y will be non-linear, too. This new model is called **generalised linear model** and the function f is called **generalised linear function**.

Now that the model returns values in a small interval we have to make it discrete. This can be achieved selecting a threshold, i.e. a value that divides the values of a class from the values of another class. For instance if we consider the interval $(0, 1)$ we can define 0.5 as threshold, thus

- All values smaller than 0.5 belong to class C_0.
- All values bigger than 0.5 belong to class C_1.

In a nutshell, **linear classification uses a model obtained from a compressed and discretised continuous linear model** (function f does both compression and discretisation).

Geometric interpretation

Let's recap what we have seen so far; to find a model for linear classification we

1. Map the regression model y_r in a compressed interval (say $(0, 1)$).
2. Put a threshold to the interval $(0, 1)$.

Now we want to understand what is the shape that divides the classes $(0, 0.5]$ and $[0.5, 1)$, i.e. what is the shape of the boundary. To achieve this goal let us equal the model to a constant k (where k is the threshold we have chosen, 0.5 in our example).

$$f(\boldsymbol{\phi}(\mathbf{x})\mathbf{w} + w_0) = k$$

This makes sense because we are investigating the shape of the model in the threshold, that is on the boundary. Now we can apply the inverse of f to both sides of the equation

$$\begin{aligned} f^{-1}\big(f(\boldsymbol{\phi}(\mathbf{x})^T\mathbf{w} + w_0)\big) &= f^{-1}(k) \\ \boldsymbol{\phi}(\mathbf{x})\mathbf{w} + w_0 &= f^{-1}(k) \end{aligned}$$

From the last equation we can notice that the boundary is an hyper-plane and $\boldsymbol{\phi}(\mathbf{x})\mathbf{w} + w_0 = f^{-1}(k)$ is the equation of the plane (i.e. the boundary) that divides the classes. Form this observation we understand that the model of linear classification is linear not because the function is linear but because the plane that divides the classes is.

Multiple outputs

Until now we have considered problems with two classes only. Linear classification is however defined for an arbitrary number of classes k, thus we have to find a way to represent classes in case of k different labels. A good notation to identify a class is the **one-hot** notation. Each target is a vector $\mathbf{t}_i$ with k elements and if the input $\mathbf{x}_i$ belongs to class n we have a 1 in position n (considering 1-indexed vectors) and 0 in all other positions. Notice that the one-hot notation allows to have only one 1. For instance if we have $k = 5$ classes and an input $\mathbf{x}_i$ should be mapped in class 2, then $\mathbf{t}_i$ would be represented as

$$\mathbf{t}_i = (0, 1, 0, 0, 0)^T$$

Notice that the vector $\mathbf{t}_i$ can also be interpreted as a probability vector, in particular we are saying that $\mathbf{x}_i$ belongs to class 2 with probability 1 and to the other classes with probability 0.

4.1.2 Approaches to classification

The linear classification problem can be tackled using three different approaches:

- **Discriminant function**. The discriminant function directly maps each output to a specific class.
- **Probabilistic discriminative approach**. The probabilistic discriminative approach models the probability $p(C_k|\mathbf{x})$ of belonging to a class C_k given the inputs $\mathbf{x}$ using parametric models like logistic regression.
- **Probabilistic generative approach**. The probabilistic generative approach infers $p(C_k|\mathbf{x})$ from the conditional density $p(\mathbf{x}|C_k)$ with the prior probabilities $p(C_k)$ using the Bayes' rule

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

4.2 Thresholds

An important part of classification is finding a threshold to tell if a point belongs to one class or another, thus it's important to analyse the core threshold functions (also called discriminant functions).

4.2.1 Double class

Let's start from the simplest scenario: two classes. If we can compress the model in the region $(-1, 1)$ we can use 0 as threshold and assign

- All points with $f(y(\mathbf{x}, \mathbf{f})) < 0$ to class C_1.
- All points with $f(y(\mathbf{x}, \mathbf{f})) > 0$ to class C_2.

4.2.2 One versus the rest

If we extend the model to K classes, the first solution is to build $K-1$ binary classifiers, one for each class, each of which that tells if an input belongs to a class or not. This method however creates regions that are ambiguously classified, as shown in Figure 4.1.

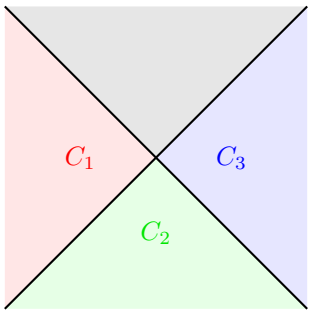


Figure 4.1: An example in which the one versus the rest classification technique creates an ambiguous region. If a sample is not in C_1 nor in C_3, it could either be in C_2 or in the region that doesn't correspond to a class.

4.2.3 One versus one

Another solution for the multi-class model is to create $\frac{K(K-1)}{2}$ classifiers, one for each couple of classes. This means that if we have three classes C_1, C_2 and C_3, we have to build a classifier for C_1 and C_2, one for C_1 and C_3, one for C_2 and C_3. Even in this cases we can have regions that are ambiguously classified, as shown in Figure 4.2

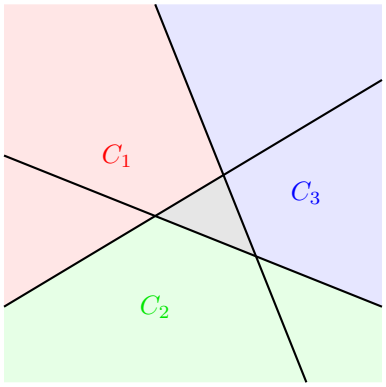


Figure 4.2: An example in which the one versus one classification technique creates an ambiguous region. If a sample is not in C_1 nor in C_3, it could either be in C_2 or in the region that doesn't correspond to a class.

4.2.4 Linear discriminant functions

To classify points in a multi-region problem without having ambiguities, we can use K linear discriminant functions

$$y_k(\mathbf{x}) = \mathbf{x}^T \mathbf{w}_k + w_{k,0} \quad \forall k \in 1, \dots, K$$

and assign a point $\mathbf{x}$ to class C_i only if $y_i(\mathbf{x})$ is bigger than the value of the other functions.

$$\mathbf{x} \mapsto C_i \iff y_i(\mathbf{x}) > y_n(\mathbf{x}) \quad \forall n \neq i$$

This method generates singly connected (i.e. a boundary divides only two classes) and convex decision boundaries, hence for every couple of points $\mathbf{x}_A$, $\mathbf{x}_B$ in region C_i, by definition

$$y_k(\mathbf{x}_A) > y_i(\mathbf{x}_A) \quad \forall i \neq k$$

and

$$y_k(\mathbf{x}_B) > y_i(\mathbf{x}_B) \quad \forall i \neq k$$

and it's true that

$$y_i(\alpha\mathbf{x}_A + (1-\alpha)\mathbf{x}_B) > y_n(\alpha\mathbf{x}_A + (1-\alpha)\mathbf{x}_B) \quad \forall n \neq i, \forall \alpha > 0$$

4.3 Discriminant functions

4.3.1 Least squares

Let us consider a general classification problem with K classes using the one-hot notation to represents targets. Each class C_k is described by its linear model

$$y_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0} \tag{4.2}$$

We can also condense all the equations and write $y(\mathbf{x})$ as

$$y(\mathbf{x}) = \tilde{\mathbf{W}}^T\tilde{\mathbf{x}}$$

where

- $\tilde{\mathbf{W}}$ is a $D \times K$ (or $M \times K$ if we consider basis functions) matrix whose columns represent the different classes. Basically, each column of the matrix is a vector $\tilde{\mathbf{w}}_k = (w_{k0}, \mathbf{w}_k^T)^T = (w_{k,0}, w_{k,1}, \dots, w_{k,D-1})^T$.
- $\tilde{\mathbf{x}} = (1, \mathbf{x}^T)^T = (1, x_1, \dots, x_{D-1})^T$.

If now we consider a data-set $\mathcal{D} = \{\mathbf{x_i}, t_i\}$ with N samples we can apply the same procedure seen for linear regression (compute the derivative of the least squares and nullify it) to obtain the values for the parameters

$$\tilde{\mathbf{W}} = \left(\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\right)^{-1}\tilde{\mathbf{X}}^T\mathbf{T}$$

where

- $\tilde{\mathbf{X}}$ is a $N \times D$ matrix whose i-th row is a sample's input $\tilde{\mathbf{x}}_i^T$.
- $\mathbf{T}$ is a $N \times K$ matrix whose i-th row is a sample's target $\mathbf{t}_i^T$.

Notice that this formula comes from the one obtained for linear regression (i.e. $\mathbf{w} = \left(\Phi^T\Phi\right)^{-1}\Phi^T\mathbf{t}$).

What we have done is adapting a model for linear regression to linear classification. This might seem ok but actually it might lead to many problems. In particular this technique is vulnerable to outliers, in fact if we consider a situation like the one in Figure 4.3 we can notice that the inputs in the lower-right corner contribute in shifting downwards the linear regression model. This shift creates two regions that are not precise and in fact two points are not correctly classified. Long story short, **do not use linear regression for classification problems**.

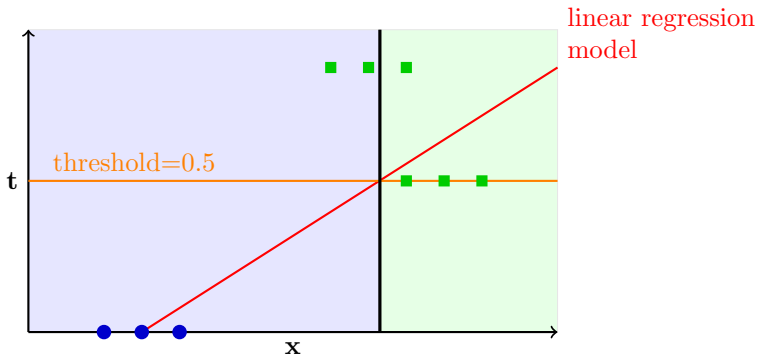


Figure 4.3: Outliers' problem with least squares. The green (squared) points up in the region should belong to class C_1 (i.e., the class of green and squared points), however some of them are in the region of C_2 (i.e., the class of rounded blue points).

4.3.2 Perceptron

The perceptron technique is an **online linear classification algorithm** that allows to classify inputs using a direct approach. In general, this algorithm starts from a boundary and tries, for each misplaced point (usually chosen at random), to fix the boundary until all points in the data-set are well placed (this is why it's defined as direct and online). Now that we have the general idea behind the algorithm we can dig deeper. Let us consider a two-class model compressed in the interval $(-1, 1)$ where

$$y(\mathbf{x}) = f(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x})) = \begin{cases} +1 & \text{if } \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) \geq 0 \\ -1 & \text{if } \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) < 0 \end{cases} \tag{4.3}$$

and the target values are $+1$ for class C_1 and -1 for class C_2.

Loss function

Now that we have a model we have to define a loss function. In particular we are going to use the **perceptron criterion** that assigns

- Zero error to a correctly classified point.
- $\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)t_n$ to a misclassified point $\mathbf{x}_n$. Notice that this penalty is proportional to the distance between the plane and the point.

This means that the loss function that has to be minimised is

$$L_P(\mathbf{w}) = -\sum_{n\in\mathcal{M}} \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)t_n$$

where $\mathcal{M}$ is the set of indices of misclassified points. This function increases when the misclassified points are far from the boundary.

The loss function can be minimised using stochastic gradient descent (i.e. we keep moving $\mathbf{w}$ in the opposite direction of the gradient by a factor α)

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\nabla L_P(\mathbf{w}) = \mathbf{w}^{(t)} + \alpha\boldsymbol{\phi}(\mathbf{x}_n)t_n \tag{4.4}$$

Notice that if we multiply the parameters of the plane for a constant, we are not changing the direction of the plane (i.e. we are not changing the boundary), thus we can fix a value for the learning rate α and still be sure to reach the same plane (i.e. the values of $\mathbf{w}$ will be different but the plane defined by them will have the same direction). For simplicity we will consider $\alpha = 1$.

Algorithm

Now that we know the basis of the perception algorithm we can write it down in pseudocode (Algorithm 1).

Algorithm 1 The gradient descent algorithm for perceptron classification.

Input: $\mathbf{x}_n \in \mathbb{R}^D$
Input: $t_n \in \{-1, 1\}$
Initialise w_0
$k \leftarrow 0$
while convergence **do**
  $k \leftarrow k + 1$
  $n \leftarrow k \mod N$
  if $\hat{t}_n \neq t_n$ **then**
    $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \boldsymbol{\phi}(\mathbf{x}_n)t_n$
  end if
end while

Perceptron convergence theorem

Sometimes, after an iteration, it seems that the algorithm has worsen the model but we have to keep into account that each iteration reduces the loss for a fixed point. In any case, sooner or later, the algorithm will reach convergence and will find a solution if it exists. The problem is that we don't know in how many iterations. To put it in another way the problem is only semi-decidable. This result is stated by the perceptron convergence theorem.

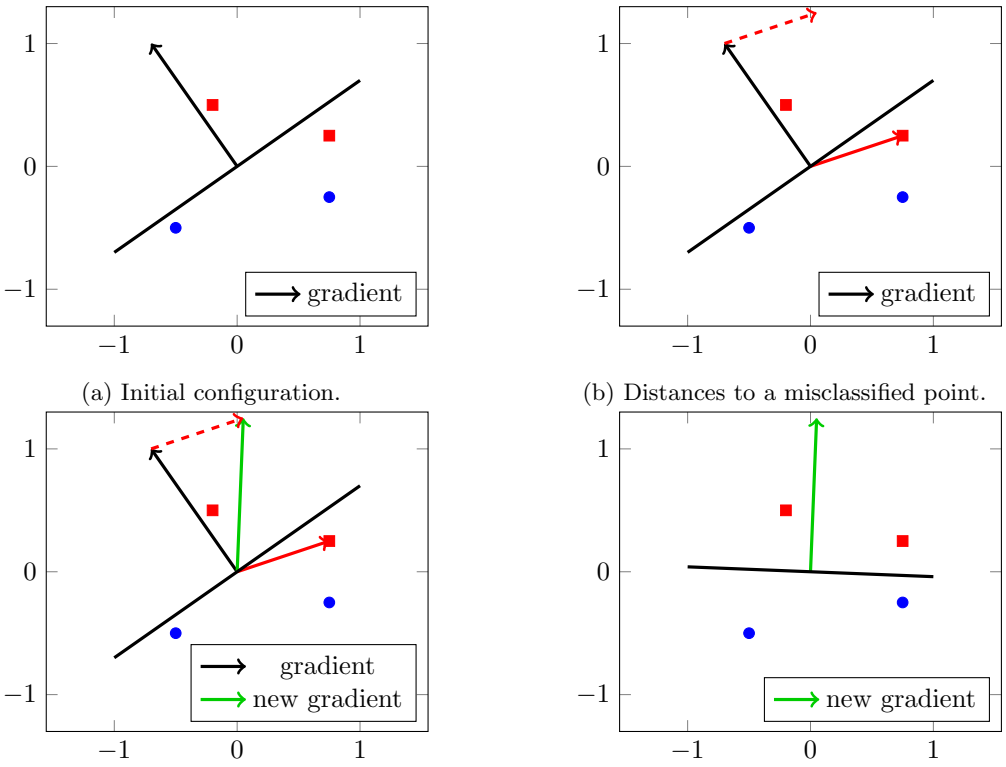
Theorem 4.1 (Perceptron convergence theorem)**.** *If the training data set is linearly separable in the feature space Φ, then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps.*

Notice that the theorem assumes that the training data-set has to be linearly separable. However, since we don't know when the algorithm will stop, we don't know if the algorithm is running because $\mathcal{D}$ is not separable or because convergence is taking a lot of time.

Another important thing to keep in mind is that, if a problem has multiple solutions (i.e. the input space can be partitioned in multiple ways), the solution returned by the algorithm depends on

- The initial values of $\mathbf{w}$.
- The order in which the points are visited.

Points are usually visited at random because evaluating them would be too expensive.



(a) Initial configuration.

(b) Distances to a misclassified point.

(c) Gradient updated.

(d) New boundary computed using the new gradient.

Figure 4.4: One iteration of the perceptron algorithm.

4.4 Probabilistic discriminative models

4.4.1 Logistic regression

Logistic regression uses a logistic sigmoid function (as in Figure 4.5) to model the probability of belonging to a class. To make things simpler we will consider a two-class classification problem with classes C_1 and C_2.

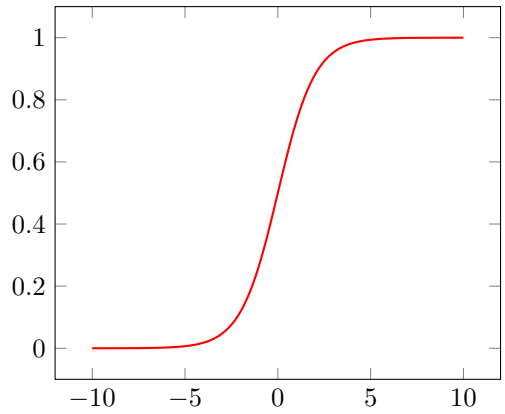


Figure 4.5: A logistic sigmoid function.

Logistic regression uses a discriminative approach, in fact we have to compute the posterior probability $p(C_1|\boldsymbol{\phi})$ of class C_1, which can be written (using the logistic sigmoid function) as

$$p(C_1|\boldsymbol{\phi}) = \frac{1}{1 + e^{-\mathbf{w}^T\boldsymbol{\phi}}} = \sigma(\mathbf{w}^T\boldsymbol{\phi})$$

Notice that in case of two classes we need to build only one classifier because the posterior probability of C_2 can be computed as the complement of the posterior probability of C_1 (remember that we are considering zero noise problems)

$$p(C_2|\boldsymbol{\phi}) = 1 - p(C_1|\boldsymbol{\phi})$$

Maximum likelihood

Let us consider a data-set $\mathcal{D} = \{\mathbf{x}_n, t_n\}$ with N samples and $t_n \in \{0, 1\}$ where 0 is associated to C_2 and 1 to C_1. To learn $\mathbf{w}$ we have to maximise the likelihood, i.e., the probability of getting the right label (i.e. the right class)

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}(1 - y_n)^{1-t_n} \tag{4.5}$$

where y_n is the sigmoid logistic function

$$y_n = \sigma(\mathbf{w}^T\boldsymbol{\phi}_n)$$

Let us analyse a litter deeper the likelihood. As we can see, if the target of a sample is 0 then the class should be C_2, thus we need $p(C_2|\boldsymbol{\phi}) = 1 - p(C_1|\boldsymbol{\phi})$. This formula correctly models this

property, in fact the term $y_n^{t_n}$ is 1 for $t_n = 0$ and only $1 - y_n$ survives. On the other hand, if the target is 1 we should consider $p(C_1|\boldsymbol{\phi})$, and so it is because $1 - y_n$, being elevated to $1 - t = 0$ becomes 1. Basically we are selecting the correct probability in a `if-else` fashion.

As we have done for linear regression we can consider the logarithmic likelihood to make things easier (remember we need only the position of the optimum, i.e., $\mathbf{w}$ and not it's value).

$$L(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}) \tag{4.6}$$

$$= -\sum_{n=1}^{N} \Big(t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n) \Big) \tag{4.7}$$

$$= \sum_{n=1}^{N} L_n \tag{4.8}$$

Now that we finally have the likelihood function we have to compute the gradient. To simplify this computation, we can start computing the gradient for a point

$$L_n = -\big(t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n) \big)$$

and then sum the results (thanks to the linearity of the derivative). The gradient in one point can be computed using the chain rule

$$\frac{\partial L_n}{\partial \mathbf{w}} = \frac{\partial L_n}{\partial y_n} \frac{\partial y_n}{\partial \mathbf{w}}$$

Let's start from the first term.

$$\begin{aligned}
\frac{\partial L_n}{\partial y_n} &= -\frac{\partial}{\partial y_n} \Big[t_n \ln(y_n) \Big] - \frac{\partial}{\partial y_n} \Big[(1 - t_n) \ln(1 - y_n) \Big] \\
&= -\frac{t_n}{y_n} - \frac{1 - t_n}{1 - y_n} \cdot -1 \\
&= \frac{-t_n(1 - y_n) + y_n(1 - t_n)}{y_n(1 - y_n)} \\
&= \frac{-t_n + t_n y_n + y_n - y_n t_n}{y_n(1 - y_n)} \\
&= \frac{y_n - t_n}{y_n(1 - y_n)}
\end{aligned}$$

Now we can move to the second derivative.

$$\begin{aligned}
\frac{\partial y_n}{\partial \mathbf{w}_n} &= \frac{\partial}{\partial \mathbf{w}_n}\left[\left(1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}\right)^{-1}\right] \\
&= -\left(1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}\right)^{-2}\cdot\frac{\partial}{\partial \mathbf{w}_n}\left[1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}\right] \\
&= -\left(1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}\right)^{-2}\cdot -\boldsymbol{\phi}_n e^{-\mathbf{w}^T\boldsymbol{\phi_n}} \\
&= \left(1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}\right)^{-2}\boldsymbol{\phi}_n e^{-\mathbf{w}^T\boldsymbol{\phi_n}} \\
&= \boldsymbol{\phi}_n\cdot\frac{1}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}\cdot\frac{e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}} \\
&= \boldsymbol{\phi}_n\cdot\frac{1}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}\cdot\frac{1-1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}} \\
&= \boldsymbol{\phi}_n\cdot\frac{1}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}\cdot\frac{1-1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}} \\
&= \boldsymbol{\phi}_n\cdot\frac{1}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}\cdot\left(1-\frac{1}{1+e^{-\mathbf{w}^T\boldsymbol{\phi_n}}}\right) \\
&= \boldsymbol{\phi}_n\cdot y_n\cdot(1-y_n)
\end{aligned}$$

Putting the two components together we obtain

$$\begin{aligned}
\frac{\partial L_n}{\partial \mathbf{w}} &= \frac{\partial L_n}{\partial y_n}\frac{\partial y_n}{\partial \mathbf{w}} \\
&= \frac{y_n-t_n}{y_n(1-y_n)}\cdot y_n(1-y_n)\boldsymbol{\phi}_n \\
&= (y_n-t_n)\boldsymbol{\phi}_n
\end{aligned}$$

Having obtained the gradient for a point, we can compute the gradient as the sum of the gradients of every point

$$\nabla L(\mathbf{w}) = \sum_{n=1}^{N}(y_n-t_n)\boldsymbol{\phi}_n \tag{4.9}$$

The gradient can be used for gradient descend but we can't obtain the closed form solution because the logistic sigmoid function (∇ depends on y_n) is non linear. Also notice that the gradient descend algorithm leads to a global solution, thus we don't have local maximum problems (which is good).

Multiclass logistic regression

Now that we know how logistic regression works on two classes we can try and extend it to multi-class problems. In particular, we can start from rewriting the posterior probability using the softmax transformation of linear functions of feature variables

$$p(C_k|\boldsymbol{\phi}) = y_k(\boldsymbol{\phi}) = \frac{e^{\mathbf{w}_k^T\boldsymbol{\phi}}}{\sum_j e^{\mathbf{w}_j^T\boldsymbol{\phi}}} \tag{4.10}$$

Now we can write the likelihood as

$$p(\mathbf{T}|\boldsymbol{\phi}, \mathbf{w}_1, \dots, \mathbf{w}_K) = \prod_{n=1}^{N} \left(\prod_{n=1}^{K} p\big(C_k|\boldsymbol{\phi}_n\big)^{t_{n,k}} \right)$$
$$= \prod_{n=1}^{N} \left(\prod_{k=1}^{K} y_{n,k}^{t_{n,k}} \right)$$

A always we can consider the logarithmic likelihood and write

$$L(\mathbf{w}_1, \dots, \mathbf{w}_K) = -\ln p(\mathbf{T}|\Phi, \mathbf{w}_1, \dots, \mathbf{w}_K)$$
$$= -\sum_{n=1}^{N} \left(\sum_{k=1}^{K} t_{n,k} \ln(y_{n,k}) \right)$$

Finally we can compute the gradient that can be used for the gradient descend algorithm

$$\nabla L_{\mathbf{w}_j}(\mathbf{w}_1, \dots, \mathbf{w}_K) = \sum_{n=1}^{N} (y_{n,j} - t_{n,j})\boldsymbol{\phi}_n$$

4.4.2 Comparison between the perceptron and logistic regression

The perceptron algorithm and logical regression, even if they use different approaches, have a similar behaviour. In particular

- The perceptron algorithm uses a step function to classify the inputs. This function is much more steep and precisely separate the inputs. In other words the probability that a point is in a class is either 0 or 1. Formally, the perceptron algorithm uses a model

$$y(\mathbf{x}, \mathbf{w}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Logistic regression uses a much smooth function to classy inputs. This leads to some misclassification. This isn't however a big issue because even for misclassified points we know the value of the likelihood function. Formally, logistic regression uses a model

$$y(\mathbf{x}, \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}\boldsymbol{\phi}(\mathbf{x})}}$$

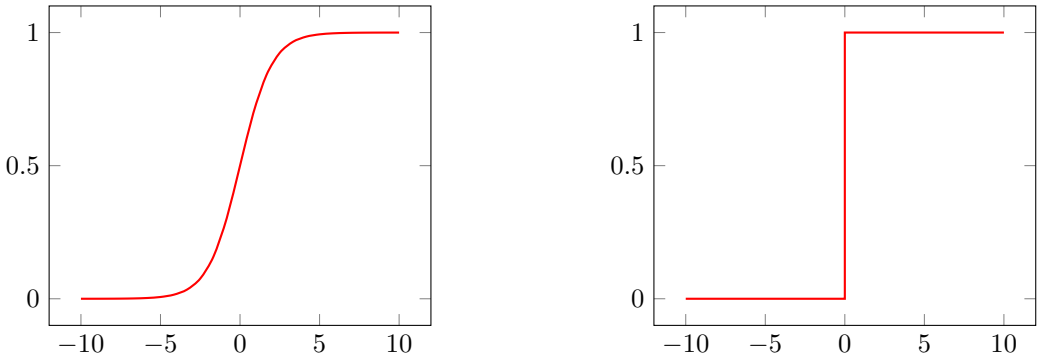
Furthermore both algorithms use the same rule to update the value of $\mathbf{w}$.

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\big(y(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\big)\boldsymbol{\phi}_n$$

4.5 Naive Bayes

The Naive Bayes classification algorithm uses the following model

$$y(\mathbf{x}_n) = \arg\max_k p(C_k) \prod_{j=1}^{M} p(x_j|C_k) \tag{4.11}$$



(a) A logistic curve.

(b) A perceptron.

Figure 4.6: The comparison between a logistic and a perceptron curve.

The error is computed using the the logarithmic likelihood, as in Equation 3.34, and it is minimised using the **Maximum Likelihood Estimation** (MLE) algorithm.

Let us now understand where does the model come from. From the name of this technique we understand that we are using a Bayesian approach. In particular, we want to compute the probability of C_k given an input $\mathbf{x}$. Directly applying the Bayes' rule we can write

$$p(C_k|\mathbf{x}) = \frac{p(C_k)p(\mathbf{x}|C_k)}{p(\mathbf{x})} = \frac{p(\mathbf{x}, C_k)}{p(\mathbf{x})} \tag{4.12}$$

This means that, the posterior $p(C_k|\mathbf{x})$ is proportional to $p(\mathbf{x}, C_k) = p(x_1, \dots, x_M, C_k)$, hence we can write

$$p(C_k|\mathbf{x}) = p(x_1, \dots, x_M, C_k) \tag{4.13}$$

Now we can apply the Bayes' rule

$$p(x_1|x_2, \dots, x_M, C_k) = \frac{p(x_1, x_2, \dots, x_M, C_k)}{p(x_2, \dots, x_M, C_k)}$$

to obtain

$$p(C_k|\mathbf{x}) = p(x_1|x_2, \dots, x_M, C_k)p(x_2, \dots, x_M, C_k) \tag{4.14}$$

Now we can keep applying the Bayes' rule (A.3 for the conditional probability, A.1 for the joint probability) to expand all the terms and obtain

$$p(C_k|\mathbf{x}) = p(C_k) \prod_{j=1}^{M} p(x_j|C_k) \tag{4.15}$$

Now, to decide to which class we have to assign $\mathbf{x}$ we have to take the k that maximises the probability of C_k given k, hence we get the model

$$\begin{aligned} y(\mathbf{x}_n) &= \arg\max_k p(C_k|\mathbf{x}) \\ &= \arg\max_k p(C_k) \prod_{j=1}^{M} p(x_j|C_k) \end{aligned}$$

Now that we know where does the model come from, we can define the prior $p(C_k)$ and the likelihood $p(x_j|C_k)$ we want to use. In particular, one option is to use

- A multinomial distribution with parameters $(p_1, \dots, p_k)$ for the prior $p(C_k)$.
- A Gaussian distribution with parameters μ_{jk} and σ^2_{jk} for each feature x_j and class C_k

This means that, practically we can compute prior and likelihood as follows

- The prior can be estimated as

$$\hat{p}(C_k) = \frac{\sum_{i=1}^{N} I\{\mathbf{x}_n \in C_k\}}{N}$$

- The likelihood can be approximated estimating the parameters of the normal distribution for each couple of feature x_i and class C_k.

$$p(\mathbf{x}|C_k) = \mathcal{N}(x_j|\hat{\mu}_{j,k}, \sigma^2_{j,k})$$

Obviously this isn't the only choice for priors and likelihood, and we should chose the appropriate distribution depending on the problem and the data-set. Note that the naive Bayes algorithm is **parametric** and **generative**.

4.6 K-nearest-neighbours

K-nearest-neighbour (KNN) is a non-parametric, discriminative algorithm. Being a non-parametric algorithm, KNN, doesn't need training and we can directly use the samples in the data-set to predict the class for a new input $\mathbf{x}$. More precisely, to predict the class for a new point $\mathbf{x}$ we have to:

1. Take the K nearest point to $\mathbf{x}$ using a distance metric (e.g., the Euclidean distance). Let us call $\mathcal{N}$ the set with the K nearest points.
2. Assign a class to $\mathbf{x}$ considering the points drawn at point 1 (i.e., $\mathcal{N}$), eventually giving more importance to some points using weights. A way to choose the class could be using majority voting. Namely, $\mathbf{x}$ is assigned to the class to which the majority of points in $\mathcal{N}$ belong.

When using this algorithm, we have to decide:

- How to compute the **distance** between points. An example is the Euclidean distance.
- **How many neighbours** to consider, namely, the value of K.
- A **weight function**, to compute on each point. An example could be giving more weight to the points which are closer to $\mathbf{x}$.
- How to **fit with local points**. An example is majority voting.

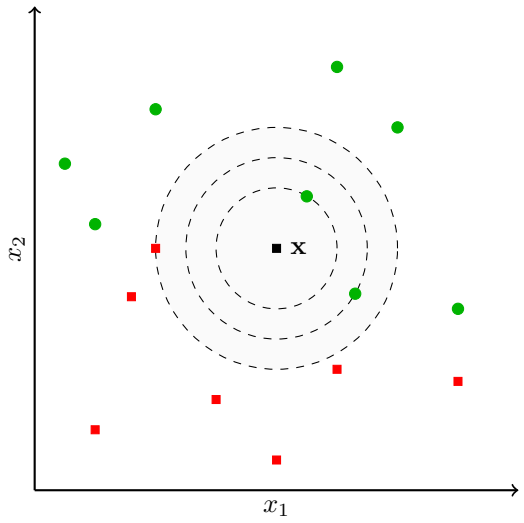


Figure 4.7: An example of classification of a point **x** using 3-nearest-neighbours using equal weights and majority voting.

4.7 Result evaluation

After classifying some points in the data set we can compute how well the model has classified them. Before analysing these metrics, we should introduce some concepts:

- True positives tp. The number of points belonging to class C_1 correctly classified in class C_1 by the model.
- True negatives tn. The number of points belonging to class C_0 correctly classified in class C_0 by the model.
- False positives fp. The number of points belonging to class C_0 wrongly classified in class C_1 by the model.
- False negatives fn. The number of points belonging to class C_1 wrongly classified in class C_0 by the model.

Now we are ready to introduce the main metrics that can be used to evaluate a model:

- **Accuracy**. The accuracy is the sum of correctly classified points over the total number of points.
$$Acc = \frac{tp + tn}{N}$$
- **Precision**. The precision is ratio between the points correctly classified in class C_1 and all the points classified in class C_1.
$$Pre = \frac{tp}{tp + fp}$$

- **Recall**. The recall is ratio between the points correctly classified in class C_1 and all the points that should have been classified in class C_1.

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

- **F1 score**.

$$F1 = \frac{2 \cdot Pre \cdot Rec}{Pre + Rec}$$

Chapter 5

Theoretical concepts

5.1 No free lunch theorems

An important result of machine learning is a set of theorems called no free lunch theorems. Before analysing them we have to define the concept of accuracy,

Definition 5.1 (Accuracy)**.** *The accuracy $ACC_G(L)$ of a learner L is the generalisation capability of L, that is the accuracy of L on unseen examples.*

In other words the accuracy measures how good a learner is when we use a data-set completely different from the one used for training.

Now that we have defined accuracy we can state one of the no free lunch theorems

Theorem 5.1 (No free lunch)**.** *For any learner L, the average accuracy over all possible functions $\mathcal{F}$ is 0.5*

$$\frac{1}{|\mathcal{F}|}\sum_{\mathcal{F}} ACC_G(L) = \frac{1}{2} \tag{5.1}$$

In the theorem just stated, $\mathcal{F}$ is the set of all possible functions $y = f(\mathbf{x})$ that the learner can learn (e.g., the functions that, for a classification problem, allow to split the input space in two regions). A function $y = f(x)$ is also called **concept**.

Theorem 5.1 provides a very strong result, in fact it states that every learner has the same performance as a random guesser. In other words, any learner is not better than a random guesser.

Looking at this result, we might think that it's completely worthless studying machine learning if the best we can do is guess at random. We will however be wrong because the no free lunch theorem averages on all possible functions f, i.e., on everything that can happen. In fact, if everything can happen then nothing can be learned because when something happens, the next thing isn't correlated to the one happened before. In other words, an event doesn't reveal any information. This is however rather wrong in the real world in fact when an event happens, it's not usually at random, hence we can learn from it. This means that in real world problems we can compress the function space $\mathcal{F}$ and use regularities to learn and generalise. Another important consequence is that we should consider different approaches to solve a problem because a learner might be more

accurate if used for some problems.

The no free lunch also has a corollary that states

Theorem 5.2 (No free lunch corollary)**.** *For any two learners L_1 and L_2, if there exists a problem P so that the accuracy of L_1 is strictly bigger then the one of L_2, then there must exist a different problem P' for which the accuracy of L_2 is bigger than the one of L_1*

$$\exists P : ACC_G(L_1) > ACC_G(L_2) \Rightarrow \exists P' \neq P : ACC_G(L_2) > ACC_G(L_1) \tag{5.2}$$

5.2 Bias-variance trade-off

One of the most important concepts in machine learning is the trade-off between bias and variance. To study this problem let us consider a data set $\mathcal{D}$ with N samples obtained by a deterministic function f over the inputs $\mathbf{x_i}$ and the some noise ε

$$t_i = f(\mathbf{x}_i) + \varepsilon \quad \forall i \in N \tag{5.3}$$

Let us also assume that the noise has 0 mean and variance σ^2. The goal of machine learning is to find a model $y(\mathbf{x})$ that approximates well f. As we have seen before we can measure how well y approximates f, in practical terms, using the square error (i.e. $(t - y(\mathbf{x}))^2$). Now we can take an input $\mathbf{x}$ and compute the expected value of the square error.

$$\begin{aligned}
\mathbb{E}\big[(t - y(\mathbf{x}))^2\big] &= \mathbb{E}\big[t^2 + y(\mathbf{x})^2 - 2ty(\mathbf{x})\big] && (5.4)\\
&= \mathbb{E}\big[t^2\big] + \mathbb{E}\big[y(\mathbf{x})^2\big] - 2t\mathbb{E}\big[y(\mathbf{x})\big] && (5.5)\\
&= \mathbb{E}\big[t^2\big] + \mathbb{E}\big[t\big]^2 - \mathbb{E}\big[t\big]^2 + \mathbb{E}\big[y(\mathbf{x})^2\big] + \mathbb{E}\big[y(\mathbf{x})\big]^2 - \mathbb{E}\big[y(\mathbf{x})\big]^2 - 2f(\mathbf{x})\mathbb{E}\big[y(\mathbf{x})\big] && (5.6)\\
&= Var\big[t\big] + \mathbb{E}\big[t\big]^2 + Var\big[y(\mathbf{x})\big] + \mathbb{E}\big[y(\mathbf{x})\big]^2 - 2f(\mathbf{x})\mathbb{E}\big[y(\mathbf{x})\big] && (5.7)\\
&= Var\big[t\big] + Var\big[y(\mathbf{x})^2\big] + \big(f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})]\big)^2 && (5.8)\\
&= Var\big[t\big] + Var\big[y(\mathbf{x})^2\big] + \big(\mathbb{E}[f(\mathbf{x})] - \mathbb{E}[y(\mathbf{x})]\big)^2 && (5.9)\\
&= Var\big[t\big] + Var\big[y(\mathbf{x}) + \mathbb{E}\big[f(\mathbf{x}) - y(\mathbf{x})\big]^2 && (5.10)
\end{aligned}$$

where in

- 5.6 we have replaced $t = f(\mathbf{x}) + \varepsilon$ but, since the noise is not related to the model, only $2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})]$ survives.
- 5.7 we have used the fact that $Var[A] = \mathbb{E}[A^2] - \mathbb{E}[A]^2$.
- 5.9 we have exploited the fact that $\mathbb{E}[f(\mathbf{x})] = f(\mathbf{x})$, since f is deterministic.

Before going on, let us analyse a little bit deeper the variance of t, in particular we can write

$$\begin{aligned}
Var[t] &= Var[(t - \mathbb{E}[t])^2] && (5.11)\\
&= Var[(t - \mathbb{E}[f(\mathbf{x}) + \varepsilon])^2] && (5.12)\\
&= Var[(t - \mathbb{E}[f(\mathbf{x})] + \mathbb{E}[\varepsilon])^2] && (5.13)\\
&= Var[(t - f(\mathbf{x}))^2] && (5.14)\\
&= Var[\varepsilon^2] && (5.15)\\
&= \sigma^2 && (5.16)
\end{aligned}$$

Long story short, we obtain

$$\mathbb{E}\big[(t - y(\mathbf{x}))^2\big] = Var\big[t\big] + Var\big[y(\mathbf{x})\big] + \mathbb{E}\big[f(\mathbf{x}) - y(\mathbf{x})\big]^2 \tag{5.17}$$

In Equation 5.17 we can recognise three different components:

- $Var\big[t\big] = \sigma^2$ which is the **irreducible error** given by the noise. Since we can't act on this component, we can forget about it.
- $Var\big[y(\mathbf{x})\big]$ which is the **variance** of point $\mathbf{x}$ computed on all the possible models y.
- $\mathbb{E}\big[f(\mathbf{x}) - y(\mathbf{x})\big]^2$ which is the squared **bias**. From this expression we can notice that the bias is the expected value of the difference between the function f we are trying to approximate and our approximation y.

To sum things up, we can write the expected value of the squared error as

$$\mathbb{E}\big[(t - y(\mathbf{x}))^2\big] = \sigma^2 + \text{variance} + \text{bias}^2 \tag{5.18}$$

Variance and bias are fundamental in machine learning and have to be minimised, however we should always keep in mind that they should be minimised together, as a sum, i.e., as $Var\big[y(\mathbf{x})\big] + \mathbb{E}\big[f(\mathbf{x}) - y(\mathbf{x})\big]^2$. The ideal situation would be having low bias and variance but this is rarely the case, in fact we usually have models that have:

- **Low variance and high bias**. In this situation the hypothesis space is too small but the model is stable. In other words the model is too simple (i.e. approximates too much).
- **Low bias and high variance**. In this situation every time we train the model we get different results (i.e. the model is unstable).

The graph in Figure 5.1 shown how variance and bias should be minimised as a sum.

5.2.1 Bias

Let's start with the analysis of bias.

Definition 5.2 (Bias)**.** *The bias is the expected value of the squared difference between the model $y(\mathbf{x})$ the algorithm is learning and the function $f(\mathbf{x})$ we are approximating.*

$$Bias^2 = \mathbb{E}\big[f(\mathbf{x}) - y(\mathbf{x})\big]^2 \tag{5.19}$$

$$= \int \big(f(\mathbf{x}) - E\big[y(\mathbf{x})\big]\big)^2 p(\mathbf{x})\ d\mathbf{x} \tag{5.20}$$

Notice that the bias can't be practically computed because we don't know the probability distribution of the input $\mathbf{x}$, thus it's just a theoretical quantity that can only be estimated.

How to reduce bias Bias can be reduced increasing the hypothesis space $\mathcal{F}$, hence increasing the complexity of the model. The reason being that the more complex the model, the more expressive it is.

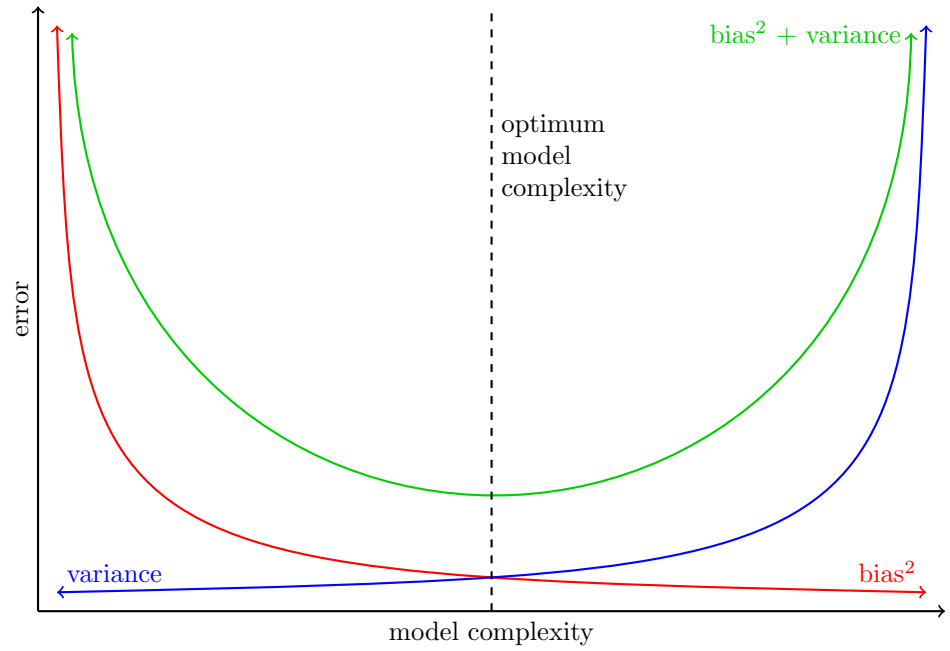


Figure 5.1: A graphical representation of bias and variance.

5.2.2 Variance

Let us now analyse variance.

Definition 5.3 (Variance)**.** *The variance is the difference between what we learn from a particular data-set and what we expect to learn*

$$\textit{Variance} = \int \mathbb{E}\big[(y(\mathbf{x}) - \bar{y}(\mathbf{x}))^2\big] p(\mathbf{x})\ d\mathbf{x} \tag{5.21}$$

where

$$\bar{y}(\mathbf{x}) = \mathbb{E}[y(\mathbf{x})]$$

As for bias, variance can only be estimated and therefore it's only a theoretical quantity.

How to reduce variance Variance can be reduced

- Using **simpler models** (i.e. models with less features).
- **Increasing the number of samples**. An increase in samples allows us to better estimate the loss function and create a model that follows less the training set and more the actual loss function.

The relationship between bias and variance can be graphically schematised as in Figure 5.2.

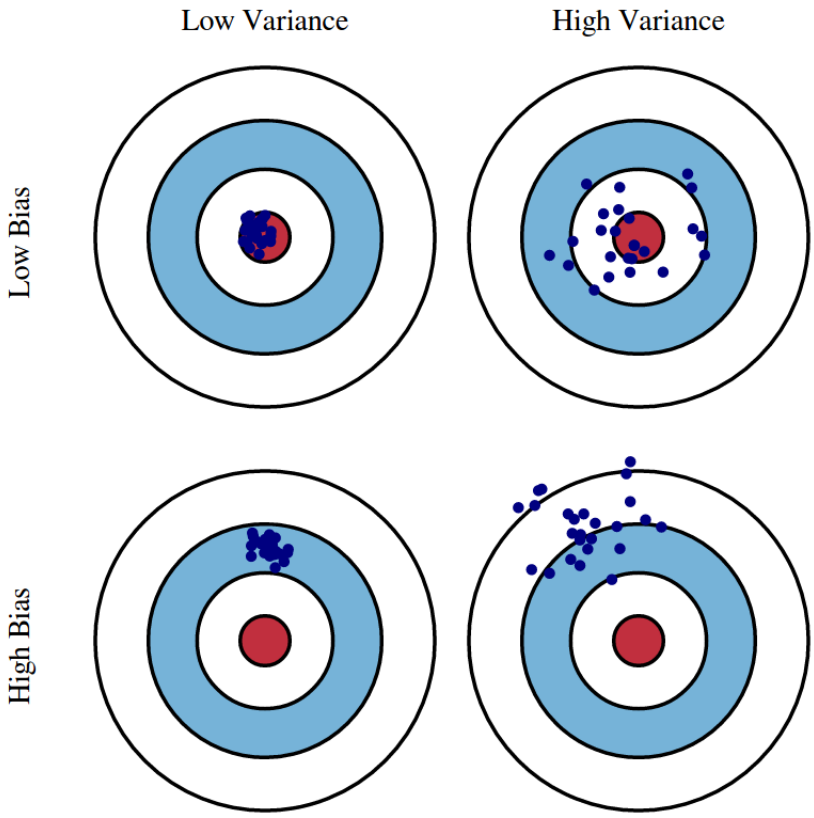


Figure 5.2: The relationship between bias and variance represented as points on a target. The centre of the target is the target model f and the points are different models $y(\mathbf{x})$ trained with different training sets $\mathcal{D}$. How close we can get to the bullseye represents bias (the lowest the bias, the closer we are to the true model). The dispersion around the bias represents the variance. The lower the variance, the closer points (i.e., different models) are one to the other. This means that the model doesn't depend on the data set.

	Bias	Variance
Decreases with	more complex models	less complex models larger data sets
Increases with	less complex models	more complex models

Table 5.1: How to increase and decrease bias and variance.

Chapter 6

Testing and validation

When approaching a machine learning problem we have to:

1. Select a model. This phase involves model **validation** which allows to evaluate different models (e.g., models with different number of features) and select the best one.
2. **Train** the model we selected.
3. **Test** the performance of the model after being trained.

In this chapter we will analyse the last two steps of this process, namely, how to validate and test a model. In particular, we will start from testing and then we will move to validation.

In some cases testing might highlight that the model we selected is performing badly. In such cases, trying to repeat the process of training and validating new models might be wrong. In fact, after some iterations, we might get a good model but it will be just because of the noise in the data an not because the model correctly approximates f. Usually the best way to deal with this issue is to get better training data.

6.1 Measuring error and evaluating models

6.1.1 Training error

The training error allows to determine the approximation error of model $y(\mathbf{x})$ with respect to the training set $\mathcal{D}$. The training error is computed

- For **regression** as
$$L_{train} = \frac{1}{N}\sum_{N=1}^{N}\big(t_n - y(\mathbf{x}_n)\big)^2$$
- For **classification** as
$$L_{train} = \frac{1}{N}\sum_{N=1}^{N} I\big(t_n \neq y(\mathbf{x}_n)\big)$$
where $I\big(t_n \neq y(\mathbf{x}_n)\big)$ is 0 if the model predicts the class correctly (i.e. if $t_n = y(\mathbf{x}_n)$), 1 otherwise.

where N is the number of samples of the data set $\mathcal{D}$. In practice the training error, for every sample $(\mathbf{x}_i, t_i)$, considers the difference between the output of the model $y(\mathbf{x}_i)$ and the true target t_i.

We can immediately notice that the train error only considers how close a model is to the training set, thus it has the same trend of the bias, but it doesn't considers variance. In other words, when the model fits perfectly the training set, i.e. it has a low bias, we obtain a really small training error, but the model has a very high variance because it fits only the data set and not the function f we are approximating. In other words, the the training error decreases with the complexity of the model, hence it will always choose a complex model over a simple one. This means that the train error isn't a good way to evaluate the bias-variance trade-off.

6.1.2 Prediction error

The main issue with the training error is that it uses only the points in the data set. This means that, the training error isn't enough to evaluate the bias-variance trade-off, and we should consider the error on all input points (not only those of the data set). This means that, the error should be computed

- For **regression** as
$$L_{true} = \int \big(t_n - y(\mathbf{x}_n)\big)^2 p(\mathbf{x})\ d\mathbf{x}$$
- For **classification** as
$$L_{true} = \int I\big(t_n \neq y(\mathbf{x}_n)\big) p(\mathbf{x})\ d\mathbf{x}$$

This new error is called **prediction error** and can't be practically computed because it's an integral on all points and needs the probability of a point $\mathbf{x}$ to be in the data set, which is unknown for some points.

Notice that the training error is an optimistically biased estimate of the prediction error because it considers only the points that are in the data set (i.e. it's biased on those points).

Also notice that, differently from the training error, the prediction error considers both bias and variance because we are considering all points. Long story short, the prediction error is a good theoretical evaluator for the bias-variance trade-off, which can't be used in practice.

6.1.3 Test error

The prediction error can't be precisely computed, but we can try to estimate it. In particular we can sacrifice some samples of the training set $\mathcal{D}$ and use them to test how good the model is (i.e. how good we have trained the model). This means that the extracted samples aren't used for training, instead we predict the output $\hat{t}_a$ of a test sample $\mathbf{x}_a$ and the we compare it with t_a (i.e. with the correct output). Algorithmically we

1. Divide the data set D in train set Tr and test set Te.
2. Use the training set Tr to optimise the model (i.e. to optimise the parameters of the model).
3. Use the test set Te to evaluate the prediction error.

The prediction error can be estimated

- For **regression** as

$$L_{test} = \frac{1}{N_{test}} \sum_{N=1}^{N_{test}} \big(t_n - y(\mathbf{x}_n)\big)^2$$

  where N_{test} is the cardinality of the test set.

- For **classification** as

$$L_{test} = \frac{1}{N_{test}} \sum_{N=1}^{N_{test}} I\big(t_n \neq y(\mathbf{x}_n)\big)$$

An important thing to keep in mind is that the test set should never be used to train the model, otherwise we would obtain a biased prediction error (i.e. something closer to the training error). Intuitively, if we test the model using the same samples used to train it, we are still evaluating how close the model is to the training set. Basically, in this case we are saying that the model is better than it actually is.

Set dimension

Another important problem related to the prediction error's estimate is how the data set is divided, in particular

- If the train set is too small we can't optimise the model and the parameters that much, thus obtaining a model with a very high bias.
- If the test set is too small we can't estimate the prediction error precisely.

If we plot the training, prediction (obtained thanks to the test error) and estimate prediction error with respect to the number of samples we notice that both the prediction and train error converge to a certain value when the number of data points is high. The distance between the two errors is a measure of the model's variance. An example of this property can be seen in Figure 6.1.

The same plot (i.e. the errors with respect to the number of samples) can be used to detect high bias and variance. In particular we notice high bias if the errors converge too fast. An example is shown in Figure 6.2. On the other hand if the variance is too high, the errors can't converge to the same value, thus the gap is always quite big. An example is shown in Figure 6.3.

6.2 Validation

Until now, we have said that we should use the test error (i.e. an estimate of the prediction error) to evaluate the goodness of a model. However, using the residual sum of squares (RSS) isn't a good choice when we want to compare different models that have a different number of features because the RSS doesn't considers model complexity (we use the prediction $y(\mathbf{x})$ of the model without considering the number of features). This means that we have to find other ways to estimate the test error, in particular we can

- Estimate the test error directly, using a **validation** approach.
- Make an **adjustment** to the training error to account for model complexity.

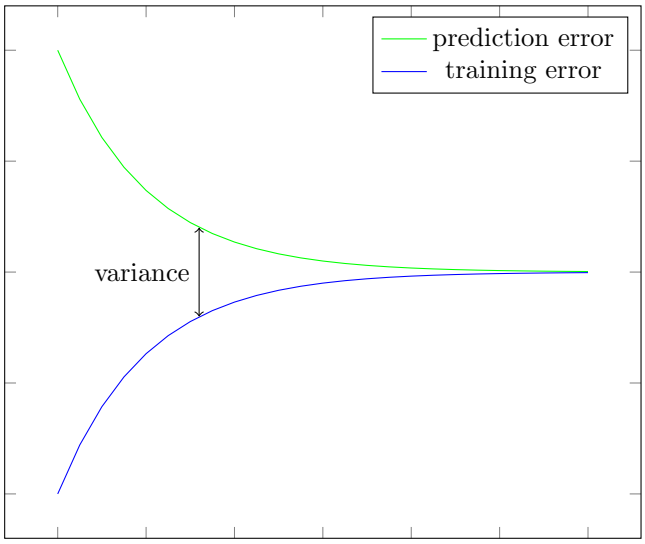


Figure 6.1: Prediction and training error with respect to the number of samples.

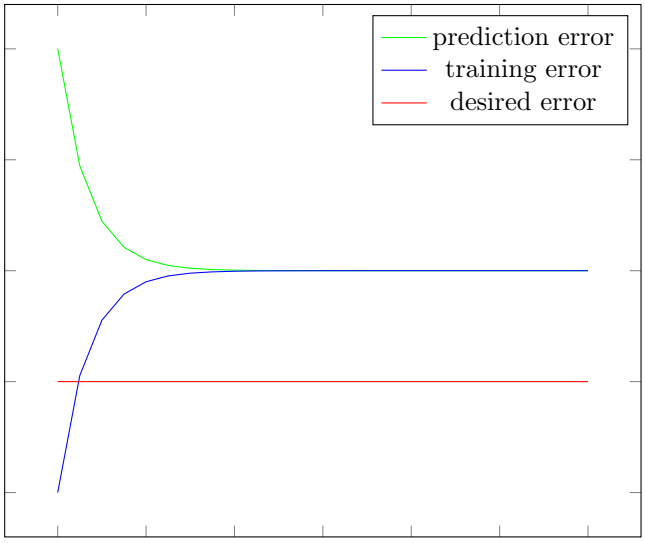


Figure 6.2: High bias on the error plot.

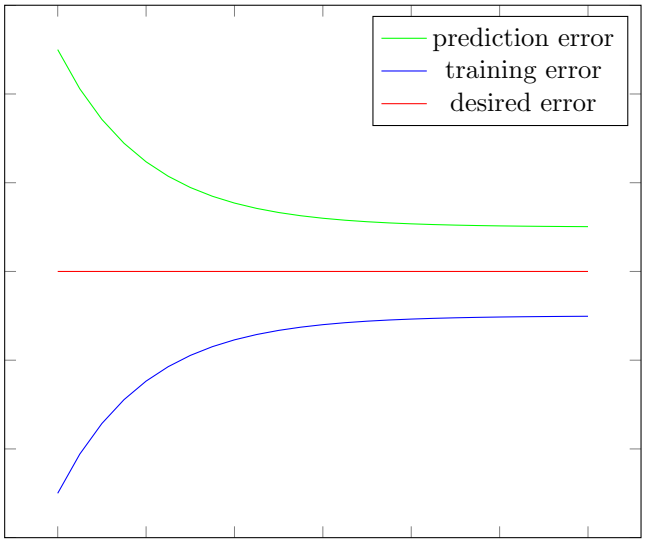


Figure 6.3: High variance on the error plot.

6.2.1 Adjustment techniques

The prediction error can also be estimated using the train error. Basically, what we are trying to do is to adjust the train error adding a correction term that penalises models with a lot of parameters (i.e., complex models). The correction error gets bigger with more complex models so that the train error also increases with the complexity of the model. Some adjustment techniques are

- C_p.

$$C_p = \frac{1}{N}\left(RSS + 2d\sigma^2\right)$$

  where

  - d is the number of parameters.
  - σ^2 is an estimate of the variance of the noise.

  Notice that in this case the correction term is $2d\sigma^2$. Also notice that if we increase the number of dimensions (i.e. the number of features), the prediction error becomes bigger because of the correction error.

- **AIC**.

$$AIC = -2\log L + 2d$$

  where

  - L is the maximised value of the likelihood function for the estimated model.
  - d is the number of parameters.

  In this case the correction term is $2d$.

- **BIC**.
$$BIC = \frac{1}{N}\big(RSS + \log(N)d\sigma^2\big)$$
In this case the correction term is $\log(N)d\sigma^2$. Notice that BIC replaces C_p's correction term with $\log(N)d\sigma^2$. In BIC we multiply $d\sigma^2$ by a logarithm while in C_p by 2 and since $\log N > 2$, then BIC selects smaller models.

- **Adjusted** R^2.
$$AdjustedR^2 = 1 - \frac{RSS \cdot TSS}{(N-d-1)(N-1)}$$
where TSS is the total sum of squares. In this case a large $AdjustedR^2$ is sign of a model with small test error.

6.2.2 Validation

The validation approach requires to obtain another subset from the data set. This means that we should partition the data set in

- A **training set** to train the model.
- A **validation set** to evaluate different models with different number of features (e.g., after applying model selection techniques, which will be analysed later on).
- A **test set** to evaluate the goodness of the model selected. This set has to be preserved to the very last phase in which we are sure about the model. In any case this set should be used for any of the two previous phases.

We have understood that we need validation to choose the best model with the best number of features, now we have to describe what techniques can be used to achieve this goal. In particular, we will analyse

- **Leave-One-Out cross validation** (LOO cross validation).
- **k-fold cross validation**.

Both techniques are cross validation techniques, hence they use mixed validation and training sets to evaluate different models.

Leave-One-Out cross validation

Before introducing this technique, let us call $\mathcal{A}$ the set that contain the data set minus the test set. The LOO cross validation techniques partitions $\mathcal{A}$ in two sets

- The validation data
$$\mathcal{V} = \{n\}$$
where $\{n\}$ is the n-th sample of $\mathcal{A}$. Basically, the validation set $\mathcal{V}$ contains only one sample.
- The training data
$$\mathcal{T} = \mathcal{A} \setminus \{n\}$$
that contains all samples execpt for the one used for validation.

We can use $\mathcal{T}$ to train a model and then we use $\mathcal{V} = \{n\}$ to check the goodness of the model (i.e. the difference between t_n and the model trained with $\mathcal{T} = \mathcal{A} \setminus \{n\}$). We can now repeat the same process for all values of n and average the results

$$L_{LOO} = \frac{1}{N} \sum_{n=1}^{N} \left(t_n - y_{\mathcal{A}\setminus\{n\}}(\mathbf{x}_n)\right)^2 \tag{6.1}$$

Basically, for each model, we are evaluating it using only one point and then averaging the results. This validation technique is almost unbiased (it's slightly pessimistic) and very accurate (because the models are trained with a lot of data) but it's computationally expensive and in practice unfeasible because we have to train many models (one for each data point).

k-fold cross validation

k-fold cross validation is a generalisation of Leave One Out and allows to get better computational performances while slightly decreasing accuracy. In particular, we have to divide the set $\mathcal{A}$ (data set $\mathcal{D}$ minus test data) in k equal parts $\mathcal{A}_1, \ldots, \mathcal{A}_k$ and for each partition i we

1. Learn the model $y_{\mathcal{A}\setminus\mathcal{A}_i}$ using all data not in $\mathcal{A}_i$.
2. Estimate the error of $y_{\mathcal{A}\setminus\mathcal{A}_i}$ using the points in $\mathcal{A}_i$

$$L_{\mathcal{A}_i} = \frac{k}{N} \sum_{(\mathbf{x}_n, t_n)\in\mathcal{A}_i} \left(t_n - y_{\mathcal{A}\setminus\mathcal{A}_i}(\mathbf{x}_n)\right)^2$$

After considering all partitions we can average the errors for each partition hence obtaining

$$L_{kfold} = \frac{1}{k} \sum_{i=1}^{k} L_{\mathcal{A}_i} \tag{6.2}$$

Notice that the bigger the value of k, the better the result is and the worst the computational performance is (for $k = N$ we obtain LOO cross validation,). A typical value for k is 10.

Chapter 7

Managing the bias-variance trade-off

As we have understood, managing the bias-variance trade-off is fundamental to build the best model possible. At first, we can use **model selection** techniques to reduce one or the other quantity. In particular, we will analyse

- **Feature selection**. Features selection allows to keep only useful features, hence reducing the hypothesis space.
- **Regularisation**. Regularisation uses all features but some coefficients are penalised (i.e. shrunken to 0).
- **Dimension reduction**. Dimension reduction allows to combine features to get a smaller number of features. Notice that this technique is different from feature selection in fact the former projects the input on new features while the latter uses a subset of the features.

Such techniques reduce one quantity, yet increasing the other. Luckily there are other techniques, called **model ensemble** techniques, that allow to reduce bias or variance while not changing the other. In particular such techniques are

- **Bagging**. Bagging reduces variance without increasing bias.
- **Boosting**. Boosting reduces bias without increasing variance.

Unfortunately these techniques can't be applied on the same model (i.e. don't generate good models if used together).

7.1 Model selection

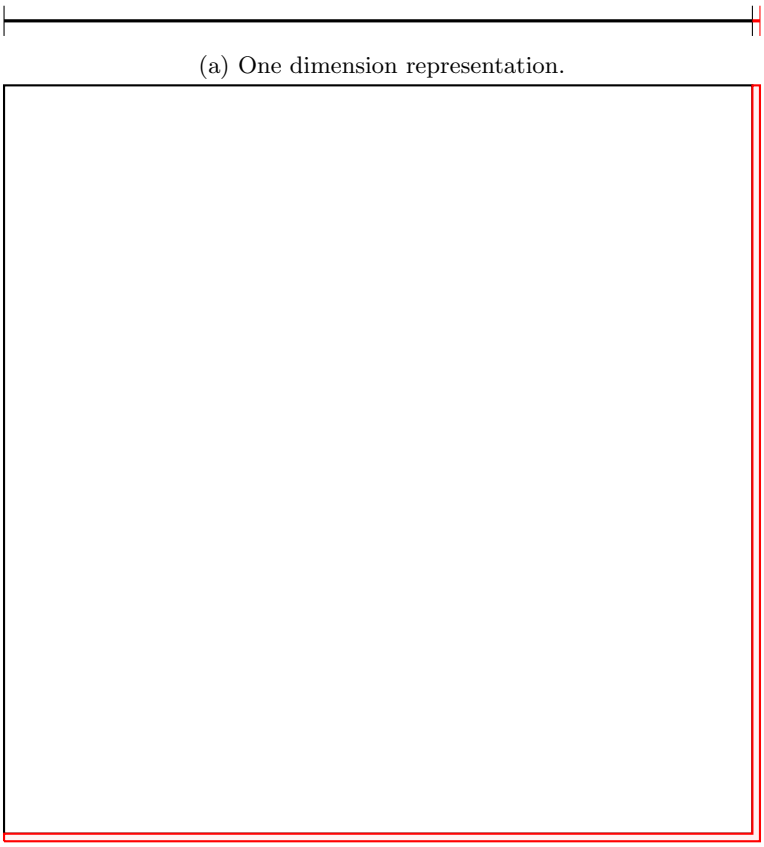
Model selection deals with the fact that having input data with a lot of features might lead to some problems, some of which are

- The model has a **large variance**, because a model with too many features usually models the training set instead of the actual function f.
- **Many samples** are needed, in fact we need N^D samples.

- Optimising the parameters is **expensive**.

Furthermore, when a model performs badly, one might think to add some features to better model f. However this might lead to worst model.

To better understand this concept, consider the following example. Let us consider a line of length 1 and say we can take a sample from one point in the line with uniform probability. In this configuration, the probability to pick a point from 0.99 to 1 is $0.01 = 1 - 0.99$. If we extend the problem to two dimensions, the probability to pick a point in the red area (Figure 7.1b) is $1 - 0.99^2$. Now we can easily generalise and if we consider N dimensions, the probability becomes $1 - 0.99^N$. When N goes to infinity (i.e. we increase the number of dimensions of the model a lot) we notice that the probability goes to 1.



(a) One dimension representation.

(b) Two dimensions representation.

Figure 7.1: A graphical representation of the problem of adding dimensions.

7.1.1 Feature selection

There exists many classes of techniques (i.e. meta-heuristics) to select relevant features:

- **Filters**. Filters rank the features and select the top k.

- **Embedded feature selection**. Embedded techniques allow to select features while training the model. We have already seen some embedded techniques, for instance lasso was able to shrink the value of some parameters to 0 (i.e. to eliminate some features). Other embedded techniques are **decision trees** and **auto encoding**. In particular, in decision trees, leaves represent classes (i.e. we are talking about classification) and branches represent combination of features that lead to those class labels. The nodes closer to the root are the most important ones, while those closer to the leaves are less important, thus can be pruned. Removing a node means removing some features. Notice that regularisation techniques like lasso can also be used to do feature selection only, namely, the model has to be trained with other techniques.
- **Wrappers**. Wrappers start from a subset of features and try to add or remove some other features to obtain a better model. In particular
  - **Forward selection** starts from an empty model and adds the best features, one at a time (i.e. at each iteration we add the best feature among the ones not selected).
  - **Backward selection** starts from a model with all features and removes the least useful feature one at a time.

7.1.2 Regularisation

Regularisation techniques like ridge and lasso can be used to shrink some parameters towards zero (and in case of lasso even to 0). If we can decrease the values of the parameters or even null them, we can reduce the variance of the model. Regularisation uses an adjusted loss function that penalises complex models

$$L(\mathbf{w}) = L_D(\mathbf{w}) + \lambda L_W(\mathbf{w})$$

thus we have to choose an appropriate value of λ.

If we choose λ that minimises the train error, we would obtain $\lambda = 0$ because the train error is low when the model fits perfectly the training set. Therefore the solution is to use cross validation. The basic idea is to compute the error using cross validation for different values of λ and then choose the value of λ with the best error result (i.e. we select the model with the smallest error computed using cross validation). Finally, the model is re-fit using all points in the data-set (apart from the ones used for testing) and the selected λ.

7.1.3 Dimension reduction

Dimension reduction techniques take the original feature space and compress it in a new feature space with less dimensions. Consider for instance a 3d space; if the points lie (approximately) on a plane, then we need only two coordinates to describe them. This means that we can project the points in an appropriate new 2d space, thus reducing the number of dimensions. If the previous example isn't enough, here is another. Consider an algorithm that has to classify some faces (e.g., say if an image of a face represents a smiling person or not). In general the algorithm is fed with all the pixels of the image. This representation is good to describe any image, however we need to represent only images of faces (which are much less than all possible images). The idea is to find a better representation of an image that is able to describe faces only. Basically, we are trying to find a new representation that better fits best the problem. Notice that this means that dimension reduction is an unsupervised learning problem because we have no data that tells us what the best representation is (i.e., we have no target).

There exist many techniques to reduce the number of dimensions of the input space, the most known are

- **Principal Component Analysis** (PCA).
- **Independent Component Analysis** (ICA).
- **Self-organising maps**.
- **Auto-encoders**.

Principal Component Analysis

The basic idea behind PCA is to consider the directions along which the input data has the largest dispersion (i.e., the highest variance). Algorithmically,

1. Find the direction with the largest dispersion. Such direction is called Principal Component.
2. Remove the direction found at the previous point.
3. Find the direction (orthogonal to the already selected ones) with the largest dispersion among the remaining ones.
4. Remove the direction found at the previous point.
5. Repeat from step 3 until we have found enough directions.

In other words, PCA sorts the directions in decreasing order of variance and selects the ones with largest variance. After obtaining the best dimensions, we have to project the original input points in the new space. This means that we will lose some information but, since we selected the directions with highest variance (i.e., we have discarded to ones with lowest variance), the information loss is minimal. Let us further stress two concepts

- When we select a direction, we select the one such that when the data is projected onto that direction, it has the **maximum variance**.
- When we select a new direction, it **has to be orthogonal** to the directions already selected because we want to find an orthogonal basis.

Algorithm The PCA algorithm works as follows:

1. Centre the data on the mean.
$$\bar{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$
We are going to use the mean-centred inputs $\bar{\mathbf{x}}$ to compute the principal components.
2. Compute the covariance matrix $\mathbf{S}$.
$$\mathbf{S} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T$$
Remember that, $\mathbf{S}$ is a $D \times D$ matrix, with D number of features, because it's the sum of the product of $D \times 1$ and $1 \times D$ matrices. Moreover, $\mathbf{S}$ is has variance values on the diagonal and the covariance values elsewhere since it computes $\frac{1}{N-1}(\mathbf{x}_i - \mathbb{E}[\mathbf{x}_i])(\mathbf{x}_j - \mathbb{E}[\mathbf{x}_j])$, for every couple of inputs $\mathbf{x}_i$, $\mathbf{x}_j$ (which is the definition of the covariance).

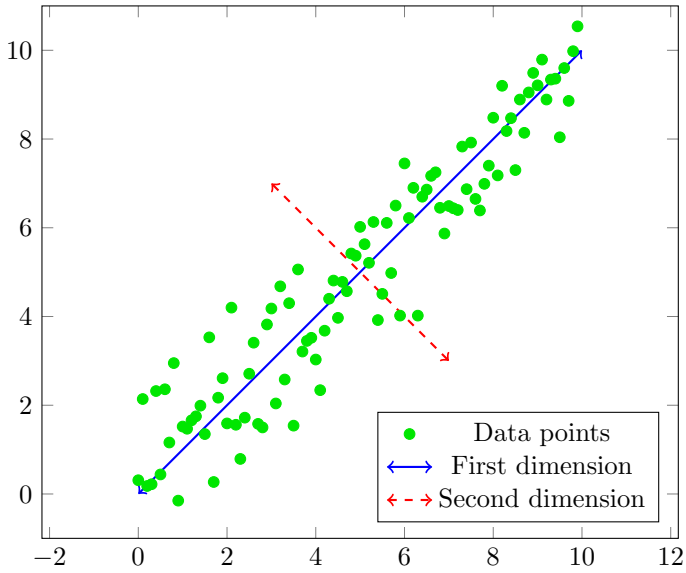


Figure 7.2: Principal components in PCA.

3. Compute the eigenvalues λ_i and eigenvectors e_i of $\mathbf{S}$ (see Appendix B.3). Each eigenvector represents a direction that can be chosen as principal component.

4. The eigenvector e_i with the largest eigenvalue (remember that each eigenvector is associated to an eigenvalue) is the first principal component. If we want to select k components, we have to take the k eigenvectors with the highest eigenvalues. Notice that

$$\frac{\lambda_k}{\sum_i \lambda_i}$$

is the proportion of variance captured by the k-th component. Also remember that we choose the eigenvalues in decreasing order because eigenvalues tell how points are spread along an eigenvector (i.e. what is the dispersion).

After selecting k components we have to project the input points in the new space (i.e. on the new orthogonal basis). To do so, we have to compute the linear combination of the old features using the values of the eigenvectors as coefficients. Basically, the new input data $\mathbf{X}'$ can be computed as

$$\mathbf{X}' = \mathbf{X}\mathbf{E}_k$$

where $\mathbf{E}_k$ is the matrix of the eigenvectors (i.e. every column in an eigenvector)

$$\mathbf{E}_k = \left(\mathbf{e}_1, \ldots, \mathbf{e}_k\right)$$

Notice that $\mathbf{E}_k$ is a $D \times k$ matrix thus if we post-multiply it for $\mathbf{X}$ (which is a $N \times D$, because each row of $\mathbf{X}$ is a data vector), we obtain a $N \times k$ matrix, and in fact $\mathbf{X}'$ is should have the same number of samples but less dimensions (from D to k).

Note that, the new representation introduces some error because we have lost some information, still, most of the times, the benefits brought by dimension reduction are much more than the error introduced.

Choosing the number of components PCA selects k components, thus we have to choose a value for k. We can use for instance the following method

1. Say we want to keep $\varphi = 0.9$ (i.e. 90 %) of the dimensions.
2. We select eigenvalues (and eigenvectors) until the sum of the selected eigenvalues is φ percent the total sum of eigenvalues.
$$k : \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{D} \lambda_i} = \varphi$$

Characteristics PCA can help

- **Supervised learning** because, having less dimensions, we obtain a simpler hypothesis space that reduces the risk of over-fitting.
- **Noise reduction**.

PCA it's not always enough, in particular

- It may fail when data is divided in clusters.
- It may fail when the direction with the greatest variance isn't the most informative one.
- With many dimensions, PCA might be computationally hard.
- PCA works when the linear combination of features is enough to describe the new input space, otherwise we should choose another technique. However, many times, data lies on a linear dimensions (in such cases we can use the ISOMAP method).

7.2 Model ensembles

Model selection methods allow us to reduce bias while increasing variance or vice versa. However, there are other techniques that, if applied on the right models, allow to reduce one quantity without increasing the other. In particular

- **Bagging** reduces **variance** without increasing bias.
- **Boosting** reduces **bias** without increasing variance.

The basic idea behind these techniques is to combine different models to obtain a better model.

7.2.1 Bagging

Bagging is a technique that allows to reduce the variance of a model without increasing its bias. Some important assumptions have to be done, in fact bagging works if

- The model is **over-fitting** (i.e., has high variance).
- The **samples** in the data-set are independent.

In particular, bagging exploits the fact that averaging reduces the variance, thus we can average many over-fitting models to decrease variance without increasing bias.

The main problem with bagging is that we want to build many over-fitting models with only one data set $\mathcal{D}$. One solution for solving this issue is splitting $\mathcal{D}$ in many data sets $\mathcal{D}_i$, each of which can be used for a different model. This means that every model is trained with a small data-set, hence increasing variance (which is a problem). To solve this issue we have to find a trade-off between having small training set and not using the same samples on multiple models. In practice, we can use as much data as possible that is only a little correlated.

Bootstrap aggregation

The bootstrap aggregation algorithm

1. Generates B bootstrap samples of the training set using random sampling with replacement. More precisely, given an input set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, we create B input sets $\mathbf{X}_1, \ldots, \mathbf{X}_B$ containing N input points each. For each set $\mathbf{X}_i$, the input points are drawn at random from $\mathbf{X}$, with replacement, so that $\mathbf{X}_i$ might contain the same point of $\mathbf{X}$ multiple times, whereas other points in $\mathbf{X}$ may be absent from $\mathbf{X}_i$. This allows to obtain samples that have the same dimension of the data-set $\mathcal{D}$ but that are not very correlated.
2. Trains a classifier or a regression function using each bootstrap sample. The prediction is done
   - By **majority voting** for the classifier. Basically, the algorithm chooses the class that was chosen by the majority of the classifiers.
   - **Averaging** the predicted values for the regression function.

This algorithm

- Reduces variation.
- Improves performance for unstable learners (i.e. that vary if the input samples change).
- Works well with decision trees and random forests.

7.2.2 Boosting

Boosting is a technique that allows to reduce the bias of a model (of a set of models) without increasing the variance. Boosting starts from simple models (i.e., under-fitting models with high bias) that have to be slightly better than a random predictor. Such weak models are trained and then sequentially combined to have a much better model (i.e., a more accurate one) with low bias. In general, a boosting algorithm works as follows

1. Assign equal weights to all samples in the training set.
2. Take a weak model.
3. Train the model with the training set in order to have a predictor that is slightly better than a random one.
4. Compute the error of the model on the weighted train set.
5. Train a new model giving more weight to the samples $\mathbf{x}_i$ on which the error (e.g., $t_i - \mathbf{x}_i$) is greater (i.e., on the samples where model gets wrong).

6. Repeat from point 4, until tired.

7. The final model is a weighted prediction of each model

Notice that starting from weak models is very important, in fact otherwise we would increase the variance of the final model. Another important thing to keep in mind is that the algorithm can improve performance only after some iterations, thus at the beginning we might not notice improvements in the predictions. AdaBoost, an implementation of boosting, is shown in Algorithm 2.

Algorithm 2 AdaBoost.

Require: Training set: $\mathcal{D} = \{(x_i, t_i) : i \in \{1, \dots, m\}\}$
Require: Learner(training set, weights): $L(\mathcal{D}, w)$
Require: Number of rounds: T
  for all $i \in \mathbf{D}$ **do**
    $w_1(i) = \frac{1}{N}$ ▷ initialise weights with uniform distribution
  end for
  for $r = 1$ to T **do**
    for all $i \in \mathbf{D}$ **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$ ▷ Normalise the weights
    end for
    $y_r = L(\mathcal{D}, p_r)$ ▷ train the model using logistic regression
    $\varepsilon_r = \sum_j p_r(j) 1[y_r(j) \neq t_j]$ ▷ compute the weighted error (sum of weights of unclassified points)
    if $\varepsilon_r > 0.5$ **then** ▷ if the model is worst than a random predictor, stop
      $T = r - 1$
      exit
    end if
    $\beta_r = \frac{\varepsilon_r}{1-\varepsilon_r}$ ▷ $\beta_r \in [0, 1]$ where 0 is for perfect classifiers, 1 for random ones
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta^{1 - \mathbf{1}[y_r(i) \neq t_i]}$ ▷ update the weights of the classified points reducing them (the exponent of β_r is between 0 and 1)
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \mathbf{1}[y_r(x) = t] \right)$ ▷ $\log \frac{1}{\beta_r}$ is the weight used for voting the prediction

7.2.3 Comparison

Now that we have analysed bagging and boosting, let us list their characteristics

- **Stable models**.
  - Bagging doesn't work well with stable models since it needs over-fitting models, which are inherently unstable (i.e., they change when trained with different data-sets).
  - Boosting might decrease bias when it uses stable models.

- **Performance on noisy data-sets**.
  - Bagging has no performance degradation when using noisy data-sets.
  - Boosting hurts performance on noisy data-sets.
- **Parallelism**.
  - Bagging is easier to parallelise since we can build and train different models in parallel and then average their outputs.
  - Boosting is inherently not parallel because every new model trained depends on the previous one.
- **Average help**. On average, boosting helps more than bagging, but it's also more common for boosting to hurt performance. On the other hand bagging can always improve performance, even by just a little.

Chapter 8

Balancing hypothesis space and data set

The dimension of the hypothesis space $\mathcal{H}$ in which we search a model and the number of samples in the data set $\mathcal{D}$ are related. In particular, when we want to evaluate one, we have to compare it to the other. For instance, we can't say if an hypothesis space is in general too large, but we can say if it's too large for a given data set $\mathcal{D}$ because we compare the dimensions of $\mathcal{H}$ and $\mathcal{D}$. There exist many technique to evaluate the dimension of one set given the other, but we are going to analyse

- **PAC-Learning**.
- **VC-Dimension**.

8.1 PAC-Learning

Overfitting in a model usually happens because the training error is a bad estimate of the prediction error (i.e. the generalisation error). To solve this problem we have introduced adjustment techniques that add a correction term (that discriminates complex models) to the training error. PAC-Learning allows us to obtain a similar result, in fact it tries to say something about the prediction error starting from the training error. Moreover, PAC-Learning allows to estimate how many samples (i.e. $|\mathcal{D}|$) are enough to get a good (i.e. not overfitting) model.

Premise Before diving into the details of PAC-Learning, let us describe the setting we will work in. Notice that this setting is very specific, but the important thing is the methodology used to analyse it. After understanding how PAC-Learning works in such environment, we can apply the same reasoning to another settings. Let us consider

- A **classification problem**.
- A set of input samples $\mathbf{X}$.
- A finite set of hypothesis $\mathcal{H}$.
- A set C of possible Boolean functions that generated the output of $\mathbf{X}$. Let us call this set target concepts set. Notice that the functions are Boolean because we are considering a binary

classification problem, thus the function has to say if a sample belongs to a class (and if not, we infer it belongs to the other class).

- Training instances generated by a fixed, unknown probability distribution $\mathcal{P}$ over $\mathbf{X}$. Basically, $\mathcal{P}$ returns the probability that a certain sample belongs to the data set $\mathcal{D}$.
- A data set $\mathcal{D} = \{\langle x, c(x)\rangle\}$ where each sample is made of a feature $x \in \mathbf{X}$ and a target obtained from the concept $c \in C$ that we are trying to model. Notice that, x is draw with probability given by $\mathcal{P}$ and c is deterministic.

The learner observes $\mathcal{D}$ and must learn and output a function $h \in \mathcal{H}$ that estimates c with the highest accuracy possible. Such function is evaluated using the true loss function L_{true} that computes the probability that the model h is different from the true concept c

$$L_{true} = Pr_{x \in P}\big[c(x) \neq h(x)\big] \tag{8.1}$$

Basically, L_{true} is the probability of the error knowing the probability distribution $\mathcal{P}$ with which the samples are drawn. Notice that we can't compute this value because $\mathcal{P}$ is unknown. The goal of PAC-Learning is to find an upper bound of L_{true} (which is unknown) given the training error L_{train} (that can be computed and is somehow an estimator of the bias). In formulas, we want to find a function g of the training error L_{train} that approximates L_{true}

$$L_{true} \leq g(L_{train})$$

Before finding the actual function g, let us analyse some properties it should have

- It has to increase when the hypothesis space enlarges.
- It has to decrease when the number of samples increases.

Finally it's all set up, now we can start describing PAC-Learning in the details. In particular we will

1. Start with simple assumptions (version spaces).
2. Drop such assumptions and deal with a more generic case.

8.1.1 Version spaces

Let us consider that in the hypothesis space $\mathcal{H}$ there exists some function that has null training error (i.e., $L_{train} = 0$). The subset of $\mathcal{H}$ that contains such functions is called version space $VS_{\mathcal{H},\mathcal{D}}$. Notice that we highlight not only the hypothesis space but also the data set on which VS has been computed, because the train error depends on the samples in $\mathcal{D}$. The true error of the functions in $VS_{\mathcal{H},\mathcal{D}}$ can be whatever value. The following theorem defines an upper-bound for the probability to pick a bad hypothesis.

Theorem 8.1 (PAC learning in version spaces)**.** *If the hypothesis space $\mathcal{H}$ is finite and $\mathcal{D}$ is a sequence of $N \geq 1$ independent random examples of some target concept $c \in C$, then for any accuracy $0 \leq \varepsilon \leq 1$, the probability that $VS_{H,D}$ contains an hypothesis (true) error L_{true} greater than ε is less then $|\mathcal{H}|e^{-\varepsilon N}$.*

$$Pr\big(\exists h \in \mathcal{H} : L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon\big) \leq |\mathcal{H}|e^{-\varepsilon N} \tag{8.2}$$

Before understanding how we get to such result, let us highlight some characteristics of Equation 8.2,

- The probability of getting a bad model (i.e., a model with L_{true} greater than ε) increases with increasing dimensions of the hypothesis space $\mathcal{H}$.
- The probability of getting a bad model decreases when the number of samples increases.

Let us now obtain and prove the result of Theorem 8.1. We can start rewriting the probability that there exists a model h as the disjunction of the probability that a model h_i has true error greater than the threshold ε and 0 training error, for each $h_i \in \mathcal{H}$.

$$\begin{aligned} Pr\big(\exists h \in \mathcal{H} : L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon\big) = Pr\big((L_{train}(h_1) = 0 \wedge L_{true}(h_1) \geq \varepsilon) \\ \vee Pr\big((L_{train}(h_2) = 0 \wedge L_{true}(h_2) \geq \varepsilon) \\ \vee \dots \\ \vee (L_{train}(h_{|\mathcal{H}|}) = 0 \wedge L_{true}(h_{|\mathcal{H}|}) \geq \varepsilon)\big) \end{aligned} \tag{8.3}$$

$$\tag{8.4}$$

Now we can do the first approximation (i.e. the union bound) and disregard the intersection between events (i.e. we don't consider the dependency between events). In formulas, the probability of multiple events to happen can be approximated with the sum of the single probabilities (remember that $p(a \vee b) = p(a) + p(b) - p(a \wedge b)$).

$$Pr\big(\exists h \in \mathcal{H} : L_{train} = 0 \wedge L_{true}(h) \geq \varepsilon\big) \leq \sum_{h \in \mathcal{H}} Pr(L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon) \tag{8.5}$$

Now we can use the Bayes' rule ($Pr(A|B) = \frac{Pr(A \wedge B)}{Pr(B)}$) to rewrite the sum. In particular the Bayes' rule can be written as

$$Pr(L_{train}(h) = 0 | L_{true}(h) \geq \varepsilon) = \frac{Pr(L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon)}{L_{true}(h) \geq \varepsilon} \tag{8.6}$$

Thus we can obtain

$$Pr(L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon) = Pr(L_{train}(h) = 0 | L_{true}(h) \geq \varepsilon) \cdot Pr(L_{true}(h) \geq \varepsilon) \tag{8.7}$$

If we assume that h is a bad model, then the error is surely greater than ε and $Pr(L_{true}(h) \geq \varepsilon)$ is 1. Therefore, we can write

$$Pr\big(\exists h \in \mathcal{H} : L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon\big) \leq \sum_{h \in \mathcal{H}_{bad}} Pr(L_{train}(h) = 0 | L_{true}(h) \geq \varepsilon) \tag{8.8}$$

Note that this is a further approximation since we are only considering bad models. Let us now analyse the probability $Pr(L_{train}(h) = 0 | L_{true}(h) \geq \varepsilon)$. First of all let us notice that, the probability is conditioned to the event $L_{true}(h) \geq \varepsilon$, then we know that the true error done by h is at least ε (i.e. it's ε in the best scenario). Moreover, we are considering only bad models (i.e., $h \in \mathcal{H}_{bad}$), thus we know that $L_{true}(h)$ is the true error of bad models. If we recall Equation 8.1, we understand that L_{true} is the probability that the model is not accurate, hence $L_{true} = Pr(h \text{ not accurate}) = \varepsilon$. Putting all together, the probability that the model doesn't do an error (i.e., $L_{train} = 0$) is 1 minus

the probability that the model is not good, i.e. ε. For this reason, then the probability that the train error is 0 can be computed as $(1-\varepsilon)$ for each of the N samples. This allows us to write

$$Pr\big(\exists h \in \mathcal{H} : L_{train}(h) = 0 \wedge L_{true}(h) \geq \varepsilon\big) \leq \sum_{h \in \mathcal{H}_{bad}} (1-\varepsilon)^N \tag{8.9}$$

If we now consider all possible models (not only the bad ones) we obtain a further estimation because the former are more than the latter. Furthermore, since $(1-\varepsilon)^N$ doesn't depend on h we can write

$$Pr\big(\exists h \in \mathcal{H} : L_{train} = 0 \wedge L_{true}(h) \geq \varepsilon\big) \leq |\mathcal{H}|(1-\varepsilon)^N \tag{8.10}$$

We are basically done, since the last approximation contains all factors we need. In some cases it's useful to further approximate this result to better compare this evaluation with other techniques. In particular we use the inequality $1-\varepsilon \leq e^{-\varepsilon}$ to write

$$Pr\big(\exists h \in \mathcal{H} : L_{train} = 0 \wedge L_{true}(h) \geq \varepsilon\big) \leq |\mathcal{H}|e^{-\varepsilon N} \tag{8.11}$$

PAC-Learning

Now we can decide a value δ for the probability $Pr\big(\exists h \in \mathcal{H} : L_{train} = 0 \wedge L_{true}(h) \geq \varepsilon\big)$. Basically, we are imposing that it has to be at most δ.

$$|\mathcal{H}|e^{-\varepsilon N} \leq \delta \tag{8.12}$$

Thanks to Equation 8.12 we can fix a desired value of δ and of the accuracy error ε to obtain the minimum number of samples needed to obtain such probability and accuracy

$$N \geq \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\left(\frac{1}{\delta}\right)\right) \tag{8.13}$$

Basically we are asking what is the number of samples N needed to have an accuracy of ε with a probability δ. Notice that the number of samples increases if we want to obtain a small error, because $0 \leq \varepsilon \leq 1$ (in the limit case, if we want 0 error we need an infinite number of samples).

In the same way, we can fix a desired number of samples N and a probability δ and obtain the accuracy error with probability δ having N samples.

$$\varepsilon \geq \frac{1}{N}\left(\ln|\mathcal{H}| + \ln\left(\frac{1}{\delta}\right)\right) \tag{8.14}$$

In this case we want to know the accuracy obtained, with probability δ, if we have N samples. Notice that the error decreases (i.e. the accuracy increases) when the number of samples N increases (in the limit case with infinite samples we have 0 error).

Now that we know how PAC-Learning works, we can understand why it's called like that. In particular PAC means Probably Approximately Correct, in fact the result we obtain (either ε given N or vice-versa) holds in probability, in particular with a probability δ.

One final remark: the dimension of the hypothesis space is considered with the logarithm. Say we have M Boolean inputs to the functions in $\mathcal{H}$ (i.e. $h(a_1, \ldots, a_m) \forall h \in \mathcal{H}$), then we have 2^{2^M} functions. If we consider the logarithm of that number we still obtain an exponential growth of $|\mathcal{H}|$ with the number of inputs. If we consider a smaller space, like the one in which the functions are conjunction of M literals, then we have 3^M functions because a literal is either positive, negative or not present. In this case the hypothesis space grows linearly with the number of literals.

PAC-learnability

Consider a class C of possible target concepts defined over a set of instances $\mathbf{X}$ of length n and a learner L using hypothesis space $\mathcal{H}$.

Definition 8.1 (PAC-learnable)**.** *A class C is PAC-learnable if there exists an algorithm L such that for every $f \in C$, for any distribution $\mathcal{P}$, for any ε such that $0 \leq \varepsilon < \frac{1}{2}$, and δ such that $0 \leq \delta < \frac{1}{2}$, algorithm L, with probability at least $1 - \delta$, outputs a concept h such that $L_{true}(h) \leq \varepsilon$ using a number of samples that is polynomial of $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$.*

Definition 8.2 (Efficiently PAC-learnable)**.** *A class C is efficiently PAC-learnable by L using $\mathcal{H}$ if and only if for all $c \in C$, distributions $\mathcal{P}$ over $\mathbf{X}$, ε such that $0 < \varepsilon < \frac{1}{2}$, and δ such that $0 < \delta < \frac{1}{2}$, learner L will with probability at least $1 - \delta$ output a hypothesis $h \in \mathcal{H}$ such that $L_{true} \leq \varepsilon$, in time that is polynomial in $\frac{1}{\varepsilon}$, $\frac{1}{\delta}$, M and size(c).*

8.1.2 Agnostic learner

Usually the train error isn't zero (i.e. the version space is empty), thus we have to define a new bound to the true error. In formulas,

$$L_{true} \leq L_{train} + \varepsilon \tag{8.15}$$

where ε is the desired accuracy. To obtain this new bound we have to use the Hoeffding bound, i.e. for N independent identically distributed binary events $X_1, \ldots, X_N$ where X_i is a Bernoulli (i.e. $X_i \in \{0, 1\}$) and $0 < \varepsilon < 1$, we get the following bound

$$Pr\big(\mathbb{E}[\bar{X}] - \bar{X} > \varepsilon\big) < e^{-2N\varepsilon^2} \tag{8.16}$$

where $\bar{X}$ is the empirical mean defined as

$$\bar{X} = \frac{1}{N}(X_1 + \cdots + X_N)$$

This bound can be rewritten in terms of L_{true} and L_{train} to obtain the following theorem

Theorem 8.2. *Given a finite hypothesis space $\mathcal{H}$, a data set $\mathcal{D}$ with N independent identically distributed samples and $0 < \varepsilon < 1$, then for any learned hypothesis h holds*

$$Pr(\exists h \in \mathcal{H} : L_{true}(h) - L_{train}(h) > \varepsilon) \leq |\mathcal{H}| e^{-2N\varepsilon^2} \tag{8.17}$$

Thanks to Theorem 8.2 we can fix a maximum value δ for the probability to obtain

$$|\mathcal{H}| e^{-2N\varepsilon^2} \leq \delta \tag{8.18}$$

Now we can finally obtain the value of ε

$$\varepsilon \geq \sqrt{\frac{\ln |\mathcal{H}| + \ln \frac{1}{\delta}}{2N}}$$

and use it to estimate the true error

$$L_{true} \leq L_{train} + \sqrt{\frac{\ln |\mathcal{H}| + \ln \frac{1}{\delta}}{2N}} \tag{8.19}$$

Notice that in Equation 8.19 the L_{train} represents bias, while the error ε represents the variance of the model. In particular we can analyse the variance and say that, as expected,

- The variance increases with a larger hypothesis space $\mathcal{H}$.
- The variance decreases with a large number N of samples.

Also notice that we can use Equation 8.18 to compute (in probability), how many samples are needed to obtain an error ε with a probability δ

$$N \geq \frac{1}{2\varepsilon^2}\left(\ln |\mathcal{H}| + \ln \frac{1}{\delta}\right) \tag{8.20}$$

8.1.3 Continuous hypothesis space

Until now, we have considered a finite hypothesis space $\mathcal{H}$. However $\mathcal{H}$ could be continuous, thus having infinite models (i.e. $|\mathcal{H}| = \infty$). This means that we would need an infinite number N of samples (see Equation 8.20). However, the variance considers many different functions, but not functions that are very similar one another. This means that in case of continuous hypothesis space we have to add the concept of how informative $\mathcal{H}$ is.

8.2 VC-Dimension

8.2.1 Dichotomy and shattering

Before explaining what VC-Dimension is, we have to introduce two important concepts.

> **Definition 8.3** (Dichotomy)**.** *A dichotomy of a set of samples S is a partition of S into two disjoint subsets.*

> **Definition 8.4** (Shattering)**.** *A set of instances S is shattered by the hypothesis space $\mathcal{H}$ if and only if for every dichotomy of S there exists some hypothesis $h \in \mathcal{H}$ consistent with this dichotomy (i.e. that correctly classifies all samples in S).*

In other words, S i shattered by $\mathcal{H}$ if for every partition of S, i.e., for every way we can classify the points in S, we can find a model $h \in \mathcal{H}$ that can correctly classify the points as defined by the partition. If a set S is shattered by an hypothesis space $\mathcal{H}$ then it means that $\mathcal{H}$ is very flexible.

Example

Let us consider the sample space $\mathbf{X}$ (i.e., S) in Figure 8.1 and the space $\mathcal{H}$ of linear classifiers. The figure shows a model that classifies all three points for a given dichotomy, however one can verify that whatever dichotomy (i.e., whatever way to classify the points) we choose, there always is a

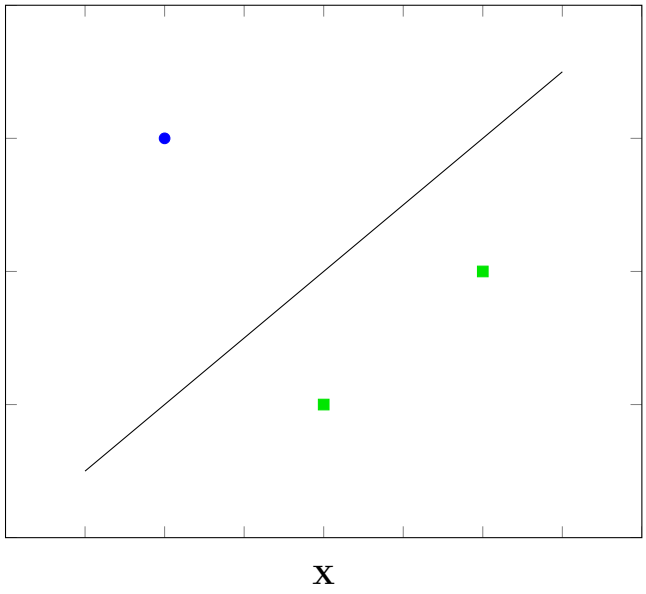


Figure 8.1: A sample set $\mathbf{X}$ shattered by the linear model hypothesis space.

linear model consistent with the dichotomy. If we consider the sample space in Figure 8.2 and the same hypothesis space of linear classifiers, we can notice that it doesn't exist a linear model that can correctly classify every point.

8.2.2 VC-Dimension

Now that we have introduced the concepts of dichotomy and shattering, we can define a VC dimension.

> **Definition 8.5** (VC-Dimension)**.** *The Vapnik-Chervonenkis dimension $VC(\mathcal{H})$ of hypothesis space $\mathcal{H}$ defined over instance space $\mathbf{X}$ is the size of the largest finite subset of $\mathbf{X}$ shattered by $\mathcal{H}$. If an arbitrarily large finite set of $\mathbf{X}$ can be shuttered by $\mathcal{H}$, then $VC(\mathcal{H}) \equiv \infty$.*

Depending on the sample space, we can define what is the VC-Dimension of the hypothesis space $\mathcal{H}$. In particular, for a M-dimensions sample space $\mathbf{X}$ and a linear classifier, the VC-Dimension is $M + 1$.

One can expect that the number of parameters of a model matches the maximum number of points in the sample space (i.e., the VC dimension), however in general this isn't always true. For instance, there are cases in which the number of parameters is infinite but the VC-Dimension is finite (which is bad, because, since the number of parameters is infinite, we have infinite variance). On the other hand, there exist hypothesis spaces with one parameter (very low variance) and an infinite VC-Dimension.

Examples

Here's a list of VC-Dimensions for different hypothesis spaces

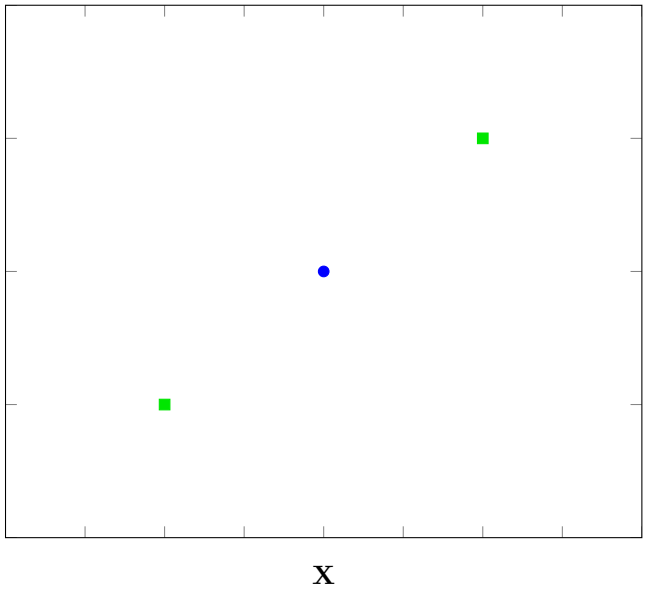


Figure 8.2: A sample set $\mathbf{X}$ not shattered by the linear model hypothesis space.

- A linear classifier in a M-dimensional sample space (i.e. a linear classifier with M features, plus a constant term) has a VC-Dimension of $M + 1$

$$VC(\mathcal{H}_{linear}) = M + 1$$

- A neural network has a VC-Dimension equal to the number of parameters P.

$$VC(\mathcal{H}_{neuralnetwork}) = P$$

- A 1-Nearest neighbour classifier has a VC-Dimension of infinite

$$VC(\mathcal{H}_{1-nearest}) = \infty$$

- A SVM with Gaussian kernel has a VC-Dimension of infinite

$$VC(\mathcal{H}_{SVM}) = \infty$$

8.2.3 Estimating the data set size

The VC-Dimension of an hypothesis space can be used to estimate how many samples are needed to get an error of at most ε with probability at least $1 - \delta$

$$N \geq \frac{1}{\varepsilon}\left(4\log_2\left(\frac{2}{\delta}\right) + 8VC(\mathcal{H})\log_2\left(\frac{13}{\varepsilon}\right)\right) \tag{8.21}$$

As we can see from Equation 8.21, the size of the data set $\mathcal{D}$ increases linearly with the required error ε and the VC-Dimension.

As we have done for PAC-Learning we can obtain ε and replace it in Equation 8.15. In this case we get ε from Equation 8.21

$$\varepsilon = \sqrt{\frac{VC(\mathcal{H})\big(\ln \frac{2N}{VC(\mathcal{H})} + 1\big) + \ln \frac{4}{\delta}}{N}}$$

to obtain

$$L_{true}(\mathcal{H}) \leq L_{train}(\mathcal{H}) + \sqrt{\frac{VC(\mathcal{H})\big(\ln \frac{2N}{VC(\mathcal{H})} + 1\big) + \ln \frac{4}{\delta}}{N}} \tag{8.22}$$

As for the PAC case,

- $L_{train}(\mathcal{H})$ represents the bias.
- ε is a bound to the variance.
- The approximation of L_{true} can be used instead of the adjustment techniques. In particular, we have to choose the hypothesis space $\mathcal{H}$ that minimises the bound on L_{true}. Such task is called **Structural Risk Minimisation**.

8.2.4 Properties

VC-Dimensions have some important properties,

- The VC-Dimension of a finite hypothesis space $\mathcal{H}$ is bounded from the logarithm of the cardinality of $\mathcal{H}$.
$$VC(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$$
- The concept class C with $VC(C) = \infty$ is not PAC-learnable, because even with infinite samples we can't face an infinite VC-Dimension.

Chapter 9

Kernel methods

9.1 Kernel

Kernel methods are a category of supervised learning methods. In particular, they are memory-based (or instance-based), which means that the training data is used to predict a new value. In other words, kernels are non-parametric methods. For instance, K-Nearest-Neighbours is a memory-based supervised learning method. These techniques differ from parametric methods because in the latter case, we use the data to train the model (i.e. to find the values of the model's parameters) and then the model's parameters are used to classify new points. Let us point out the differences between parameter-based and memory-based (i.e., non-parametric) techniques

- **Parameter-based** models are **long to train** but can **predict in a short time**.
- **Memory-based** models are very **fast to train** (they don't even need a proper training, but only some data preparation phase) but **need a lot of time for prediction** because they have to use the whole data-set to classify a new point.

Parametric methods and memory-based methods are strongly related, in fact, some parametric methods are strongly related to kernel methods and we can represent a parametric method as a kernel method. To achieve this goal, we will use dual problems, starting from the parametric method and obtaining the kernel one. Another important characteristic of memory-based methods is that they need a metric to say how similar two points are one to the other (i.e., how much correlated their target values are). The metric is a way to insert knowledge about the problem, in fact it's a way for the designer to define when two points are similar. Kernels do exactly this job, in fact they measure the similarity between two points.

If we consider kernel methods (and not memory-based models in general), the prediction will be a combination of the target values in the data-set and some weights that depends on the kernel.

9.1.1 Motivations

In parametric methods, we used non-linear features (thanks to basis functions), linearly combined, to model non linear patters (i.e. to represent non linear relations between the input and the targets). Kernels play the same role, in fact, starting from input variables (not features, there are no features in kernel methods, since we have no parameters), *kernels try to capture non linear patters by saying when two points are similar*. The advantage of kernel methods is that the same result is obtained

with a smaller computational cost, in fact we don't have to generate all the features, for all the input points. In particular, the computation complexity in parametric methods is dominated by the number of features (squared), while in instance-based methods by the number of input points. This means that *kernel methods are fast as long as the number of input points doesn't get too big.*

The power of kernels is that *they can work in a very large feature space (theoretically even an infinite one) without needing to compute all the features* because the work in the sample (i.e., input) space, not in the feature space.

9.1.2 Kernels

The most important building blocks of kernel methods are kernels.

Definition 9.1 (Kernel)**.** *A kernel is a scalar function (i.e., its output is a number) that takes as input two input vectors (i.e., two points) and outputs a number that, the higher it is, the more similar (i.e., the closer) the two input points are.*

$$k : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R} \tag{9.1}$$

A kernel function k must have one more characteristic: it has to be valid.

Definition 9.2 (Valid kernel)**.** *A kernel k is said to be valid if we can write it as the scalar product between two feature vectors of the input points $\mathbf{x}$ and $\mathbf{x}'$*

$$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \cdot \boldsymbol{\phi}(\mathbf{x}') \tag{9.2}$$

In other words, **it must exist a feature mapping (i.e., a basis function) $\boldsymbol{\phi}$ that applied to $\mathbf{x}$ and $\mathbf{x}'$ and computed the scalar product between $\boldsymbol{\phi}(\mathbf{x})^T$ and $\boldsymbol{\phi}(\mathbf{x}')$, returns the same value of the kernel function**. Notice that, we don't have to compute the scalar product (otherwise it would be useless to use kernel methods) but we have to guarantee that the kernel function satisfies this property. Since the scalar product is symmetric, the kernel function is symmetric, too.

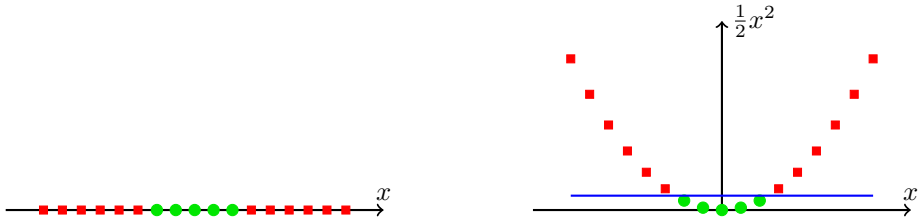
$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$$

Another important property to remember is that, two points $\mathbf{x}$, $\mathbf{x}'$ have similarity 0 (i.e. are not similar at all), when the vectors $\mathbf{x}$ and $\mathbf{x}'$ are orthogonal.

Note that Property 9.2 allows us to map data in higher dimensions so that it can display linear behaviours and we can apply linear regression techniques. Explicitly mapping the input points and then applying parametric linear regression models can be expensive since we have to deal with a large feature space (which slows down training). On the other hand, *kernels allow to map the input in a new higher-dimensional space without paying an high computational price*, since we don't have to explicitly compute the product $\boldsymbol{\phi}(\mathbf{x})^T \cdot \boldsymbol{\phi}(\mathbf{x}')$. A visual representation of how, thanks to a transformation $\boldsymbol{\phi}$, we obtain a linearly separable representation of the input data is shown in Figure 9.1.

From parametric to non-parametric models

Writing the kernel function as the scalar product between the feature vectors of two points allows us to move from parametric models to memory-based models. In particular, we can consider a



(a) Input data with feature space $\mathbf{x} = \{x\}$. (b) Input data with feature space $\boldsymbol{\phi} = \{x, \frac{1}{2}x^2\}$.

Figure 9.1: The points of a data set represented in a one dimensional space and then mapped to a two dimensional space so that they can be linearly classified.

parametric linear regression model and rewrite its solution so that it only contains the scalar product of feature vectors. Since the kernel function can be written as the product of two feature vectors, we can say that the solution of the parametric model can be computed with a kernel function. Now, if we can find a simple kernel function, that respects the property $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}(\mathbf{x}')$, then we have obtained a memory-based model that depends only on the input variables (and not on all the features like the original parametric model) and that has a theoretically infinite number of features. Some examples of memory-based models are

- **Ridge regression** that starts from the parametric version of ridge.
- **Gaussian processes** that start from Bayesian linear regression.
- **Support Vector Machines** (SVMs) that are derived from the perceptron models.

Linear kernel

The simplest kernel we can build has only one feature. Practically, the feature mapping maps the features into themselves (i.e., $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$) and we can write

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \cdot \mathbf{x}' \tag{9.3}$$

Notice that, since we are considering only a linear feature, the linear kernel can only learn linear models.

Definitions

Before exploring into the details some kernel methods, we need some definitions.

Definition 9.3 (Stationary kernel)**.** *A kernel is stationary if it depends only on the difference between the input vectors. In other words, a kernel is stationary if it's a function of* $\mathbf{x} - \mathbf{x}'$*.*

$$k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}') \tag{9.4}$$

Definition 9.4 (Homogeneous kernel)**.** *A kernel is homogeneous if it's a function only of*

the Euclidean norm (i.e., distance) between the input vectors.

$$k(\mathbf{x}, \mathbf{x}') = k(\|\mathbf{x} - \mathbf{x}'\|) \tag{9.5}$$

9.1.3 Valid kernels

Direct application of the definition

The first way to build a valid kernel function is to use the definition.

$$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}(\mathbf{x}') = \sum_{n=1}^{M} \phi_n(\mathbf{x}) \phi_n(\mathbf{x}')$$

where M is the dimension of the feature space. Basically, we can verify if a kernel function is valid if we can rewrite the kernel as the scalar product of the feature vectors of the input points.

Mercer's theorem

Applying the definition of kernel directly is computationally too expensive, hence we have to find other ways to check if a kernel function is valid. The Mercer's theorem defines a necessary and sufficient condition for a function to be a kernel.

Theorem 9.1 (Mercer)**.** *Any continuous, symmetric, positive semi-definite (i.e., symmetric with all eigenvalues not negative) Gram matrix $K = \Phi\Phi^T$ can be expressed as a dot product in an high dimensional space.*

The Gram matrix K is a $N \times N$ matrix in which, each cell $\langle n, m \rangle$ contains the kernel computed on $\mathbf{x}_n$ and $\mathbf{x}_m$.

$$K = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix} \tag{9.6}$$

Also note that, every element of the matrix is the product of two feature vectors $\phi(\mathbf{x}_i)$

$$K_{nm} = \phi(\mathbf{x}_n)^T \cdot \phi(\mathbf{x}_n) = \Phi\Phi^T$$

In simpler words, Mercer's theorem states that, if a Gram matrix is symmetric and semi-definite positive, for any set of input points (i.e., any possible data-set), then it represents a valid kernel function.

Notice that, since the theorem should be verified on all possible data-sets, we can only say if a kernel function isn't valid.

Constructive proof

The last method to prove that a kernel function is valid, is based on known valid kernel functions. In particular, we can take some kernel functions that are proven to be valid and combine them using functions that maintain the validity of the input function. Here's a list of functions that preserve the validity of kernel functions.

1. Given a valid kernel function $k_1(\mathbf{x}, \mathbf{x}')$, if we multiply if by a constant c, we obtain a valid kernel function k.
$$k(\mathbf{x}, \mathbf{x}') = c \cdot k_1(\mathbf{x}, \mathbf{x}') \tag{9.7}$$

2. Given two valid kernel functions $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, their sum is a valid kernel function k.
$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') \tag{9.8}$$

3. Given two valid kernel functions $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, their product is a valid kernel function k.
$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') \cdot k_2(\mathbf{x}, \mathbf{x}') \tag{9.9}$$

4. Given a valid kernel function $k_1(\mathbf{x}, \mathbf{x}')$, the kernel function
$$k(\mathbf{x}, \mathbf{x}') = e^{k_1(\mathbf{x}, \mathbf{x}')} \tag{9.10}$$
is valid.

5. Given a valid kernel function $k_1(\mathbf{x}, \mathbf{x}')$ and a function f, the kernel function
$$k(\mathbf{x}, \mathbf{x}') = f(\mathbf{x}) \cdot k_1(\mathbf{x}, \mathbf{x}') \cdot f(\mathbf{x}') \tag{9.11}$$
is valid.

6. Given a valid kernel function $k_1(\mathbf{x}, \mathbf{x}')$ and a polynomial q with non negative coefficients, the kernel function
$$k(\mathbf{x}, \mathbf{x}') = q(k(\mathbf{x}, \mathbf{x}')) \tag{9.12}$$
is valid.

7. Given a valid kernel function $k_1(\mathbf{x}, \mathbf{x}')$ and a function $\phi(\mathbf{x}) : \mathbf{x} \to \mathbb{R}^M$, the kernel function
$$k(\mathbf{x}, \mathbf{x}') = k(\phi(\mathbf{x}), \phi(\mathbf{x}')) \tag{9.13}$$
is valid.

8. Given a symmetric, positive semi-definite matrix A, the kernel function
$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T A \mathbf{x}' \tag{9.14}$$
is valid.

9. Given two valid kernel functions $k_a(\mathbf{x}_a, \mathbf{x}'_a)$ and $k_b(\mathbf{x}_b, \mathbf{x}'_b)$ with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, then the kernel function
$$k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b) \tag{9.15}$$
is valid.

10. Given two valid kernel functions $k_a(\mathbf{x}_a, \mathbf{x}'_a)$ and $k_b(\mathbf{x}_b, \mathbf{x}'_b)$ with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, then the kernel function
$$k(\mathbf{x}, \mathbf{x}') = k_a(\mathbf{x}_a, \mathbf{x}'_a) \cdot k_b(\mathbf{x}_b, \mathbf{x}'_b) \tag{9.16}$$
is valid.

In practice, we can try to write the kernel function using known valid kernel functions combined with the functions above.

9.1.4 Gaussian kernel

One commonly used kernel is the Gaussian kernel.

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2}} \tag{9.17}$$

Before exploring the Gaussian more in detail, let us check that it is valid (Definition 9.2). The first step is expanding the norm.

$$\|\mathbf{x} - \mathbf{x}'\|^2 = \mathbf{x}^T\mathbf{x} + \mathbf{x}'^T\mathbf{x}' - 2\mathbf{x}^T\mathbf{x}' \tag{9.18}$$

Now, we can replace this new representation in Equation 9.17 to obtain

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2}} \tag{9.19}$$

$$= e^{-\frac{\mathbf{x}^T\mathbf{x}+\mathbf{x}'^T\mathbf{x}'-2\mathbf{x}^T\mathbf{x}'}{2\sigma^2}} \tag{9.20}$$

$$= e^{-\frac{\mathbf{x}^T\mathbf{x}}{2\sigma^2}} \cdot e^{-\frac{\mathbf{x}'^T\mathbf{x}'}{2\sigma^2}} \cdot e^{\frac{\mathbf{x}^T\mathbf{x}'}{\sigma^2}} \tag{9.21}$$

Equation 9.21 can be written as

$$k(\mathbf{x}, \mathbf{x}') = f(\mathbf{x}) \cdot e^{\frac{\mathbf{x}^T\mathbf{x}'}{\sigma^2}} \cdot f(\mathbf{x}') \tag{9.22}$$

where $f(x) = e^{-\frac{x}{2\sigma^2}}$. This new representation matches property 9.11, hence if we can prove that $k_1 = e^{\frac{\mathbf{x}^T\mathbf{x}'}{\sigma^2}}$ is a valid kernel, then the Gaussian kernel is also valid. Thanks to rule 9.10 we can say that k_1 is a kernel because it can be written as an exponential of a kernel (in particular the exponential of the linear kernel $\mathbf{x}^T\mathbf{x}$).

Non-Euclidean distances

The Gaussian kernel can also be extended to non-Euclidean distances. In this case, the kernel is

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2\sigma^2}\big(\kappa(\mathbf{x}, \mathbf{x}) + \kappa(\mathbf{x}', \mathbf{x}') - \kappa(\mathbf{x}, \mathbf{x}')\big)\right) \tag{9.23}$$

9.2 Ridge regression

As we have seen, thanks to kernel functions, we can transform a parametric model in a non-parametric model. Let us start from Ridge regression. In particular, we have to start from the loss function of the parametric Ridge regression

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

and try to make $\mathbf{w}$ disappear to write the loss function as the product of feature vectors. As first step we can compute the gradient of $L_{\mathbf{w}}$ with respect to $\mathbf{w}$ and put it to 0 to obtain the optimal parameter vector $\mathbf{w}$. This computation has already be done when discussing Ridge regression (the result is Equation 3.58, also shown below).

$$\mathbf{w} = \big(\lambda\mathbf{I} + \Phi^T\Phi\big)^{-1}\Phi^T\mathbf{t} \tag{9.24}$$

The equation to compute the optimal parameter vector can be rewritten as

$$\mathbf{w} = \big(\lambda\mathbf{I} + \Phi^T\Phi\big)^{-1}\Phi^T\mathbf{t} \tag{9.25}$$
$$\big(\lambda\mathbf{I} + \Phi^T\Phi\big)\mathbf{w} = \Phi^T\mathbf{t} \tag{9.26}$$
$$\lambda\mathbf{I}\mathbf{w} + \Phi^T\Phi\mathbf{w} = \Phi^T\mathbf{t} \tag{9.27}$$
$$\lambda\mathbf{I}\mathbf{w} + \Phi^T\Phi\mathbf{w} - \Phi^T\mathbf{t} = 0 \tag{9.28}$$
$$\lambda\mathbf{I}\mathbf{w} + \Phi^T\big(\Phi\mathbf{w} - \mathbf{t}\big) = 0 \tag{9.29}$$
$$\Phi^T\big(\Phi\mathbf{w} - \mathbf{t}\big) = -\lambda\mathbf{I}\mathbf{w} \tag{9.30}$$
$$(-\lambda^{-1}\mathbf{I})\Phi^T\big(\Phi\mathbf{w} - \mathbf{t}\big) = \mathbf{w} \tag{9.31}$$
$$\Phi^T(-\lambda^{-1}\mathbf{I})\big(\Phi\mathbf{w} - \mathbf{t}\big) = \mathbf{w} \tag{9.32}$$
$$\tag{9.33}$$

Now we can call $\mathbf{a} = (a_1, \dots, a_N)^T$ the term $-\lambda^{-1}\mathbf{I}\big(\Phi\mathbf{w} - \mathbf{t}\big)$. The optimal parameter vector can therefore be written as

$$\mathbf{w} = \Phi^T\mathbf{a}$$

Notice that

- We have a parameter a_i for each sample of the data-set, hence $\mathbf{a}$ has N values.
- Φ^T is a matrix $D \times N$ because Φ is the matrix in which the n-th row is $\boldsymbol{\phi}(\mathbf{x_n})$ (hence Φ is a $N \times D$ matrix).

hence the product between Φ^T and $\mathbf{a}$ is a D-dimensional row vector, which makes sense because the dimension of $\mathbf{w}$ is D. Remember that, $\mathbf{w}$ can be written as $\Phi^T\mathbf{a}$ only when $L_\mathbf{w}$ is optimal (i.e. when the derivative is null, hence the loss is minimum).

If we replace $\mathbf{w} = \Phi^T\mathbf{a}$ in the loss function of Ridge regression (Equation 3.57) we get an expression that contains many times the Gram matrix (9.6). In particular we get,

$$\begin{aligned}
L_\mathbf{w} &= \frac{1}{2}\sum_{i=1}^{N}\big(\mathbf{w}^T\phi(\mathbf{x}_i) - t_i\big)^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w} \\
&= \frac{1}{2}\big(\Phi\mathbf{w} - \mathbf{t}\big)^T\big(\Phi\mathbf{w} - \mathbf{t}\big) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w} \\
&= \frac{1}{2}\big(\Phi\Phi^T\mathbf{a} - \mathbf{t}\big)^T\big(\Phi\Phi^T\mathbf{a} - \mathbf{t}\big) + \frac{\lambda}{2}(\Phi^T\mathbf{a})^T\Phi^T\mathbf{a} \\
&= \frac{1}{2}\big[(\Phi\Phi^T\mathbf{a})^T - \mathbf{t}^T\big]\big[\Phi\Phi^T\mathbf{a} - \mathbf{t}\big] + \frac{\lambda}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{a} \\
&= \frac{1}{2}(\Phi\Phi^T\mathbf{a})^T\Phi\Phi^T\mathbf{a} - \frac{1}{2}(\Phi\Phi^T\mathbf{a})^T\mathbf{t} - \frac{1}{2}\mathbf{t}^T\Phi\Phi^T\mathbf{a} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{a} \\
&= \frac{1}{2}\mathbf{a}^T\Phi\Phi^T\Phi\Phi^T\mathbf{a} - \frac{1}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{t} - \frac{1}{2}\mathbf{t}^T\Phi\Phi^T\mathbf{a} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{a} \\
&= \frac{1}{2}\mathbf{a}^T\Phi\Phi^T\Phi\Phi^T\mathbf{a} - \mathbf{a}^T\Phi\Phi^T\mathbf{t} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T\Phi\Phi^T\mathbf{a}
\end{aligned}$$

Where we have developed $(\Phi\Phi^T\mathbf{a})^T$ as

$$\begin{aligned}(\Phi\Phi^T\mathbf{a})^T &= ((\Phi\Phi^T)\mathbf{a})^T \\ &= \mathbf{a}^T(\Phi\Phi^T)^T \\ &= \mathbf{a}^T(\Phi\Phi^T)^T \\ &= \mathbf{a}^T(AB)^T \\ &= \mathbf{a}^T B^T A^T \\ &= \mathbf{a}^T(\Phi^T)^T\Phi^T \\ &= \mathbf{a}^T\Phi\Phi^T\end{aligned}$$

Now we can replace the product $\Phi\Phi^T$ with the Gram matrix K to obtain

$$L_{\mathbf{a}} = \frac{1}{2}\mathbf{a}^T KK\mathbf{a} - \mathbf{a}^T K\mathbf{t} + \frac{1}{2}\mathbf{t}^T\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^T K\mathbf{a}$$

In this new loss function we have no $\mathbf{w}$ nor Φ and all dimensions of vectors and matrices are in the number of samples N (K is a $N \times N$ matrix, $\mathbf{t}$ and $\mathbf{a}$ are $N \times 1$ vectors).

At this point, $\mathbf{a}$ is a function of $\mathbf{w}$ but we can forget about it and minimise $\mathbf{a}$. In particular, we can compute the gradient of $L_{\mathbf{a}}$ with respect to $\mathbf{a}$, put the gradient equal to 0 and solve for $\mathbf{a}$. What we get is

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

where $\mathbf{I}_N$ is the $N \times N$ identity matrix. As we can notice, this equation looks like the one we obtained for the parametric version of Ridge, however, in this case we have $K = \Phi\Phi^T$ (or dimension $N \times N$) instead of $\Phi^T\Phi$ (of dimension $D \times D$).

9.2.1 Prediction

Our discussion about non-parametric Ridge regression allowed us to compute the optimal vector $\mathbf{a}$ (which isn't a parameter vector since it has dimensions $N \times 1$). Now we can replace $\mathbf{a}$ in the linear regression model (3.1) to get a new linear regression model.

$$\begin{aligned}y(\mathbf{x}) &= \mathbf{w}^T\phi(\mathbf{x}) \\ &= (\Phi^T\mathbf{a})^T\phi(\mathbf{x}) \\ &= \mathbf{a}^T\Phi\phi(\mathbf{x}) \\ &= \mathbf{k}(\mathbf{x})^T(K + \lambda\mathbf{I}_N)^{-1}\mathbf{t}\end{aligned}$$

where $\mathbf{k}(\mathbf{x})$ is the vector of the similarities between $\mathbf{x}$ and the n samples

$$\mathbf{k}(\mathbf{x}) = \begin{bmatrix} k_1 = k(\mathbf{x}_1, \mathbf{x}) \\ \vdots \\ k_N = k(\mathbf{x}_N, \mathbf{x}) \end{bmatrix}$$

The new model

$$y(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T(K + \lambda\mathbf{I}_N)^{-1}\mathbf{t} \tag{9.34}$$

can be used to predict the output t for an input point $\mathbf{x}$. The kernel version of ridge regression returns the same result of the parametric version because the former method is the dual of and it's obtained from the latter. The difference is in the computational power needed,

- The kernel version is very useful when the input space has reduced dimensions but the problem has a very big (even infinite) feature space. In other words, kernel methods are very efficient when we're working in big feature spaces.
- The parametric version is very useful when the input space is very big and we can use a limited feature space.

9.2.2 Similarities

Kernel methods capture the similarity between points. Such similarity can be computed not only for scalar points but also for any type of object like graphs, sets, strings or even documents. For instance, one could define a kernel over sets as

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

9.2.3 Advantages of the kernel-representation

Thanks to the equation

$$\mathbf{w} = \Phi^T \mathbf{a}$$

once we obtain $\mathbf{a}$ we can recover $\mathbf{w}$ as linear combination of the basis functions $\boldsymbol{\phi}(\mathbf{x})$. Note that in this case, the computational complexity is dominated by the inversion of matrix $(K + \lambda \mathbf{I}_N)$, while in the parametric case we have to invert a $(\Phi^T \Phi)$, which is a $M \times M$ matrix. This might seem a disadvantage since the number of features M is usually smaller than the number of samples N, however, in some cases we might want to work with input vectors $\mathbf{x}$ of high (even infinite) dimensions. Moreover, using kernel functions we can avoid working with feature vectors $\boldsymbol{\phi}$.

9.3 Radial basis function networks

To introduce radial basis function networks, we have to define what a radial basis function is.

Definition 9.5 (Radial basis function)**.** *A radial basis function is a bell-shaped, homomorfic function whose value depends on the distance between a given point μ_j (usually called centre) and the point $\mathbf{x}$ where we want to evaluate the function.*

$$\phi_j(\mathbf{x}) = h(\|\mathbf{x} - \mu_j\|_s) \tag{9.35}$$

In other words, the radial function depends only on the distance (usually the Euclidean distance) from a point called centre. In particular, the value of h is bigger for points that are closer to the centre $\boldsymbol{\mu}$. This might seem strange but we have to remember that kernels measure similarity and return high values for points that are similar. For our purposes, we will build as many basis functions as samples in the data set. Each radial basis function ϕ_i will use one point $\mathbf{x}_i$ of the data-set as centre $\boldsymbol{\mu}$.

$$\begin{aligned}\boldsymbol{\phi}(\mathbf{x}) &= (\phi_1(\mathbf{x}), \ldots, \phi_N(\mathbf{x}))^T \\ &= \big(h(\mathbf{x}, \boldsymbol{\mu} = \mathbf{x}_i), \ldots, h(\mathbf{x}, \boldsymbol{\mu} = \mathbf{x}_N)\big)^T\end{aligned}$$

To properly work with radial basis functions, we actually have to consider a normalised version of the functions. This operation is done because if, we consider all possible centres (i.e. a function

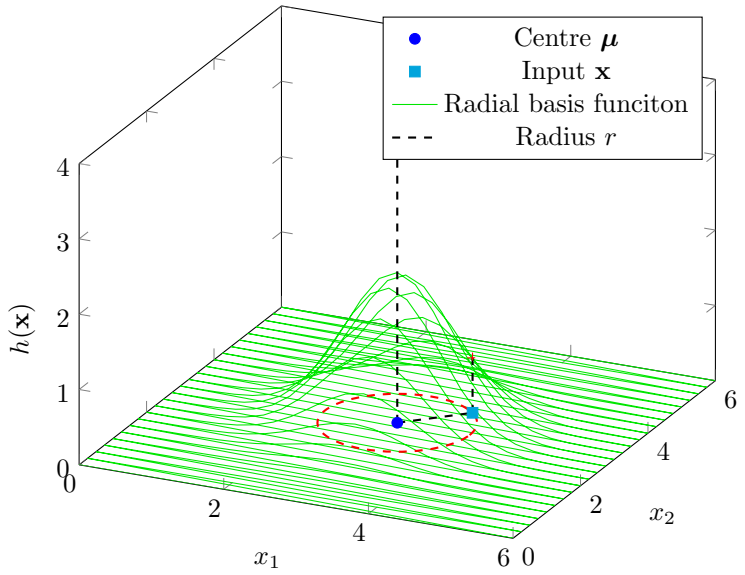


Figure 9.2: An example of a basis function. The closest the point $\mathbf{x}$ is to the centre $\boldsymbol{\mu}$, the highest the value of h.

for each point of the data-set), then in some regions of the input space all possible functions have a small value (i.e. the similarity between the points $\mathbf{x}$ in that space and the centre is low for all centres). This doesn't happen if we consider normalised radial basis functions because we impose that the sum of the function's values for all inputs $\mathbf{x}$ is constant.

$$\sum_n \phi_n(\mathbf{x}) = 1 \ \forall \mathbf{x}$$

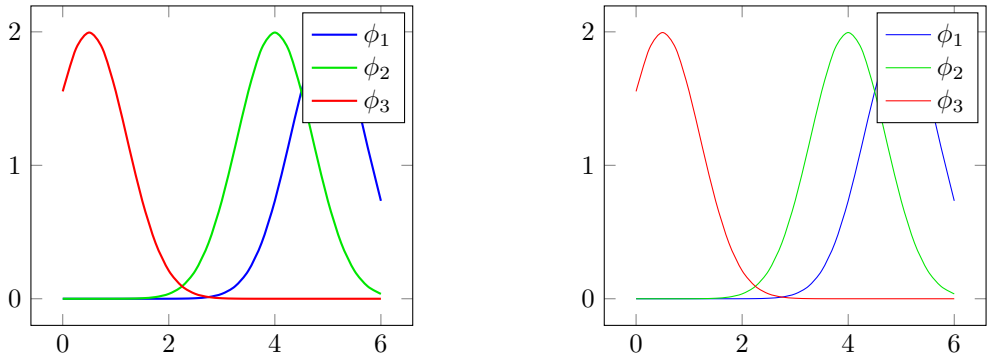
9.3.1 Nadaraya-Watson model

Radial basis functions can be used over the input/output space to model the joint probability (i.e. the joint distribution) of the input/output space. In particular, given a data-set $\{\mathbf{x}_n, t_n\}$, the joint distribution can be estimated using the Parzen window $p(\mathbf{x}, t)$

$$p(\mathbf{x}, t) = \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x} - \mathbf{x}_n, t - t_n) \tag{9.36}$$

where f is a radial basis function that considers also the target values (and not only the input). What the Parzen windows tries to do is to model the density of the input points, in fact we can use $p(\mathbf{x}, t)$ to find the expected value of t given an input point $\mathbf{x}$ (i.e. we are classifying input point). Let us analyse how this can be achieved. Let's start from what we want to reach, in particular we want to know the expected value of t given $\mathbf{x}$ (i.e., the output given a new input point $\mathbf{x}$), hence we can write

$$y(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}]$$



(a) The set of non-normalised radial basis functions. (b) The set of normalised radial basis functions.

Figure 9.3: Two graphs representing a set of three basis functions (each centred on a different input $\mathbf{x}_i$ of a data-set with three samples).

By definition, the expected value is the integral of the product between it's argument (i.e., t) and its probability (i.e., $p(t|\mathbf{x})$).

$$\begin{aligned} y(\mathbf{x}) &= \mathbb{E}[t|\mathbf{x}] \\ &= \int t \cdot p(t|\mathbf{x}) \; dt \end{aligned}$$

Thanks to the Bayes' rule (A.3) we can write the integral as the ratio between two integrals that contain only joint probabilities

$$\begin{aligned} y(\mathbf{x}) &= \mathbb{E}[t|\mathbf{x}] \\ &= \int t \cdot p(t|\mathbf{x}) \; dt \\ &= \int t \cdot \frac{p(\mathbf{x}, t)}{p(\mathbf{x})} \; dt \\ &= \frac{\int t \cdot p(\mathbf{x}, t)}{p(\mathbf{x})} \; dt \end{aligned}$$

$p(x, t)$ can finally be marginalised with respect to the target (so the integral of the joint distribution is equal to $t(x)$).

$$y(\mathbf{x}) = \frac{\int t \cdot p(\mathbf{x}, t) \; dt}{\int p(\mathbf{x}, t) \; dt}$$

Now we can replace the Parzen window (Equation 9.36) in $p(\mathbf{x}, t)$ to obtain

$$\begin{aligned} y(\mathbf{x}) &= \mathbb{E}[t|\mathbf{x}] \\ &= \frac{\int t \cdot \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x} - \mathbf{x}_n, t - t_n)\ dt}{\int \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x} - \mathbf{x}_n, t - t_n)\ dt} \\ &= \frac{\int t \cdot \sum_{n=1}^{N} f(\mathbf{x} - \mathbf{x}_n, t - t_n)\ dt}{\int \sum_{m=1}^{N} f(\mathbf{x} - \mathbf{x}_m, t - t_m)\ dt} \\ &= \frac{\sum_{n=1}^{N} \int t \cdot f(\mathbf{x} - \mathbf{x}_n, t - t_n)\ dt}{\sum_{m=1}^{N} \int f(\mathbf{x} - \mathbf{x}_m, t - t_m)\ dt} \end{aligned} \tag{9.37}$$

If we assume, for simplicity, that the component density functions have zero mean for all values of $\mathbf{x}$ so that,

$$\int_{-\infty}^{\infty} f(\mathbf{x}, t) \cdot t\ dt = 0$$

then we obtain

$$\begin{aligned} y(\mathbf{x}) &= \mathbb{E}[t|\mathbf{x}] \\ &= \frac{\sum_n g(\mathbf{x} - \mathbf{x}_n) \cdot t_n}{\sum_m g(\mathbf{x} - \mathbf{x}_m)} \\ &= \sum_n k(\mathbf{x}, \mathbf{x}_n) \cdot t_n \end{aligned} \tag{9.38}$$

Where the kernel function is

$$k(\mathbf{x}, \mathbf{x}_n) = \frac{g(\mathbf{x} - \mathbf{x}_n)}{\sum_m g(\mathbf{x} - \mathbf{x}_n)}$$

and

$$g(\mathbf{x} - \mathbf{x}_n) = \int_{-\infty}^{\infty} f(\mathbf{x}, t)\ dt$$

The function k is the kernel function, hence it measures the similarity between a point $\mathbf{x}$ and another point $\mathbf{x}_i$.

The shape of the kernel (i.e. the shape of the bell) can be modified to account for the bias-variance trade-off. In particular:

- A flat kernel increases bias.
- A narrow and high kernel increases variance.

9.4 Gaussian processes

Gaussian processes are obtained from Bayesian linear regression and are used if the number of features is too large. As always, to obtain a Gaussian process we have to start from the primal problem, i.e., the Bayesian linear regression model and then obtain the dual formulation using a kernel.

9.4.1 Bayesian linear regression

Before moving to Gaussian processes, let us revise Bayesian linear regression. Let's consider the usual linear regression model

$$y(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$$

Now let us consider a prior distribution over the the parameters $\mathbf{w}$ given by a Gaussian with mean 0 and variance α^{-1} (where α is the precision of the distribution)

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I})$$

Basically $p(\mathbf{w})$ is a vector that contains the probability distribution (which is Gaussian) of each parameter in $\mathbf{w}$. In kernel methods, we don't want to work with parameters, hence we have to **define a prior probability distribution over kernel functions directly**. Note that, the space of functions is uncountably infinite, however, for a finite training set we only need to consider the values of the function at the discrete set of input values $\mathbf{x}_i$.

Let us now call $\mathbf{y}$ the vector that contain the values $y(\mathbf{x}_i)$ for $i = 1, ..., N$ so that we can write $y(\mathbf{x})$, considering only the input points, as

$$\mathbf{y} = \Phi\mathbf{w}$$

where $\Phi_{nk} = \phi_k(\mathbf{x}_n)$. Previously, we said that the elements of $\mathbf{w}$ are distributed as a Gaussian, hence $\mathbf{y}$ is a linear combination of Gaussian distributed variables (i.e., $\mathbf{w}$) and for this reason $\mathbf{y}$ is also distributed as a Gaussian. Since $\mathbf{y}$ is a Gaussian, we have to compute its mean and covariance to define it. Let's start with the mean. The mean of each element of $\mathbf{w}$ is 0, hence also the mean of $\mathbf{y}$] is 0.

$$\mathbb{E}[\mathbf{y}] = \mathbb{E}[\Phi\mathbf{w}] = \Phi\mathbb{E}[\mathbf{w}] = \Phi \cdot \mathbf{0} = \mathbf{0}$$

Since the expected value is 0, the covariance can be computed as the expected value of the square of $\mathbf{y}$ (remember that $cov[X] = \mathbb{E}[(X - \mathbb{E}[X])(X - \mathbb{E}[X])]$)

$$\begin{aligned} cov[\mathbf{y}] &= \mathbb{E}[\mathbf{y}\mathbf{y}^T] \\ &= \mathbb{E}[(\Phi\mathbf{w})(\Phi\mathbf{w})^T] \\ &= \mathbb{E}[\Phi\mathbf{w}\mathbf{w}^T\Phi^T] \\ &= \Phi\mathbb{E}[\mathbf{w}\mathbf{w}^T]\Phi^T \end{aligned}$$

Since $\mathbf{w}$'s prior also has zero mean, $\mathbb{E}[\mathbf{w}\mathbf{w}^T]$ is the covariance matrix $\lambda\mathbf{I}$ of $\mathbf{w}$'s prior and we can write

$$\begin{aligned} cov[\mathbf{y}] &= \Phi\lambda\mathbf{I}\Phi^T \\ &= \lambda\Phi\mathbf{I}\Phi^T \\ &= \lambda\Phi\Phi^T = \frac{1}{\alpha}\Phi\Phi^T \end{aligned}$$

Where $\lambda = \frac{1}{\alpha}$ is the variance of each element of $\mathbf{w}$. We can now notice that, the matrix $\Phi\Phi^T$ is the Gram matrix K with elements

$$K_{nm} = \mathbf{k}(\mathbf{x}_n, \mathbf{x}_m) = \lambda\phi(\mathbf{x}_n)^T\phi(\mathbf{x}_m)$$

For this reason, we can rewrite the covariance of the predictor $\mathbf{y}$ as

$$cov[\mathbf{y}] = K$$

Now that we have defined expected value and covariance of the model, we can say that $\mathbf{y}$ is distributed as a Gaussian with mean $\mathbf{0}$ and covariance $K = \lambda\Phi\Phi^T$, hence the marginal distribution of $\mathbf{y}$ is

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}) \tag{9.39}$$

This allows to define a Gaussian process as follows

Definition 9.6 (Gaussian process)**.** *A Gaussian process is a probability distribution over functions of* $\mathbf{y}(\mathbf{x})$ *such that the set of values* $\mathbf{y}(\mathbf{x})$ *evaluated at an arbitrary set of points* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *jointly have a Gaussian distribution.*

Basically, we are taking the points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in our data-set and computing the joint probability distribution of $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_N)$.

Now that we have understood what a Gaussian process is, we can try to apply it to regression. To do so, we have to consider noise on the target, which means that we have to say that the prediction t_n is the sum of the model's outcome and some noise ε_n

$$t_n = y(\mathbf{x}_n) + \varepsilon_n$$

At this point we can consider a Gaussian probability distribution of t_n, given $y(\mathbf{x}_n)$ centred in $y(\mathbf{x}_n)$ and with covariance $\sigma^2 = \beta^{-1}$ (where β represents the precision of the noise).

$$p(t_n|y(\mathbf{x}_n)) = \mathcal{N}(t_n|y(\mathbf{x}_n), \beta^{-1})$$

This makes sense because t_n is centred in $y(\mathbf{x}_n)$ (i.e. it has the highest probability of being there) and then the Gaussian curve defines a lower probability, going towards the tails, that models the noise ε_n.

Since the noise is independent for each sample (i.e. for each $\{\mathbf{x}_i, t_i\}$) and $y(\mathbf{x}_n)$ is distributed as a Gaussian, then also the joint distribution on all data samples is distributed as a Gaussian

$$p(\mathbf{t}|\mathbf{y}) = \mathcal{N}(\mathbf{t}|\mathbf{y}, \sigma^2\mathbf{I}_{\mathbf{N}\times\mathbf{N}}))$$

From the definition of Gaussian process, we get that the probability distribution of $\mathbf{y}$ is

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K})$$

The probability distribution $p(\mathbf{y})$ can be used to compute the distribution of $p(\mathbf{t})$ as

$$p(\mathbf{t}) = \int p(\mathbf{t}|\mathbf{y})p(\mathbf{y})d\mathbf{y} = \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C})$$

where

$$\mathbf{C}_{nm} = C(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \sigma^2\delta_{nm}$$

and σ^2 is the irreducible noise and δ_{nm} is the covariance of the model $\mathbf{y}$. A widely used kernel for this model is the following

$$k(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 \exp\left(-\frac{\theta_1}{2}\|\mathbf{x}_n - \mathbf{x}_m\|_2^2\right) + \theta_2 + \theta_3\mathbf{x}_n^T\mathbf{x}_m$$

9.4.2 Predictions

Given the Gaussian regression model, we want to predict the value of t_{N+1} on a new sample $\mathbf{x}_{N+1}$. To do so, we need to compute the probability distribution of t_{N+1} given the training target values $\mathbf{t}_N = \{t_1, \dots, t_N\}$ and the training data samples $\mathbf{x}_1, \dots, \mathbf{x}_N$

$$p(t_n|\mathbf{t}_N, \mathbf{x}_1, \dots, \mathbf{x}_N)$$

From the model we built, we know that this probability distribution is ditributed as a Gaussian

$$p(t) = \mathcal{N}(t|\mathbf{0}, \mathbf{C}_{N+1}) \tag{9.40}$$

of mean $\mathbf{0}$ and covariance $\mathbf{C}_{N+1}$ where

$$\mathbf{C}_{N+1} = \begin{bmatrix} \mathbf{C}_N & \mathbf{k} \\ \mathbf{k}^T & c \end{bmatrix}$$

and

$$c = k(\mathbf{x}_{N+1}, \mathbf{x}_{N+1}) + \sigma^2$$

Basically the matrix $\mathbf{C}_{N+1}$ is the matrix $\mathbf{C}_N$ to which we add a column $\mathbf{k}$ and a row $\mathbf{k}$, which are the kernel functions evaluated between the new point $\mathbf{x}_{N+1}$ and each point $\mathbf{x}_i$. In other words, the element in position i of $\mathbf{k}$ is $k_i = k(\mathbf{x}_i, \mathbf{x}_{N+1})$. Now, using the same reasoning used for Bayesian regression, we can finally compute the mean m and covariance σ^2 of t_{N+1} as

- $m = \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{t}$
- $\sigma^2 = c - \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{k}$

Notice that, $\mathbf{C}$ has to be **positive definite** if and only if the kernel function is positive semi-definite.

Computational cost

Let us analyse how expensive this model is when we need to predict the target of a new input point $\mathbf{x}_{N+1}$. Let us analyse the costs of each operation needed to compute m and σ^2,

- The inversion of the matrix $\mathbf{C}_N$ has a cubic cost $\mathcal{O}(N^3)$.
- The computation of the mean and the variance, once we have computed $\mathbf{C}_N^{-1}$, has a quadratic cost because we have to compute the product between a vector and a matrix.

However, we can notice that the matrix $\mathbf{C}_N$ doesn't change for each prediction, hence it can be inverted once and then it can be used for all new data points that have to be classified. This means that, let apart the matrix inversion, the cost of the prediction is dominated by the vector-matrix multiplication.

If we compare the computational cost of Gaussian regression with Bayesian regression we notice that in the latter case the cost of inverting the matrix and doing the multiplication is $\mathcal{O}(M^3)$ and $\mathcal{O}(M^2)$ because we have to consider the basis functions (i.e. the features of the model). This means that,

- If the number of features M is smaller than the number of data points N, then Bayesian regression is more convenient.

- If the number of features M is bigger than the number of data points N, then Gaussian regression with kernels is more convenient.

Having a lot of samples can usually improve the model, however in this case (differently from non-kernel methods) it also worsen performance a lot. If we want to use Gaussian regression with large data sets, without slowing the training phase too much, we can

- Apply **random sampling**. Random sampling implies taking samples at random from the training space.
- Apply **clustering**. Clustering separates samples in regions (i.e. clusters the training points in regions where they are close one to the other) and then only one sample per cluster is taken for training.

9.4.3 Kernel

In Gaussian processes we usually use a variation of the Gaussian kernel.

$$k(\mathbf{x}_i, \mathbf{x}_j) = \phi \exp\left(- \frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2}{2l^2} \right) \tag{9.41}$$

This kernel has two parameters l and ϕ that have to be set depending on our knowledge of the problem. Moreover, we have to set the variance σ^2 of the noise. In particular,

- The **lengthscale** l controls the smoothness of the Gaussian process. The biggest the value of l, the smoother the process is.
- The variance σ^2 controls how confident we are about the prediction. The biggest the variance, the less confident we are.

9.4.4 Uncertainty

To analyse how Gaussian processes can capture the uncertainty over a prediction let us consider Figure 9.4. The graph shows, for each possible input $\mathbf{x} = x$ (we are considering a simple problem with only one feature) the distribution of the prediction t. Each prediction for each input x is distributed as a Gaussian with a certain mean and variance. This means that a Gaussian process is an infinite sequence of Gaussian distributions. If we plot all the mean we obtain the prediction, which is shown in blue in Figure 9.4. The variance of each Gaussian is shown in green and, as we can see, the variance is higher where we have less points, since we are less certain about the prediction. Visually, where we have few data points, the green lines are further from the prediction (i.e. the mean of the Gaussians), while where there are a lot of points the margin is tighter. Moreover, where we are certain about the prediction, i.e., where we have a lot of samples, the Gaussians are narrower and higher while in less dense zones, the Gaussians are looser and lower.

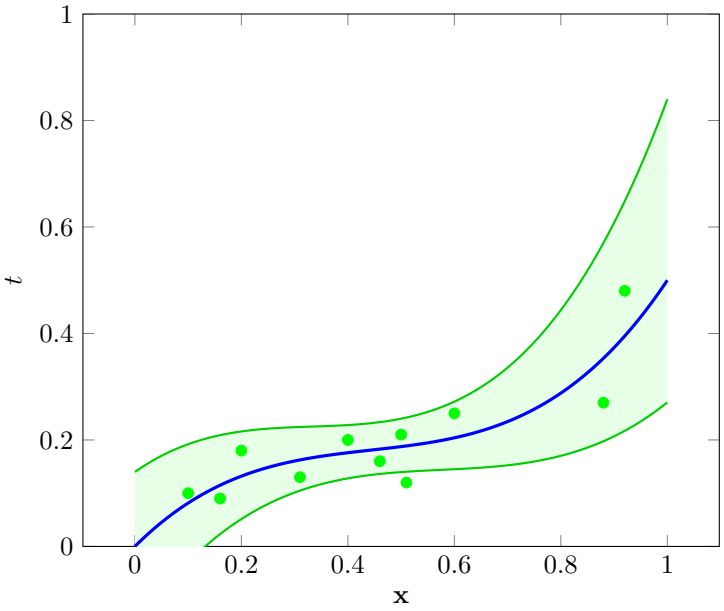


Figure 9.4: A Gaussian Process' model.

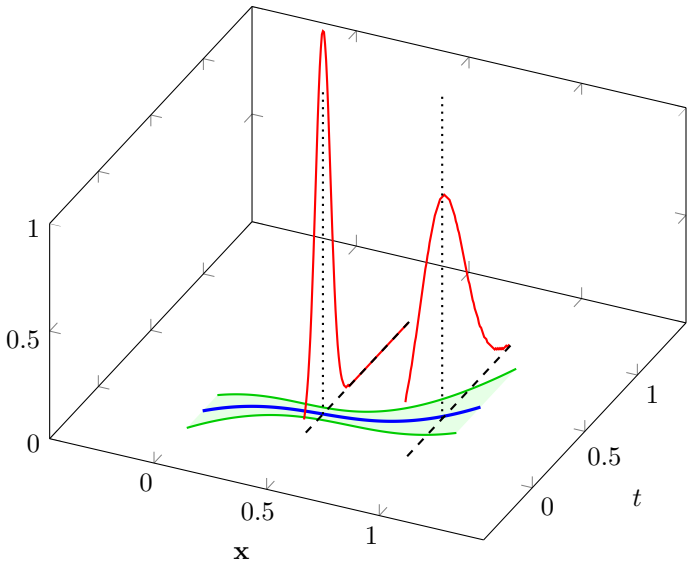


Figure 9.5: A Gaussian Process' model.

Chapter 10

Support Vector Machines

10.1 Support Vector Machines

Support Vector Machines are a class of classification techniques that use kernels. A Support Vector Machine is made of

- A **subset of training samples** $\mathcal{S} = \mathbf{x}_i$, called **support vectors**. This means that we need only a subset of the training set to classify a new point.
- A **vector of weights a** for the support vectors, each element $a_i \in \mathbf{a}$ is associated to a support vector $\mathbf{x}_i \in \mathcal{S}$.
- A similarity function $K(x, x')$, namely a **kernel**.

The class prediction (i.e. the classification) of a new point x_q, considering $t_i \in \{-1, 1\}$, is given by the function

$$f(x_q) = \text{sign}\left(\sum_{m \in \mathcal{S}} a_m t_m k(x_q, x_m) + b \right) \tag{10.1}$$

where

- $\mathcal{S}$ is the set of indices of the support vectors.
- $\sum_{m \in \mathcal{S}} a_m t_m k(x_q, x_m)$ is the linear combination, for all support vectors, of the support target values, multiplied by the weight a and the similarity between x_q and the support vector.

Classification methods define a border, or a threshold, between two classes. Support Vector Machines can achieve this goal using only a subset of input samples, called **support vectors** (which contains support points). The threshold that divides classes is placed in the middle between the support vectors of a class and the support vectors of the other class. The distance between the closest support vectors and the threshold, which is measured using the kernel function, is called margin. A graphical example is shown in Figure 10.1.

Based on the description of a Support Vector Machine, we understand that we have to

- **Choose the kernel**. The choice of kernel depends on the knowledge of the problem.
- **Choose the weights**. To choose the weights we have to maximise the margin.

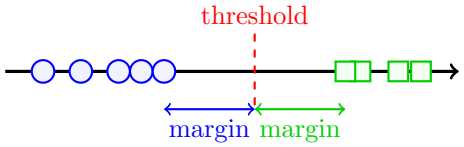


Figure 10.1: An example of SVM classification in one dimension (i.e., each support vector is simply a point).

- **Choose the subset of samples to use for classification**. This choice is a side effect of choosing the weights because, after training, some weights a_i are 0, hence we can discard the samples whose associated weight is 0. In other words, the samples used for classification are the points with a non-null weight.

10.1.1 Support Vector Machines from pereceptrons

Support Vector Machines are non-parametric methods but can be seen as a generalisation of perceptrons, which are parametric methods. In particular, we can try and write the model used for perceptron (4.3).

$$f(\mathbf{x}_q) = \text{sign}\left[\sum_{j=1}^{M} w_j \phi_j(\mathbf{x}_q)\right] \tag{10.2}$$

If we compute the weights w_j as (the formula will be obtained later on)

$$w_j = \sum_{n=1}^{N} a_n t_n \phi_j(\mathbf{x}_n) \tag{10.3}$$

we can replace the weights in the perceptron model, and obtain

$$f(\mathbf{x}_q) = \text{sign}\left[\sum_{j=1}^{M} \left(\sum_{n=1}^{N} a_n t_n \phi_j(\mathbf{x}_n)\right) \phi_j(\mathbf{x}_q)\right] \tag{10.4}$$

$$= \text{sign}\left[\sum_{n=1}^{N} a_n t_n \big(\boldsymbol{\phi}_j(\mathbf{x}_n) \boldsymbol{\phi}_j(\mathbf{x}_q)\big)\right] \tag{10.5}$$

$$= \text{sign}\left[\sum_{n=1}^{N} a_n t_n k\big(\mathbf{x}_n, \mathbf{x}_q\big)\right] \tag{10.6}$$

10.2 Margin

If we consider a two-dimensional space (or even an higher dimension space), multiple lines (or planes) can divide two classes. Among all the possible choices we want to find the one that **maximises the margin**, namely, that maximises the minimum distance between a point in the class and the threshold hyperplane. This choice is done to consider noise.

Let us consider, for instance, Figure 10.2. Say we consider the red dotted threshold, and we want to classify the black point. In this case, the SVM assigns the point to C_2 (green squared points), even if the point is much closer to the points of class C_1 (blue, rounded points). The same point, classified using the orange, dashed threshold would be classified more properly.

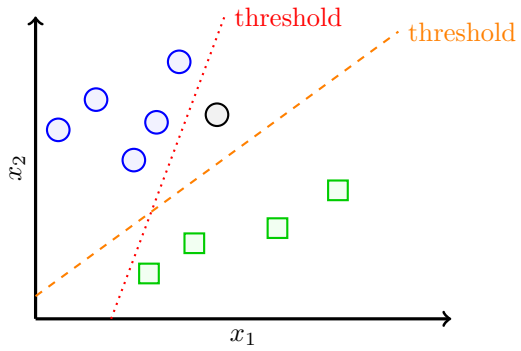


Figure 10.2: An example of SVM classification in two dimensions.

Margin maximisation

To maximise the margin, first we have to formally define it. Assuming that $\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x})$ is a plane that correctly classifies every point in the data-set, we write the margin as

$$m = \min_n t_n(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x_n}) + b) \tag{10.7}$$

In practice,

1. For each point $\mathbf{x}_i$ in the data set, we compute the unnormalised distance $\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)$ between $\mathbf{x}_i$ and the threshold plane. Remember that the distance of a point $\mathbf{x} = (a, b)$ from a plane $\alpha x_1 + \beta x_2 = k$ is $\frac{|\alpha \cdot a + \beta \cdot b + k|}{\sqrt{a^2+b^2}}$. If we don't consider the modulo and the normalisation we get that the distance is computed as $\alpha \cdot a + \beta \cdot b + k$ which is in fact $\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n)$ (being $\alpha, \beta, \ldots$ the parameters w_i and $a, b, \ldots$ the inputs).

2. For each point $\mathbf{x}_i$ in the data set, we multiply the distance computed at point 1, with the target t_i. Note that the term $t_n(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x_n}) + b)$ is always positive, since if $t_i = -1$, the distance is negative, and if $t_i = 1$, the distance is positive. In both cases, the product between t_i and the distance is positive.

3. Take the minimum between the values computed at point 2. Basically, we take the minimum distance to the threshold.

The maximum margin can be found as

$$\mathbf{w}^* = \arg\max_{\mathbf{w},b} \left(\frac{1}{\|\mathbf{w}\|_2} \min_n t_n(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x_n}) + b) \right) \tag{10.8}$$

where $\mathbf{w}^*$ considers both $\mathbf{w}$ and b (i.e., $\mathbf{w}^* = (\mathbf{w}^T, b)^T$). Solving directly Equation 10.8 is complex and hard, hence we should consider an equivalent problem which is easier to solve. Note that dividing the margin by $\|\mathbf{w}\|_2$ we are normalising the margin (remember the definition of distance we gave before). In particular, this problem has infinite solutions since we can scale $\mathbf{w}$ by a constant and the solution doesn't change. Basically, if we consider the planes $w_0 + w_1x_1 + w_2x_2 = 0$ and $c \cdot w_0 + c \cdot w_1x_1 + c \cdot w_2x_2 = 0$, we get the same plane, however, if we consider the definition of margin (which is not normalised) we get different values for the margin. Nonetheless, all the different margins refer to the same plane since we have scaled that plane by a constant c. This means that,

the solution (i.e., the threshold plane) is associated to an infinite number of margins, but to one normalised Euclidean distance. Therefore we can chose an arbitrary margin. For simplicity let us select 1 as margin. Note that, we are putting the non-normalised Euclidean distance to 1, not the normalised one. In other words, we aren't changing the distance itself but only the scale of $\mathbf{w}$. Since we have fixed the value of the margin to one we obtain

$$\mathbf{w}^* = \arg\max_{\mathbf{w},b} \left(\frac{1}{\|\mathbf{w}\|_2} \right) \tag{10.9}$$

This very problem can be seen as a minimisation of $\|\mathbf{w}\|_2$, since when x is maximum, $\frac{1}{x}$ is minimum (for positive values of x, which in fact $\|\mathbf{w}\|_2$ being a sum of squares).

$$\mathbf{w}^* = \arg\min_{\mathbf{w},b} \left(\|\mathbf{w}\|_2 \right) \tag{10.10}$$

More formally, we can write

$$\textbf{Minimise } \frac{1}{2}\|\mathbf{w}\|_2^2 \tag{10.11}$$

$$\textbf{Subject to } t_n(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x_n}) + b) \geq 1 \quad \forall n \tag{10.12}$$

Note that the problem we are trying to solve has become a minimisation problem instead of a maximisation. This is because we are considering $\|\mathbf{w}\|_2^2$ instead of $\frac{1}{\|\mathbf{w}\|_2}$. Also note that we had to add a constraint to the problem. In particular, $t_n(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x_n}) + b)$ has to be bigger than 1 because we imposed that the margin (i.e., the minimum of $t_n(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x_n}) + b)$) is 1 (hence a general distance from a point $t_n(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x_n}) + b)$ has to be bigger than the minimum distance).

The problem we are trying to solve is a quadratic programming problem, whose general form is

$$\textbf{Minimise } f(\mathbf{w}) \tag{10.13}$$

$$\textbf{Subject to } h_i(\mathbf{w}) \leq 0 \quad i = 1, 2, \dots \tag{10.14}$$

$$g_i(\mathbf{w}) = 0 \quad i = 1, 2, \dots \tag{10.15}$$

To solve this class of problems, we can use the Lagrangian function

$$L(\mathbf{w}, \boldsymbol{\lambda}) = f(\mathbf{w}) + \sum_i \lambda_i h_i(\mathbf{w}) \tag{10.16}$$

where $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \dots)$ and each λ_i is called Lagrangian multiplier. Our optimisation problem can therefor be written as

$$\textbf{Minimise } \frac{1}{2}\|\mathbf{w}\|_2^2 \tag{10.17}$$

$$\textbf{Subject to } -(t_n(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x_n}) + b) - 1) \leq 0 \quad i = 1, 2, \dots \tag{10.18}$$

$$h_i(\mathbf{w}) = 0 \quad i = 1, 2, \dots \tag{10.19}$$

The solution is obtained computing the gradient $\nabla L(\mathbf{w}^*, \boldsymbol{\lambda}^*)$ and putting it to 0. The Lagrangian function can't be directly applied to our problem since our constraints are inequalities.

In our case, the Lagrangian function can be written as

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N} a_n \big(t_n(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) + b) - 1 \big) \tag{10.20}$$

Now we can compute the gradient with respect to $\mathbf{w}$ and b.

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\mathbf{x}_n) \tag{10.21}$$

$$\frac{\partial L}{\partial b} = -\sum_{n=1}^{N} a_n t_n \tag{10.22}$$

Now, if we put the gradient to 0, we get

$$\mathbf{w} = \sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\mathbf{x}_n) \tag{10.23}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{10.24}$$

Now that we have sorted out the optimal values for $\mathbf{w}$ and b, we can replace them in the Lagrangian function (10.20) to obtain the dual solution (i.e., the non-parametric kernel solution). In particular, we obtain a new maximisation problem

$$\textbf{Maximise } L(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{M} \sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x_m}) \tag{10.25}$$

$$\textbf{Subject to } a_n \geq 0 \tag{10.26}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{10.27}$$

where the Lagrangian function to maximise has been obtained as follows

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N} a_n\big(t_n(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n) + b) - 1\big) \tag{10.28}$$

$$= \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N} a_n t_n \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n) - \sum_{n=1}^{N} a_n t_n b + \sum_{n=1}^{N} a_n \tag{10.29}$$

$$= \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N} a_n t_n \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.30}$$

$$= \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N} a_n t_n \left[\sum_{m=1}^{N} a_m t_m \boldsymbol{\phi}(\mathbf{x}_m)\right]^T \boldsymbol{\phi}(\mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.31}$$

$$= \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N} a_n t_n \left[\sum_{m=1}^{N} a_m t_m \boldsymbol{\phi}(\mathbf{x}_m)^T\right] \boldsymbol{\phi}(\mathbf{x}) + \sum_{n=1}^{N} a_n \tag{10.32}$$

$$= \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m \boldsymbol{\phi}(\mathbf{x}_m)^T\boldsymbol{\phi}(\mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.33}$$

$$= \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m k(\mathbf{x}_m, \mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.34}$$

$$= \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m k(\mathbf{x}_m, \mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.35}$$

$$= \frac{1}{2}\sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\mathbf{x}_n) \sum_{m=1}^{N} a_m t_m \boldsymbol{\phi}(\mathbf{x}_m) - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m k(\mathbf{x}_m, \mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.36}$$

$$= \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_m t_m a_n t_n \boldsymbol{\phi}(\mathbf{x}_n)^T\boldsymbol{\phi}(\mathbf{x}_m) - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m k(\mathbf{x}_m, \mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.37}$$

$$= \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_m t_m a_n t_n k(\mathbf{x}_n, \mathbf{x}_m) - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m k(\mathbf{x}_m, \mathbf{x}_n) + \sum_{n=1}^{N} a_n \tag{10.38}$$

$$= \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n t_n a_m t_m k(\mathbf{x}_m, \mathbf{x}_n) \tag{10.39}$$

where $\sum_{n=1}^{N} a_n t_n b = 0$ because we imposed $\sum_{n=1}^{N} a_n t_n = 0$.

10.2.1 Prediction

The model

$$f(\mathbf{x}) = \text{sign}\left(\sum_{m\in\mathcal{S}} a_m t_m k(\mathbf{x}_q, \mathbf{x}_m) + b\right)$$

can be used to classify a point $\mathbf{x}$. In this model we still miss one piece, in fact we haven't still defined b. This parameter can be estimated from the data-set as

$$b = \frac{1}{N_S}\sum_{n\in\mathcal{S}}\left(t_n - \sum_{m\in\mathcal{S}} a_m t_m k(\mathbf{x}, \mathbf{x}_n)\right) \tag{10.40}$$

where $N_{\mathcal{S}}$ is the number of support vector (which is much smaller than the total number of samples).

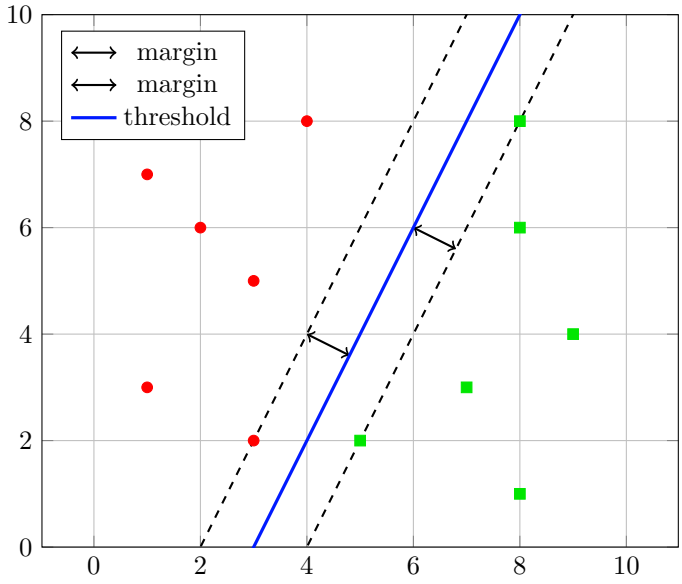


Figure 10.3: A visual representation if the threshold obtained using a Support Vector Machine

10.2.2 Dimensions

If the number of dimensions of the input vectors is high then we need more support vectors to classify new input samples. However, if we have a lot of support vectors, the model is more sensible to changes, because it depends on many points. This means that the model is overfitting.

10.3 Noisy data

In some cases, the inputs in the data-set are very noisy and some points fall in a region so that it's impossible to draw a linear boundary. Consider for instance Figure 10.4.

To solve this issue, we can use slack variables ξ_i, one for each input sample that allow to penalise some constraints. In particular, we allow constraints to be violated but we add a penalty for the constraints that are violated. In other words, we allow some points to be misclassified. Introducing misclassified points allows us to introduce two types of margins:

- **Hard margin**. An hard margin doesn't allow points to be misclassified.
- **Soft margin**. A soft margin allows points to be misclassified.

This means that the new optimisation problem becomes

$$\textbf{Minimise } \|\mathbf{w}\|_2^2 + C\sum_i \xi_i \tag{10.41}$$

$$\textbf{Subject to } t_i(\mathbf{w}^T x_i + b) \geq 1 - \xi_i \tag{10.42}$$

$$\xi_i \geq 0 \tag{10.43}$$

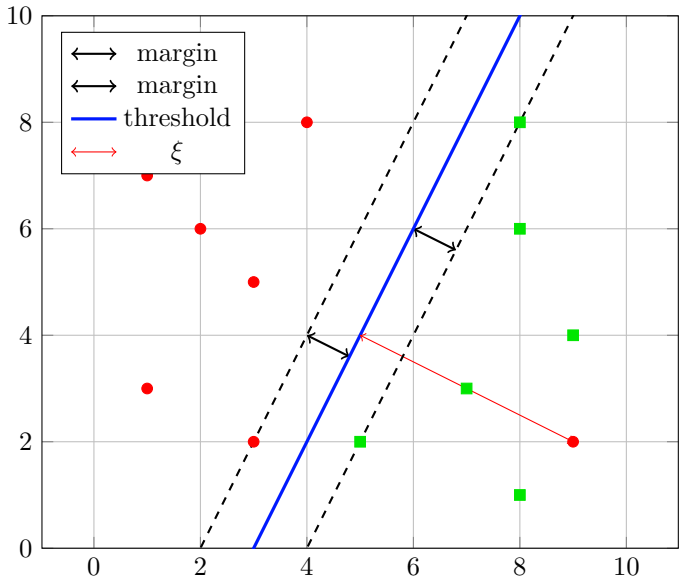


Figure 10.4: A visual representation of a noisy data-set where a point is misclassified.

where C is a coefficient that allows to handle the bias-variance trade-off. In particular,

- A small C leads to models with a big bias (i.e., to underfitting). Moreover, models with a small C has looser margins since we aren't penalising misclassified points that much.
- A big C leads to models with a big variance (i.e., to overfitting). A model with a big C has tighter margins since we penalise a lot misclassified points.

The coefficient C can be chosen using cross validation. As we can see from the new problem, slack variables:

- Are always positive.
- Also appear in the objective function, since it's a price we have to pay to handle noise.
- Allow to have a margin smaller than 1.

Adding slack variables doesn't change the kernel representation that much, in fact the only difference with the initial model is in the constraints.

$$\textbf{Maximise } L(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{M} \sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x_m}) \tag{10.44}$$

$$\textbf{Subject to } 0 \leq a_n \leq C \tag{10.45}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{10.46}$$

In particular:

- If $a_n < C$, the input to classify lies on the margin.
- If $a_n = C$, the input to classify lies inside the margin and is:
  - Correctly classified if $\xi_i \leq 1$.
  - Misclassified if $\xi_i > 1$.

Part III

Reinforcement learning

Chapter 11

Markov Decision Processes

11.1 Introduction

The goal of reinforcement learning is to select the actions that allow to maximise the cumulative reward obtained from these actions. Basically, the learner should sacrifice immediate reward to collect potentially more in the future. This happens because the decisions that the agent takes at time t may affect the reward in the future. In other words, the reward of an action at time t may be delayed.

11.1.1 Agent-environment interface

When we consider a reinforcement learning problem, we can model the problem using the agent-environment model. In particular the model is made of:

- An **agent** which is the learner that tries to maximise its future reward.
- The **environment** the agent interacts with.

The interaction between agent and environment happens as follows. At each time step t:

1. The agent executes an action a_t on the environment.
2. The agent observes the changes o_t on the environment.
3. The agent receives a reward r_t from the environment.

These components of an interaction are stored in an history h.

Definition 11.1 (History)**.** *The history h_t is a sequence of actions, observations and rewards obtained from time 1 to time t.*

$$h_t = a_1, o_1, r_1, \ldots, a_t, o_t, r_t \tag{11.1}$$

The history is used to choose what to do at time $t+1$. The history is also used to define a state s_t which is, formally, a function of the history.

$$s_t = f(h_t) \tag{11.2}$$

Note that, from this definition we understand that a state might differ from the history, in fact the state might contain more information (or not enough information) than needed for selecting the next action. The shape of the function f depends on the agent.

Environment state

The environment has a state s_t^e, however we might not be able to observe it. In particular, the agent has a representation of the state of the environment which is based on the observations o_i and could be different (or misaligned) from the actual state of the environment. The private state of the environment is used by the environment to produce the observation and the reward when an agent performs an action.

Agent state

The agent has a state s_t^a, which is the information used by the reinforcement learning agent to select the next action. The state of the agent can be any function of the history.

$$s_t^a = f(h_t) \tag{11.3}$$

Fully observable environments

In fully observable environments, the agent directly observes the state of the environment. This means that the state of the agent s_t^a coincides with the real state of the environment s_t^e.

$$s_t^a = s_t^e \tag{11.4}$$

11.1.2 Usages of reinforcement learning

Reinforcement learning is very useful when

- We don't know how the environment reacts to the actions that we perform.
- It's hard to model the environment.
- The model of the environment is known and we know how it reacts to inputs, but it's hard to solve it exactly (e.g., it has too many degrees of freedom or too many variables). In such cases, reinforcement learning approximates the solution.

11.1.3 Model definition

Before solving a reinforcement learning problem, we have to classify it. In particular, a problem can:

- Have **discrete** or **continuous states**.
- Have **discrete** or **continuous actions**.
- Be **deterministic** or **stochastic**.
- Be **fully** or **partially observable**.
- Be **single-agent** or **multi-agent**.

11.2 Markov Decision Process

11.2.1 Markov assumption

Markov Decision Processes are based on the Markov assumption.

Definition 11.2 (Markov assumption)**.** *A stochastic process X_t is said to be Markovian if and only if its value at time $t+1$ depends only on the observation at time t.*

$$\mathbb{P}(X_{t+1} = j | X_t = i, X_{t-1} = k_{t-1}, \dots, X_1 = k_1, X_0 = k_0) = \mathbb{P}(X_{t+1} = j | X_t = i) \tag{11.5}$$

Less formally, we can say that a process is Markovian if the future is independent of the past, given the present. This means that, to predict the future we need only the information about the present.

Another important thing to remember is that the state of the agent is build using the history and, once the state is known (i.e., the agent has learned the state), the history can be discarded and the state is enough to predict the future.

Moreover, if the model is stationary, the prediction doesn't even depend on the time at which it's done.

$$\mathbb{P}(X_{t+1} = j | X_t = i) = \mathbb{P}(X_1 = j | X_0 = i) \tag{11.6}$$

11.2.2 Discrete-time finite Markov Decision Process

A Markov Decision Process (MDP) is a Markov reward process with decisions. A MDP models an environment in which all states are Markovian and time is divided into stages. More formally:

Definition 11.3 (Markov Decision Process)**.** *A Markov Decision Process (MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$, where*

- *$\mathcal{S}$ is a finite set of states.*
- *$\mathcal{A}$ is a finite set of actions.*
- *$P(s'|s,a)$ is a state transition probability matrix. Basically, P says what is the probability that the agent goes in state $s' \in \mathcal{S}$, after performing given an action $a \in \mathcal{A}$ on state $s \in \mathcal{S}$.*
- *$R(s,a) = \mathbb{E}[r|s,a]$ is a reward function that returns the expected value of the reward obtained when performing action s from state s.*
- *$\gamma \in [0,1]$ is the discount factor. It important to remember that the discount factor is not a parameter of the learning algorithm but of the problem itself.*
- *μ is the set of probabilities that, from each state s, s is an initial state.*

A MDP can be represented as a finite state machine in which each action is a transition from a state to another. Each transition a is also associated to a reward, which is the reward obtained after performing action a.

11.3 Rewards and goals

We have understood that agents try to maximise the accumulation of rewards. Now, we have to understand if rewards can also be used to model the notion of goal. In particular, we want to understand if a scalar reward (i.e., a number) is good enough to model the objective of an agent (e.g., win a game of chess). The Sutton hypothesis tries to give an answer to this question.

Theorem 11.1 (Sutton hypothesis)**.** *All of what we mean by goals and purposes can be well thought as the maximisation of the cumulative sum of a received scalar signal (i.e., a reward).*

Long story short, the Sutton answer to our question is positive. The Sutton hypothesis isn't however fully correct, in fact in some cases the reward isn't enough to model a goal. Still, if we are interested in Markovian processes the Sutton hypothesis is correct and we can say that rewards are good for modelling a goal.

Now that we know that, at least for our purposes, rewards are good for modelling goals, let us list some of their properties.

- **The same goal can be specified using an infinite number of reward functions**. For instance, we can scale the reward function by a constant factor k, thus obtaining an infinite number of functions.
- **A goal must be outside the control of the agent**. The agent should only be able to read the reward.
- **The agent must be able to measure the success** (i.e., the fact that it has reached a goal) explicitly and frequently.

Let us focus a bit more on the first property. Since there are a lot of possible reward functions, we have to find the ones that are precise enough for the learner to actually learn. In particular we should

- Design dense reward functions.
- Suggest what actions are better or worst.
- Give a reward not only at the end (i.e., when the goal is reached).
- Be able to understand what actions are responsible for a positive cumulative reward at every time instant.

11.3.1 Time horizon

Reinforcement learning agents learn thanks to the cumulative rewards. Since rewards are accumulated over time, we should ask ourselves what is the time horizon in which rewards are accumulated. In particular, we can identify:

- **Finite time horizons**. The agent has to do a fixed finite number of steps before reaching its goal. In this context, the solution depends on time. In other words, the solution is not stationary since an action at a certain time depends on when we take that action.
- **Indefinite time horizons**. The agent will eventually reach its goal, however we don't know after how many time instants.

- **Infinite time horizons**. The interaction between agent and environment goes on forever.

11.3.2 Cumulative reward

As for now, we have said that reinforcement learning is based on the cumulative reward, however we haven't described how it's computed. Some examples of cumulative rewards V are

- **Total reward**. The total reward is the sum of all the rewards obtained.

$$V = \sum_{i=1}^{\infty} r_i \tag{11.7}$$

  The total reward works fine for finite and indefinite time horizons, however it doesn't work for infinite time horizons, since we would get an infinite reward.

- **Average reward**. The average reward is the average of all the rewards obtained. Since we want to consider also infinite time horizons, we have to consider the limit of the average for $n \to \infty$.

$$V = \lim_{n\to\infty} \frac{\sum_i r_i}{n} \tag{11.8}$$

- **Discounted reward**. The discounted reward is a total reward in which we give less importance to future rewards using a discount factor $\gamma \in [0, 1]$.

$$V = \sum_{i=1}^{N} \gamma^{i-1} r_i \tag{11.9}$$

- **Mean variance reward**.

Discounted rewards

Among all ways we can compute cumulative rewards, discounted rewards are very important, thus let us analyse them a more in detail. Let us start by giving a more formal definition of cumulative reward.

Definition 11.4 (Discounted reward)**.** *A return v_k is the total discounted reward from time-step t*

$$v_t = \sum_{i=0}^{\infty} \gamma^k r_{t+k+1} \tag{11.10}$$

where

- $\gamma \in [0, 1)$ *is the discount factor.*
- r_{t+k+1} *is the reward obtained in the future.*

Note how, from Definition 11.4, the return is computed using future rewards and not past rewards. Let us now analyse a little more in depth the discount factor. In particular we can say that:

- The discount factor, being a number smaller than 1, gives less weight to future rewards.

- A discount factor close to 0 penalises a lot rewards in the far future. We can say that that the evaluation of the return is myopic.
- A discount factor close to 1 given more or less the same weight to all rewards. In this case we say that the evaluation of the return is far-sighted.
- The discount factor can be interpreted as the probability that the process will go on. In fact, the higher is the discount, the more confident we are that the process will go on because we give importance also to rewards in the far future.

Note that for now we have considered a constant reward, however, we can even use time dependent discounts that change in time. In this case, the problem to solve is harder, for this reason we will use constant discounts. Since we are going to use the return function, it's important to prove that it converges for $n \to \infty$ (i.e., that we can use it for infinite time horizons). Let us start from the definition of return (11.4).

$$v_t = \sum_{i=0}^{\infty} \gamma^k r_{t+k+1} \tag{11.11}$$

Now we can compute an upper bound for v_t. In particular, if we consider the maximum reward the agent can receive, and we call it R_{max} we can write

$$v_t = \sum_{i=0}^{\infty} \gamma^k r_{t+k+1} \tag{11.12}$$

$$\leq \sum_{i=0}^{\infty} \gamma^k R_{max} \tag{11.13}$$

$$\leq R_{max} \sum_{i=0}^{\infty} \gamma^k \tag{11.14}$$

The series in Equation 11.14 is a geometric series which converges to $\frac{1}{1-\gamma}$ if $\gamma < 1$. Since we have stated that γ has to be taken in the interval $[0, 1)$, we are sure that the series converges, hence the return sums up to a finite value, even for an infinite time horizon. Note that the approximation obtained in Equation 11.14 can also be used for finite and indefinite time horizons, if the number of time-steps is large enough.

11.4 Policies

Policies are a fundamental concept in reinforcement learning.

Definition 11.5 (Policy)**.** *A policy π, at any given point in time, decides what action an agent should perform.*

In other words, a policy defines the behavior of an agent. Policies can be

- **Markovian** or **history-dependent**. A Markovian policy uses only the current state of the agent, whilst an history-dependent policy uses history. The former is a subset of the latter.

- **Deterministic** or **stochastic**. A deterministic policy, given a state, always chooses the same action. On the other hand, a stochastic policy, given a state, chooses an action with some probability distribution. The former is a subset of the latter.
- **Stationary** or **non-stationary**. A stationary policy doesn't depend on time (i.e., its choices doesn't depend on time). On the other hand, a non-stationary policy depends on time. The former is a subset of the latter.

An important property to remember is that *every Markov Decision Problem can be always solved with a Markovian, deterministic and stationary policy.* This means that it always exists at least one optimal policy which is Markovian, deterministic and stationary. That being said, we will use stationary stochastic Markovian policies.

11.4.1 Stationary stochastic Markovian policies

Among all types of policies we will consider stationary stochastic Markovian policies.

Definition 11.6 (Stationary stochastic Markovian policy)**.** *Stationary stochastic Markovian policy π is a distribution over actions, given a state s*

$$\pi(a|s) = \mathbb{P}[a|s] \tag{11.15}$$

In other words, the policy $\pi(a|s)$ returns the probability of taking action a given that we are in state s.

Markov reward process

If we combine state transitions (defined by P) and the policy π we obtain a matrix P^π of dimensions $|\mathcal{S}| \times |\mathcal{S}|$ so that, each cell $\langle i, j \langle$ contains the probability of reaching state s_j from state s_i with one step using policy π. The matrix P^π is called **transition kernel** of the agent. The transition kernel can be computed as

$$P^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) P(s'|s, a) \tag{11.16}$$

The transaction matrix is therefor

$$P^\pi = \begin{bmatrix} \sum_{a \in \mathcal{A}} \pi(a|s_1)\mathbb{P}(s_1|s_1, a) & \dots & \sum_{a \in \mathcal{A}} \pi(a|s_{|\mathcal{S}|})\mathbb{P}(s_{|\mathcal{S}|}|s_1, a) \\ \vdots & \ddots & \vdots \\ \sum_{a \in \mathcal{A}} \pi(a|s_{|\mathcal{S}|})\mathbb{P}(s_{|\mathcal{S}|}|s_1, a) & \dots & \sum_{a \in \mathcal{A}} \pi(a|s_{|\mathcal{S}|})\mathbb{P}(s_{|\mathcal{S}|}|s_{|\mathcal{S}|}, a) \end{bmatrix} \tag{11.17}$$

Now we can reiterate the same idea with the rewards to obtain the vector R^π of rewards in which each element i contains the reward obtained when the agent is in state s_i. Note that, the reward vector R^π is a column vector of dimension $|\mathcal{S}| \times 1$.

$$R^\pi = (r(s_1), \dots, r(s_{|\mathcal{S}|}))^T \tag{11.18}$$

The reward vector can be computed as the product between the probability of choosing action a when being in state s and the reward obtained when the agent is in state s and chooses action a.

$$R^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) R(s, a) \tag{11.19}$$

Note that, when we fix a policy π, we don't need the set of actions $\mathcal{A}$ anymore since we have fixed, for each state, what is the best action to perform (or better, what is the probability to do a certain action and go to a certain state).

Thanks to the transition kernel and the reward vector, we can say that a sequence of rewards and states $r_1, s_1, r_2, s_2, \dots$ is a **Markov reward process** defined by the tuple

$$\langle \mathcal{S}, P^\pi, R^\pi, \gamma, \mu \rangle \tag{11.20}$$

Depending on the realisations of the sequence of realisations and states (since both are stochastic) we get a total reward (i.e., the return v_t).

11.5 Bellman Expectation Equation

11.5.1 Policy evaluation

Computing the return v_t requires to sum the future rewards, which we actually don't know. This means that we have to estimate it. The expected return v_t can be estimated using the state-value function V^π.

Definition 11.7 (State-value function)**.** *The state-value function $V^\pi(s)$ of a Markov Decision Process is the expected return v_t starting from state s and then following policy π.*

$$V^\pi(s) = \mathbb{E}_\pi[v_t | s_t = s] \tag{11.21}$$

Basically, the state-value function estimates the return v_t when we start from state s and we choose the next actions following a policy π. This estimation is called policy evaluation.

Definition 11.8 (Policy evaluation)**.** *Policy evaluation allows to compute the utility of each state $s \in \mathcal{S}$, i.e., the return we obtain starting from each state $s \in \mathcal{S}$.*

In some cases it's also useful to compute the the value of each action in each state. In this case we talk about action-value function.

Definition 11.9 (Action-value function)**.** *The action-value function $Q^\pi(s, a)$ of a Markov Decision Process is the expected return v_t starting from state s, taking action a and then following policy π.*

$$Q^\pi(s, a) = \mathbb{E}_\pi[v_t | s_t = s, a_t = a] \tag{11.22}$$

Note that, action a is performed only in the iteration from time t to time $t+1$. Then, after time $t+1$, the agent chooses the best action following policy π.

The state-value function $V^\pi(s)$ and the action-value function $Q^\pi(s, a)$ are related by the equation

$$V^\pi(s) = \sum \pi(a|s) Q^\pi(s, a) \tag{11.23}$$

In other words, V^π is obtained averaging Q^π on the probabilities that the policy chooses each action a.

11.5.2 Bellman Expectation Equation

The state-value function V^π can be decomposed in

- Immediate reward r_{t+1}.
- Future rewards, which can be computed using the (discounted) state-value function, starting from s_{t+1}.

$$V^\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s] \tag{11.24}$$

The expected value can be computed averaging its argument on the policy $\pi(a|s)$ since the returns, for each action a, the probability to take that action.

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\Big(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^\pi(s')\Big) \tag{11.25}$$

Let us understand why this is the expression for the state-value function:

- The first term in the parenthesis is the reward obtained when taking action a from state s. This reward is averaged using policy π on all actions a (i.e., $\sum_a \pi(a|s)R(s,a)$).
- The second term in the parenthesis is the discounted future value. A part from the discount, it's interesting to evaluate the sum $\sum_{s'\in\mathcal{S}}$. This sum averages the future reward that we obtain after applying action a from s. In particular we average the future value using the probability that, with action a from state s the agent goes in state s'.

Reasoning in the same way we can obtain a similar equation for the action-value function $Q^\pi(s,a)$.

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a] && (11.26)\\ &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^\pi(s') && (11.27)\\ &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \sum_{a'\in\mathcal{A}} \pi(a'|s')Q^\pi(s',a') && (11.28)\end{aligned}$$

Thanks to Equation 11.19 and Equation 11.16 (or if not convinced by that, maybe 11.17 is more clear), we can write Equation 11.25 in matrix form as

$$\begin{aligned} V^\pi &= \sum_{a\in\mathcal{A}} \pi(a|s)R(s,a) + \gamma \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} \pi(a|s)P(s'|s,a)V^\pi(s') && (11.29)\\ &= \sum_{a\in\mathcal{A}} \pi(a|s)R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^\pi V^\pi(s') && (11.30)\\ &= R^\pi + \gamma P^\pi V^\pi && (11.31)\end{aligned}$$

where

- N is the number of states.
- R^π is a $N \times 1$ column vector containing, for each state, the estimated reward obtained in that state.
- P^π is a $N \times N$ matrix containing, for each state s_i, the probability to go to state s_j.

- V^π is a $N \times 1$ column vector containing, for each state s_i the return obtained starting from s_i.

Equation 11.31 can be solved to obtain a closed form solution for V^π, as follows.

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

This means that the Bellman Expectation Equation has a closed form solution

$$V^\pi = \big(\mathbf{I} - \gamma P^\pi\big)^{-1} R^\pi \tag{11.32}$$

Note that, even if we have a closed form solution, we still have to pay a quadratic cost to invert the matrix $\big(\mathbf{I} - \gamma P^\pi\big)^{-1}$, which has a cubic cost. Moreover, remember that such matrix can be inverted only if $\gamma < 1$ (which in our case is).

11.5.3 Bellman operators

Another important concept used in Markov Decision Processes is Bellman operators.

Definition 11.10 (Bellman operator)**.** *The Bellman operator is a function $T^\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ that maps value functions to value functions.*

$$\big(T^\pi V\big)(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \Bigg(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \Bigg) \tag{11.33}$$

Fixed point

Note that, if we compute the Bellman operator for $V = V^\pi$, and we can replace Equation 11.25, we obtain

$$\big(T^\pi V^\pi\big)(s) = V^\pi(s) \tag{11.34}$$

which can also be written in matrix form as

$$T^\pi V^\pi = V^\pi \tag{11.35}$$

Note that, when we apply the Bellman operator to V^π, we obtain the same vector V^π.

Definition 11.11 (Fixed point)**.** *The state-value vector V^π is the fixed point of the Bellman operator.*

The fixed point of the Bellman operation is very important and has some fundamental properties:

- V^π is the only fixed point of the Bellman operator.
- T^π is a contraction of V^π. This means that if we apply T^π to a value vector V, we get closer to V^π until. When the Bellman operator returns the same vector we passed as input, we know that the input is V^π.

We can use the second property to obtain V^π using gradient descend-like approach. In particular we can start from a random vector V and keep applying the Bellman operator T^π until we obtain an output that is the same as the input. Basically, we have found an alternative for computing the state-value vector V^π without using the closed form solution. This solution has a quadratic cost in the number of states, and it's useful when the number of states is big (since the direct approach has a cubic cost). Moreover, the speed at which the algorithm converges to V^π depends on γ (the smaller is γ, the faster the algorithm converges).

Reasoning in the same way we can define the Bellman operator also for the action-value function.

Definition 11.12 (Bellman operator)**.** *The Bellman operator is a function $T^\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ that maps value functions to value functions.*

$$\big(T^\pi V^\pi\big)(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q(s',a') \tag{11.36}$$

As before, we can write the action-value Bellman operator in matrix form as

$$T^\pi Q^\pi = Q^\pi \tag{11.37}$$

and Q^π is the fixed point of the Bellman operator.

11.5.4 Optimal value function

Until now, we have considered general value functions that follow a policy π. What we are actually interested in is the value optimal value function, i.e., the one for which the value function is maximum. This means that we want to find the policy π that maximises the value function.

Definition 11.13 (Optimal state-value function)**.** *The optimal state-value function V^* is the maximum value function over all policies π.*

$$V^*(s) = \max_\pi V^\pi(s) \tag{11.38}$$

Basically, the optimal value function returns the best value an agent can get (starting from each state, being a vector). We say that a Markov Decision Problem is solved when we find the optimal value function. Note that, Equation 11.38 is hard to compute since the deterministic case we have $|\mathcal{A}|^{|\mathcal{S}|}$ to search through (we have to evaluate all possible actions for every possible state).

As always we can reason in the same way for the action-value function and define an optimal action-value function.

Definition 11.14 (Optimal action-value function)**.** *The optimal action-value function Q^* is the maximum value function over all policies π.*

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{11.39}$$

11.5.5 Optimal policies

Value functions define a partial ordering over policies since, given two policies π and π', we say that π is better than π' (and write $\pi \geq \pi'$) if the value obtained from policy π is bigger than the one obtained from π'.

$$\pi \geq \pi' \iff V^{\pi}(s) \geq V^{\pi'}(s) \quad \forall s \in \mathcal{S} \tag{11.40}$$

Thanks to the definition of order between policies, we can state the following theorem.

Theorem 11.2 (Optimal policy)**.** *For any Markov Decision Process*

- *There exists an optimal policy π^* that is better than or equal to all other policies.*

$$\exists \pi^* : \pi^* \geq \pi \quad \forall \pi$$

- *All optimal policies achieve the optimal value function.*

$$V^{\pi^*}(s, a) = V^*(s)$$

- *All optimal policies achieve the optimal action-value function.*

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

- *There is always a deterministic optimal policy for any Markov Decision Process which can be found maximising the action choice for all the states.*

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = arg\max_{a \in \mathcal{A}} Q^*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

Bellman optimality equations

The state-value function can be seen as the average, on all actions, of the action-value function (Equation 11.23). If we take the maximum value (i.e., we choose the action that returns the maximum action-value) of the optimal action-state instead of averaging the values, we obtain the optimal state-value function.

$$V^*(s) = \max_a Q^*(s, a) \tag{11.41}$$

$$= \max_a \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^*(s') \right\} \tag{11.42}$$

Equation 11.41 can be used in Equation 11.28 to obtain the optimal action-value function.

$$Q^*(s) = R(s, a) + \gamma \sum_{s,a} P(s'|s, a) V^*(s') \tag{11.43}$$

$$= R(s, a) + \gamma \sum_{s,a} P(s'|s, a) \max_{a'} Q^*(s', a') \tag{11.44}$$

Thanks to the newly found representation of the optimal value functions, we can define the Bellman optimality operator, which closely resembles the Bellman operator (Definition 11.10).

Definition 11.15 (Bellman optimality operator)**.** *The Bellman optimality operator is a function $T^* : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ is a function that maps value functions in value functions.*

$$\big(T^*V\big)(s) = \max_{a\in\mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V(s') \right\} \tag{11.45}$$

As for the Bellman operator, we can compute the output of the Bellman optimality operator for $V = V^*$ to obtain

$$\big(T^*V^*\big)(s) = \max_{a\in\mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^*(s') \right\} \tag{11.46}$$

The right-hand side can be replaced with $V^*(s)$ using Equation 11.41 to obtain

$$\big(T^*V^*\big)(s) = V^*(s) \tag{11.47}$$

which can also be written in matrix form as

$$\big(T^*V^*\big) = V^* \tag{11.48}$$

As we can see, the optimal value function V^* is the **fixed point** for the Bellman optimality operator (in the same way as V^π was the fixed point of the Bellman operator).

As always, we can apply the same reasoning to the action-value function and obtain the Bellman optimality operator for $Q(s,a)$.

Definition 11.16 (Bellman optimality operator)**.** *The Bellman optimality operator is a function $T^* : \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ is a function that maps value functions in value functions.*

$$\big(T^*Q\big)(s) = R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \max_{a\in\mathcal{A}} \left\{ Q^*(s',a') \right\} \tag{11.49}$$

Computing the output of the Bellman optimality operator for $Q = Q^*$ we obtain

$$\big(T^*Q^*\big)(s) = Q^*(s) \tag{11.50}$$

which can also be written in matrix form as

$$\big(T^*Q^*\big) = Q^* \tag{11.51}$$

and Q^* is the fixed point for the Bellman optimality operator.

Before describing how to find the optimal value function, let us list some properties of the Bellman optimality operator:

- **Monotonicity**. If vector f_1 is component-wise smaller than vector f_2 (i.e., $f_{1,i} \leq f_{2,i} \forall i$), then applying the Bellman operator we it holds,

  $$T^\pi f_1 \leq T^\pi f_2 \tag{11.52}$$

  and

  $$T^* f_1 \leq T^* f_2 \tag{11.53}$$

- **Max-norm contraction**. For any two vectors f_1 and f_2, the infinite norm of the difference between $T^\pi f_1$ and $T^\pi f_1$ is smaller than the infinite norm of the difference between f_1 and f_2, scaled by a factor $\gamma < 1$. In formulas

$$\|T^\pi f_1 - T^\pi f_2\|_\infty \leq \gamma \|f_1 - f_2\|_\infty \tag{11.54}$$

and

$$\|T^* f_1 - T^* f_2\|_\infty \leq \gamma \|f_1 - f_2\|_\infty \tag{11.55}$$

Basically, we are saying that the distance between f_1 and f_2 decreases exponentially with γ. Remember that the infinite norm of vector $\mathbf{v}$ is the maximum between all the values of $\mathbf{v}$. If we replace $f_1 = V^*$, the contraction allows to measure the error of a value function f_2.

- V^π is the only fixed point of T^π.
- V^* is the only fixed point of T^*.
- For any vector $f \in \mathbb{R}^{|\mathcal{S}|}$ and policy π, it holds

$$\lim_{k \to \infty} (T^\pi)^k f = V^\pi \tag{11.56}$$

and

$$\lim_{k \to \infty} (T^*)^k f = V^* \tag{11.57}$$

Solving the Bellman optimality equation

The Bellman optimality equation 11.42 is non-linear because of the maximum function, hence it hasn't got, in general, a closed form solution. This means that we can use iterative solution methods to compute its solution. Some examples are:

- **Dynamic programming algorithms** like value iteration and policy iteration.
- **Linear programming**.
- **Reinforcement Learning algorithms** like Q-learning and SARSA.

We will explore the algorithm above later on.

Chapter 12

Dynamic programming

12.1 Introduction

Solving a Markov Decision Process means finding the optimal policy π^* that maximises the value V^{π^*}. Enumerating all possible policies would be infeasible since the space to search is very large (all actions in all possible states, hence $|\mathcal{A}|^{\mathcal{S}}$).

An alternative way to solve a Markov Decision Process is using dynamic programming. The idea of dynamic programming is to split a problem is smaller and easier sub-problems with a recursive structure and then combine the solutions of the sub-problems. This approach fits well for MDP, in fact the Bellman equation requires to recursively compute the value function ($V^{\pi}(s_t) = \cdots + V^{\pi}(s_{t+1})$).

Dynamic programming is a very general framework, however, we can define some important characteristics that DP algorithm should have:

- **Optimal substructure**. The optimal solution must be decomposable in sub-problems.
- **Overlapping sub-problems**. Sub-problems might recur many times, hence the algorithm should be able to cache the solution of the sub-problems so that we can directly use the solution of a sub-problem if it has already been computed. This allows to compute the solution of recurring sub-problems only one.

Luckily, in our case both properties are satisfied since:

- The Bellman equation is recursive.
- The output of a value function can be cached and reused.

12.1.1 Prediction and control

Dynamic programming assumes full knowledge over the MDP and it's used for planning in the MDP (i.e., for finding the optimal sequence of actions to solve a certain task). In particular, two important classes of problems can be solved:

- **Prediction**. Prediction, given a MDP and a policy π, outputs the value function V^{π}.

$$\text{Prediction} : \langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle, \pi \rightarrow V^{\pi} \tag{12.1}$$

$$\text{Prediction} : \langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle, \langle \mathcal{S}, P^{\pi}, R^{\pi}, \gamma, \mu \rangle \rightarrow V^{\pi} \tag{12.2}$$

- **Control**. Control, given a MDP, outputs the value function V^π and the optimal policy π^*.

$$\text{Control} : \langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle \rightarrow V^\pi, \pi \tag{12.3}$$
$$\text{Control} : \langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle \rightarrow V^\pi, \langle \mathcal{S}, P^\pi, R^\pi, \gamma, \mu \rangle \tag{12.4}$$

12.1.2 Finite-horizon dynamic programming

To show how dynamic programming is applied to Markov Decision Processes, we can start from finite-horizon problems since we can reach the end of the interaction and then propagate the results back (i.e., we can recursively compute the components of the Bellman equation). In particular, the optimal value function $V^*(s)$ can be decomposed in sub-problems

$$V_k^*(s) = \max_{a \in \mathcal{A}_k} \left\{ R_k(s,a) + \gamma \sum_{s' \in S_{k+1}} P(s'|s,a) V_{k+1}^*(s') \right\} \tag{12.5}$$

which can be solved recursively. The same idea can be used to compute the optimal policy, in fact we can decompose the problem is easier sub-problems

$$\pi_k^*(s) = \arg\max_{a \in \mathcal{A}_k} \left\{ R_k(s,a) + \gamma \sum_{s' \in S_{k+1}} P(s'|s,a) V_{k+1}^*(s') \right\} \tag{12.6}$$

which can be solved recursively.

Computing the solution to a MDP problem using dynamic programming (in case of finite time horizons) has a cost of

$$N|\mathcal{S}||\mathcal{A}|$$

Having sorted out how to apply dynamic programming for a finite time horizon, we should shift our attention to infinite-horizon problems.

12.2 Policy Iteration

Policy iteration is a **dynamic programming algorithm for control**, i.e., for computing the optimal state-value function V^* and policy π^* for infinite-horizon problems.

The general idea behind policy iteration is to iteratively compute the value function using Equation 11.25 and an iteratively improving greedy policy. In particular:

1. Start from a random policy π.
2. Compute the value V^π for the random policy π.
3. Produce a new improved greedy policy π' using the value function V^π.
4. Repeat from step 2, until we can't improve anymore (i.e., $V^\pi = V^{\pi'}$).

From the general description of the algorithm above, we can split each iteration in two phases:

1. **Policy evaluation**. Policy evaluation allows to estimate the value V^π for a (greedy) policy π. In other words, evaluation returns an estimate of a policy's value, given a policy π.

$$\pi \xrightarrow{V \rightarrow V^\pi} V$$

2. **Policy improvement**. Policy improvement generates a greedy policy π' which performs better than π (i.e., $\pi' \geq \pi$, as for Definition 11.40). In other words, policy improvement returns a new policy π, given the value V^π of a policy π.

$$V \xrightarrow{\pi \to greedy(V)} \pi$$

Figure 12.1 shows a graphical representation of the policy iteration algorithm.

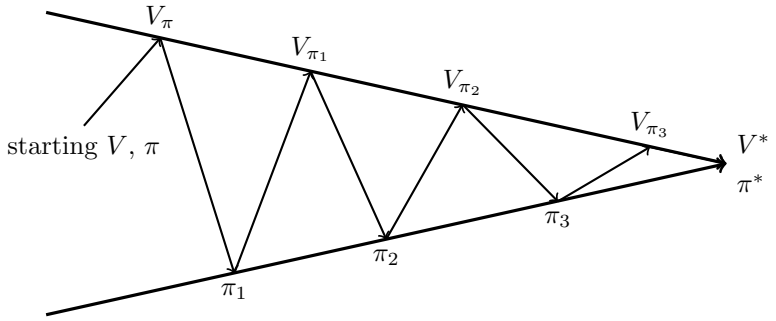


Figure 12.1: A graphical representation of the policy iteration algorithm.

12.2.1 Policy evaluation

Given a deterministic policy π the first phase of policy iteration requires us to compute its value V^π. Depending on the problem we can have multiple options to choose from to compute V^π:

- **Direct solution** (11.32) using the closed form solution.
- **Iterative application of the Bellman expectation equation** until convergence.

Note that policy evaluation is an iterative process itself, i.e., we have a loop (policy evaluation), inside a loop(policy iteration). Moreover, policy evaluation doesn't always start from the the the initial value function V_0 but it begins from the value function V^π obtained at the previous iteration of the policy iteration loop. Finally note that policy evaluation can be used separately from policy iteration to solve a prediction problem because it returns V^π given π.

Policy evaluation backup

The Bellman equation can be iteratively applied to compute the value function V^π of a policy π.

$$V_0 \to V_1 \to \cdots \to V_k \to V_{k+1} \to \cdots \to V^\pi$$

In particular we can use a full policy-evaluation backup

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

that, at each iteration $k+1$, for each state s updates $V_{k+1}(s)$ using the values of $V_k(s)$.

12.2.2 Policy improvement

Given a deterministic greedy policy $\pi_t(s)$ we want to understand if we can improve it and generate a new policy $\pi_{t+1}(s)$. Basically we want to understand if, after evaluating policy $\pi_t(s)$ we want to understand if there exists an action a that has has a better return than $\pi_t(s)$. In particular, we can improve policy $\pi_t(s)$ choosing the action $a \in \mathcal{A}$ that has the highest value.

$$\pi_{t+1}(s) = \arg\max_{a \in \mathcal{A}} Q^{\pi}(s, a) \tag{12.7}$$

Choosing the best greedy policy improves the value from any state s over one step (i.e., over the next step).

$$Q^{\pi}(s, \pi_{t+1}(s)) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \geq Q^{\pi}(s, \pi_t(s)) = V^{\pi}(s) \tag{12.8}$$

This property is supported by the policy improvement theorem.

Theorem 12.1 (Policy improvement)**.** *Let π and π' be any pair of deterministic policies such that*

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \quad \forall s \in \mathcal{S}$$

then the policy π' must be as good as, or better than π.

$$V^{\pi'}(s) \geq V^{\pi}(s) \quad \forall s \in \mathcal{S}$$

A greedy policy can't be implemented indefinitely, in fact, at some point, the value we get from an improved policy π' is the same as the value of π. Basically,

$$Q^{\pi}(s, \pi'(s)) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) = Q^{\pi}(s, \pi_t(s)) = V^{\pi}(s)$$

This means that, after applying the Bellman optimality operator T^* to the value V^{π}, we have obtained a value $V^{\pi'} = V^{\pi}$, hence $V^{\pi'}$ must be the fixed point and therefore the optimal value V^*. Moreover, if $V^{\pi'}$ is the optimal value, then π' is the optimal policy.

Stopping condition

Policy iteration will eventually converge to the optimal policy π^* and value V^*, however we don't know in how many steps. In some cases, it might be useful to define a stopping condition that terminates the algorithm before convergence. An example could be to define a maximum number of iteration.

12.3 Value Iteration

Value iteration is a **dynamic programming algorithm for control**, i.e., for computing the optimal value V^{π^*} and policy π^*. Value iteration iteratively applies the Bellman optimality backup without an explicit policy.

Value iteration works thanks to the following theorem.

Algorithm 3 The policy iteration algorithm.

for s in $\mathcal{S}$ **do** ▷ Initialisation
  $V(s) \leftarrow$ random initialisation
  $\pi(s) \leftarrow$ random initialisation
end for
$V(terminal) \leftarrow 0$
policy-stable $\leftarrow True$
while policy-stable $\neq True$ **do**
  while $\Delta < \delta$ **do** ▷ Policy evaluation
    $\Delta \leftarrow 0$
    for s in $\mathcal{S}$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
  end while
  policy-stable $\leftarrow True$
  for s in $\mathcal{S}$ **do** ▷ Policy improvement
    old-action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    if old-action $\neq \pi(s)$ **then**
      policy-stable $\leftarrow False$
    end if
  end for
end while
Return: V, π

Theorem 12.2 (Value iteration convergence)**.** *Value iteration converges to the optimal state-value function.*

$$\lim_{k\to\infty} V_k = V^* \tag{12.9}$$

Theorem 12.2 states that value iteration will eventually stop and return the optimal value-function V^*, however, in some cases, we'd like to have a stopping condition to terminate execution before convergence, since it might take a lot of time. The following theorem helps us with that.

Theorem 12.3 (Value iteration error)**.** *Given two value functions V_i and V_{i+1}, if the infinite norm of their difference is smaller than an error ε, then the difference between V_{i+1} and the optimal value function V^* is smaller than $\frac{2\varepsilon\lambda}{1-\lambda}$.*

$$\|V_{i+1} - V_i\|_\infty < \varepsilon \Rightarrow \|V_{i+1} - V^*\|_\infty < \frac{2\varepsilon\lambda}{1-\lambda} \tag{12.10}$$

where

$$\|V\|_\infty = \max_s |V(s)|$$

This means that, if the error ε is very small, then V^* can't be to far away from our current guess V.

If we want an error with respect to V^* of ε^* we have to choose $\varepsilon = frac\varepsilon^*(1-\lambda)2\lambda$ and stop when $\|V_{i+1} - V_i\|_\infty$ is smaller than ε.

Algorithm 4 The value iteration algorithm.

while $\Delta < \delta$ **do**
  $\Delta \leftarrow 0$
  for s in $\mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
end while
for s in $\mathcal{S}$ **do**
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
end for
Return: π

12.4 Complexity and efficiency

Policy iteration repeats a certain number of times K (bounded by a stopping criterion or not) policy evaluation and policy improvement. This means that the total cost can be written as

$$C_{tot} = K(C_{eval} + C_{imp}) \tag{12.11}$$

where

- Policy evaluation has a cost dominated by a term which is quadratic, or in some cases cubic, in the number of states.
- Policy iteration that has a cost which is linear in the number of actions.

In particular:

- Each iteration of value iteration has a cost which is quadratic with the number of states.

$$C_{eval,\ vi} = \mathcal{O}(|\mathcal{A}||\mathcal{S}|^2) \tag{12.12}$$

- Each iteration of value iteration has a cost which is quadratic with the number of states

$$C_{eval,\ pi} = \mathcal{O}(|\mathcal{S}|^2 \frac{\log(\varepsilon^{-1})}{\log(\gamma^{-1})}) \tag{12.13}$$

  if we use an iterative approach or cubic (or with exponent 2.373 for efficient matrix inversion algorithms)

$$C_{eval,\ pi} = \mathcal{O}(|\mathcal{S}|^3) \tag{12.14}$$

  if we use the closed form solution.

This means that the cost of policy iteration is dominated by the square of the number of states.

$$C_{policy\ iteration} = \mathcal{O}(C_{evaluation}) \tag{12.15}$$

Unfortunately, the number of states of a problem is usually very big, hence this solution is sometimes impractical in cases where we have to deal with a huge number of states.

Asynchronous dynamic programming techniques can be used for larger problems.

12.5 Linear programming

12.5.1 Optimal value-function

Equation 11.42, also shown above,

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

can be written as a linear programming problem. In particular, we obtain the following problem

$$\begin{aligned} &\min_V \sum_{s \in \mathcal{S}} \mu(s) V(s) \\ &\text{s.t.} V(s) \geq R(s,a) + \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \end{aligned} \tag{12.16}$$

Basically we are saying that we want to find the smallest value function V that is bigger than any other value function. This problems has

- As many variables as number of states $|\mathcal{S}|$.
- As many constraints as the product of the number of states and actions $|\mathcal{S}||\mathcal{A}|$.

Thanks to Problem 12.16 we can say that

Theorem 12.4 (Solution for linear programming)**.** *The optimal value-function V^* is the solution of Problem 12.16.*

Let us demonstrate Theorem 12.4. In particular, we can start from writing the problem using the optimal Bellman operator to obtain

$$\begin{aligned}&\min_V \sum_{s\in\mathcal{S}} \mu^T V\\ &\text{s.t.} V \geq T^*(V)\end{aligned}$$

Thanks to the monotonicity property of the Bellman optimal operator, if $U \geq V$, then $T^*(U) \geq T^*(V)$, hence

$$T^*(V) \geq T^*(T^*(V))$$

If we keep applying the Bellman optimal operator, and thanks to its monotonicity we get

$$V \geq T^*(V) \geq T^*(T^*(V)) \geq T^{*3}(V) \geq T^{*4}(V) \geq \cdots \geq T^{*\infty}(V) = V^*$$

Note that we can write the last equality because V^* is the only fixed point of the Bellman operator, which is eventually always reached. Basically, we have shown that

$$V \geq V^*$$

hence, assuming all values of μ are positive, V^* is the optimal solution to the linear programming problem.

12.5.2 Optimal policy

To find the optimal policy, we can write the dual of Problem 12.16

$$\begin{aligned}&\max_V \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \lambda(s,a)R(s,a)\\ &\text{s.t.} \sum_{a'\in\mathcal{A}} \lambda(s',a') = \mu(s) + \gamma \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \lambda(s,a)P(s'|s,a) \quad \forall s\in\mathcal{S}, \forall a\in\mathcal{A}\\ &\quad \lambda(s,a) \geq 0 \quad \forall s\in\mathcal{S}\end{aligned} \tag{12.17}$$

where

- $\lambda(s,a) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s, a_t = a)$.
- The first constraint ensures that the sum of probabilities is positive (as it should be, being a sum of probabilities).
- The objective function maximises the expected discounted sum of rewards.

Problem 12.17 allows to compute the optimal policy as

$$\pi^*(s) = \arg\max_a \lambda(s,a) \tag{12.18}$$

12.5.3 Complexity

Linear programming converges faster with respect to dynamic programming techniques in the worst-case, however, it' more impractical w.r.t. dynamic programming for problems with a small number of states.

Chapter 13

Reinforcement Learning techniques

13.1 Classification

13.1.1 Techniques

Reinforcement Learning (RL) techniques allow to solve Markov Decision Processes. The first thing we should do is classify RL techniques. RL techniques can be

- **Model-free** or **model-based**.
- **On-policy** or **off-policy**.
- **Online** or **offline**.
  - *Online techniques* learn while collecting data.
  - *Offline techniques* collect data and then learn.
- **Tabular** or **function approximation-based**.
  - *Tabular techniques* learn the value or the utility of each couple state-action pair in a table. This approach doesn't work well with a big number of states.
  - *Function approximation-based techniques* approximate the value function, for each state, action pair, with an actual function instead of using a table. This approach works well with a big number of states, however it's hard to build the approximation function.
- **Value-based**, **policy-based** or **actor-critic**.
  - *Value-based techniques* learn a value function and improve its performance exploring a subset of the policies. Value-based techniques exploit the process, but are not good with a continuous environment.
  - *Policy-based techniques* learn a policy. Policy-based techniques are good in a continuous environment because they use the gradient, however they are less efficient and can get stuck a local optimum. Policy based techniques are also good for non-fully observable environments.

  - *Actor-critic techniques* are a combination of value and policy-based techniques. In particular, the actor part searches a policy and the critic part evaluates it (i.e., computes the value). Basically, this category of techniques takes the best of value and policy-based ones.

13.1.2 Problems

Reinforcement Learning problems can be divided in

- **Model-free prediction**. Prediction problems require to estimate the value function of an unknown Markov Decision Problem and a policy.
- **Model-free control**. Control problems require to optimise the value function of an unknown Markov Decision Process.

Chapter 14

Model-free prediction

14.1 Monte-Carlo reinforcement learning

Monte-Carlo (MC) is a reinforcement learning technique that learns directly from experience, hence it can be applied only to episodic problems where we can reach an absorbing state. Basically an episode is a series of events in which there is a state in which the episode ends and we can update the value function. This means that MC is partially online since the algorithm collects a number of observations until reaching the absorbing state where it can learn. The value function is estimated using the mean of the samples obtained from the episodes.

An example of episodic problem could be learning to park a car. The algorithm handles the car until it crashes or successfully parks. The sequence of states between the starting state and the state in which it crashes or parks is an episode.

Monte-Carlo can be applied for

- **Prediction**. In this case, the algorithm takes as input episodes $\{s_1, a_1, r_2, \dots, s_T\}$ generated by a known policy π and outputs the value function V^π.
- **Control**. In this case, the algorithm takes as input episodes $\{s_1, a_1, r_2, \dots, s_T\}$ and outputs the optimal value function V^* and the optimal policy π^*.

In both cases, the input contains episodes, each of which is a series of actions, states and rewards that ends with an absorbing state s_T.

14.1.1 Mean estimation

Monte-Carlo is based on the mean return of the episodes. Since the mean is so important let us recall some concepts. Let X be a random variable with mean $\mu = \mathbb{E}[x]$ and variance $\sigma^2 = Var[X]$ and $x_i \sim X$ independent identically distributed realisation of X. The empirical mean of X is

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{14.1}$$

The expected value of the empirical mean is the mean of X.

$$\mathbb{E}[\hat{\mu}_n] = \mu = \mathbb{E}[X] \tag{14.2}$$

The variance of the empirical mean is

$$Var[\hat{\mu}_n] = \frac{Var[X]}{n} \tag{14.3}$$

14.1.2 Monte-Carlo policy evaluation

Monte-Carlo uses the empirical mean of the return, using a certain policy π, to estimate the value function.

$$V^\pi = \mathbb{E}[v_t | s_t = s] \tag{14.4}$$

where v_t is the discounted return (as defined by Equation 11.10)

$$v_t = \sum_{i=1}^{T} \gamma^{i-1} r_{t+i} \tag{14.5}$$

Basically, we are estimating $\mathbb{E}[v_t|s_t = s]$ with the empirical mean (14.1). Monte-Carlo uses two different approaches when computing the empirical mean:

- **First visit**. First visit averages the return using one sample for each state we visit. This means that, if we visit a state s multiple times in an episode, we only consider one sample from that state. Estimators that use this technique are unbiased.
- **Every visit**. Every visit averages the return using all samples collected, even multiple samples for a state s. Estimators that use this technique are biased but consistent.

The general idea behind Monte-Carlo policy evaluation is:

1. Generate an episode $\mathcal{E}$.
2. For all states s in the episode save the return obtained starting from s.
3. Average the returns obtained starting from s. This average is computed on the returns obtained in all episodes until now, and not only on the current one.
4. Repeat from step 2, or stop.

A more formal version of the general algorithm is shown in Algorithm 5 and 6.

Comparison between first visit and every visit

To understand in which cases one model is better than the other we can plot the average root mean square error with respect to the number of episodes we consider. From Figure 14.1, we notice that the error of the first-visit algorithm is bigger when the number of episodes is small, however it improves and becomes smaller than the error of the every-time algorithm for a large number of episodes.

Algorithm 5 First-visit Monte-Carlo

```
π ← policy to evaluate
V ← an arbitrary state-value function
ℰ ← episode
Returns(s) ← {}                          ▷ list of returns, one for each state s
Visited ← {}                             ▷ states already visited
while True do
    ℰ ← episode using π
    Visited ← {}
    for s in ℰ do
        Visited ← Visited + s
        if s not in Visited then
            R ← return starting from s
            Returns(s) ← Returns(s) + R
            V(s) ← average(Returns(s))
        end if
    end for
end while
```

Algorithm 6 Every-visit Monte-Carlo

```
π ← policy to evaluate
V ← an arbitrary state-value function
ℰ episode
Returns(s) ← {}                          ▷ list of returns, one for each state s
while True do
    ℰ ← episode using π
    for s in ℰ do
        R ← return starting from s
        Returns(s) ← Returns(s) + R
        V(s) ← average(Returns(s))
    end for
end while
```

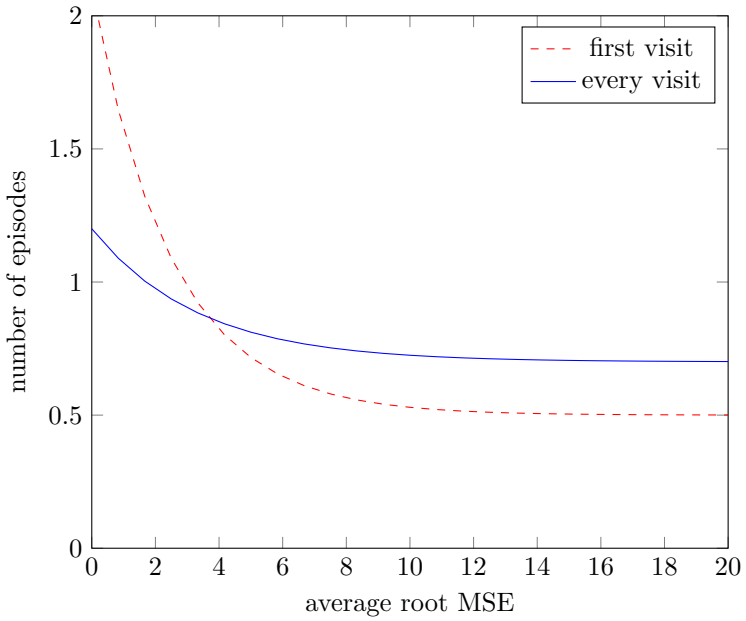


Figure 14.1: Mean square error of first-visit and every-visit Monte-Carlo.

14.1.3 Incremental mean

Since one might want to use a lot of episodes, computing the average from zero every time might be costly. For this reason it's useful to introduce a way to update the mean of a sequence of realisations $x_1, x_2, \ldots$.

$$\hat{\mu}_k = \frac{1}{k}\sum_{j=1}^{k} x_j \tag{14.6}$$

$$= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \tag{14.7}$$

$$= \frac{1}{k}(x_k + (k-1)\hat{\mu}_{k-1}) \tag{14.8}$$

$$= \hat{\mu}_k + \frac{1}{k}(x_k - \hat{\mu}_{k-1}) \tag{14.9}$$

Note that we have obtained Equation 14.8 from equality

$$\hat{\mu}_{k-1} = \frac{1}{k-1}\sum_{j=1}^{k-1} x_j$$

$$(k-1)\hat{\mu}_{k-1} = \sum_{j=1}^{k-1} x_j$$

Equation 14.9 can be applied to compute the approximation of the value function $V(s)$. In particular, for each state s_t with return v_t (i.e., we get v_t after reaching s_t), we can write

$$V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}\big(v_t - V(s_t)\big) \tag{14.10}$$

where $N(s_t)$ counts the number of times state s_t has been visited. In some cases (e.g., for non-stationary problems) it's useful to compute the running mean, namely to forget old episodes. In this case, we use a parameter α that allows to give more priority to new samples (i.e., to forget older ones).

$$V(s_t) \leftarrow V(s_t) + \alpha\big(v_t - V(s_t)\big) \tag{14.11}$$

In particular,

- If $\alpha > \frac{1}{N}$, we give more importance to recent weights.
- If $\alpha < \frac{1}{N}$, we give less importance to recent weights.

Instead of considering a single learning factor α we can use a sequence of learning factors α_i to obtain what's called the **exponential average**.

$$V(s_t) \leftarrow V(s_t) + \alpha_t\big(v_t - V(s_t)\big) \tag{14.12}$$

Note that, if

- $\sum_i \alpha_i = \infty$.
- $\sum_i \alpha_i^2 < \infty$

then the estimated mean $\hat{\mu}_n$ converges to μ_n and we say that the estimator $\hat{\mu}_n$ is consistent. This means that the estimation of $V(s)$ converges to $V(s)$.

14.2 Temporal Difference

Temporal Difference (TD) is a **model-free estimator for the value function** that learns directly from episodes. Differently from Monte-Carlo, TD doesn't have to wait the end of an episode to start learning, hence it is a purely online algorithm. For this reason we say that Temporal Difference uses bootstrapping, because it learns from incomplete episodes. One might ask, how can TD estimate V if it doesn't have all the states of an episode. The answer is that the unknown states are replaced by a guess, generated using an empirical version of the Bellman equation.

14.2.1 Temporal Difference-0

The most basic version of Temporal Difference is TD 0 and uses the incremental value equation (14.11) used for Monte-Carlo. Let's start by writing the running incremental value.

$$V(s_t) \leftarrow V(s_t) + \alpha(v_t - V(s_t)) \tag{14.13}$$

Since we are estimating purely online, we don't know v_t, because it's the return starting from s_t but we know only the states until s_t. This means that we have to replace it with an estimate. The easiest way to do it is by using the estimate return

$$r_{t+1} + \gamma V(s_{t+1})$$

to obtain

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{14.14}$$

where

- $r_{t+1} + \gamma V(s_{t+1})$ is called **Temporal Difference target**.
- $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called **Temporal Difference error**.

Note that $V(s_{t+1})$ is known because it simply is the value-function of a state s_{t+1} which we might have already seen in the past.

14.2.2 Comparison with Monte-Carlo

Monte-Carlo and Temporal Difference use a similar approach, hence it's good to compare them:

- Temporal Difference can learn before knowing the final outcome of the episode, i.e., it can update the value function after each event. This means that temporal difference can work in non-terminating environments (i.e., where we don't have an absorbing state).
- Monte-Carlo has to wait the end of an episode to before estimating the value function. This means that Monte-Carlo works only for episodic environments.

Another important analysis we can do regards bias-variance. In particular:

- Monte-Carlo generates an unbiased estimate of $V^\pi(s)$ since it uses the actual rewards, hence the model closely resembles the real model. However, this means that we have a lot of variance.
- Temporal Difference generates a biased estimate of $V^\pi(s)$ which however has much lower variance since it depends only on one random action, one transition and one reward.

Long story short:

- Monte-Carlo has **high variance** but **zero bias**. This means that it
  - Has good convergence properties.
  - Works well with function approximation.
  - Is not very sensitive to the initial value of V.
  - Is very simple to understand and use.
- Temporal Difference has **low variance** but **some bias**. This means that it
  - It is usually more efficient than Monte-Carlo.
  - TD(0) converges to $V^\pi(s)$
  - Doesn't work well with function approximation.
  - Is very sensitive to the initial value of V.

Another important analysis we can do is how the learning rate α affects variance. Monte-Carlo can't use a big learning rate α since it would introduce high variance. If we remember that we can put $\alpha = \frac{1}{N}$, this means that we have to use a large number of episodes for having low variance when using Monte-Carlo.

Another difference between Monte-Carlo and Temporal Difference is that

- Temporal Difference exploits the Markov property, hence it works efficiently in Markov environments.
- Monte-Carlo doesn't exploit the Markov property, hence it works more efficiently in non-Markov environments.

14.3 Temporal Difference-Lambda

TD(0) uses only one step ahead to estimate the value-function, while Monte-Carlo considers all steps (∞ to the limit) ahead. In the middle we have a range of algorithms, called Temporal Difference Lambda TD(λ) which use n steps after the current state. Note that n allows to balance variance, since TD(0) has low variance while Monte-Carlo has high variance. Moreover, the largest is the value of n, the smallest is the optimal learning rate α (the bigger n, the more is the variance, the smallest has to be the learning rate).

In particular, the difference between different values of n is how the TD target is computed. In particular we can write

$$v_t^{(n)} = \sum_{i=0}^{n} \gamma^{i-1} r_{t+i} + \gamma^n V(s_{t+n}) \tag{14.15}$$

For instance we have

- For $n = 1$, i.e., TD(0)
$$v_t^{(1)} = r_{t+1} + \gamma V(s_{t+1}) \tag{14.16}$$
- For $n = 4$
$$v_t^{(4)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \gamma^4 V(s_{t+4}) \tag{14.17}$$
- For $n = \infty$, i.e., Monte-Carlo
$$v_t^{(\infty)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-1} r_T \tag{14.18}$$

The n TD target (14.15) can be used to compute the estimate of the value.

$$V(s_t) \leftarrow V(s_t) + \alpha\big(v_t^{(n)} - V(s_t)\big) \tag{14.19}$$

14.3.1 Lambda return

Now that we know how to compute the n-step return, we can average the returns with different ns with a weight defined by the parameter λ. For instance, we can approximate the return v_t as

$$v_t(s) = \frac{1}{2} v^{(2)}(s) + \frac{1}{2} v^{(4)}(s)$$

This allows to combine information form different time-steps.

In general, we can define the λ-return v_t^λ as the return that combines all n-steps returns $v_t^{(n)}$ using λ as weight.

$$v_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} v_t^{(n)} \tag{14.20}$$

Note that,

- For $\lambda = 0$, we have **TD(0)**.
- For $\lambda = 1$, we have a roughly equivalent version of **Monte-Carlo**.

The λ-return can be used in different ways to compute the value-function. In particular, the two main techniques are

- **Forward-view TD(λ)**.
- **Backward-view TD(λ)**.

14.3.2 Forward-view TD-Lambda

Forward-view TD(λ) updates the value function $V(s_t)$ for state s_t using future rewards. This means that we are basically using the λ-return in the running incremental value function equation (14.11).

$$V(s_t) \leftarrow V(s_t) + \alpha\big(v_t^\lambda + V(s_t)\big) \tag{14.21}$$

In other words, forward TD(λ) looks into the future to compute v_t^λ. This means that forward-view TD(λ) needs complete episodes, because it needs the returns after state s_t.

One last consideration regarding forward TD(λ). Since λ decides how close are we to TD(0) with respect to Monte-Carlo, we can say that the smaller is λ, the larger α can be since for small λs we get close to TD(0) which has low variance and can use an high value of α.

14.3.3 Backward-view TD-Lambda

Forward view TD(λ) is theoretically valid and perfectly describes how Reinforcement Learning techniques work. That being said, this technique is hard to practically use. Backward-view TD(λ) comes into play for practically compute the value-function $V(s_t)$ using a pure only process that, step by step (i.e., using incomplete sequences), upgrades the value of $V(s_t)$.

To describe how backward-view TD(λ) works, we have to introduce the concept of eligibility trace. An eligibility trace allows to assign credits to

- The most recently seen states.
- The most frequently seen states.

In particular, the eligibility e_{t+1} assigned at state s is

$$e_{t+1}(s) = \gamma\lambda e_t(s) + \mathbf{1}(s = s_t) \tag{14.22}$$

where

- λ is used to define how long the trace has to be, namely how much we care about the previous eligibility. For $\lambda = 0$, we don't care about the previous value and for $\lambda = 1$ we care about all previous values.
- $\mathbf{1}(s = s_t)$ returns 1 if the state s is equal to the current state s_t.

Basically, for each state $s \in \mathcal{S}$ we update $e_{t+1}(s)$ using Equation 14.22. The eligibility is used to update the value function $V(s)$ on all the states s that have eligibility. In particular, the equation used to update the eligibility is

$$V(s) \leftarrow V(s) + \alpha\delta_t e_t(s) \tag{14.23}$$

As we can notice, the bigger is the eligibility $e_t(s)$ for a state s, the more we update the value function. If a state s has no eligibility (i.e. $e_t(s) = 0$), it's value function $V(s)$ doesn't change. Also note that for $\lambda = 0$, i.e., for TD(λ), only the current state is updated since $e(s)$ is 0 for all states different than the current one. The complete algorithm is shown in Algorithm 7.

Accumulating traces for frequently visited states can have eligibility greater than 1. This can be a problem for convergence. To solve this problem we can use a modified version of eligibility that replaces the eligibility with 1 instead of adding 1, if the state is the current state. The new eligiblity function is

$$e_{t+1}(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \textbf{if } s \neq s_t \\ 1 & \textbf{if } s = s_t \end{cases} \tag{14.24}$$

Algorithm 7 The backward-view TD(λ) algorithm.

for s in $\mathcal{S}$ **do**
  $V(s) \leftarrow$ random initialisation
end for
for episode t in $\mathcal{T}$ **do**
  for s in $\mathcal{S}$ **do**
    $e(s) \leftarrow 0$
  end for
  $s \leftarrow$ initial state of the episode
  while s not terminal **do**
    $a \leftarrow$ action given by π for s
    Take action a
    Observe reward r
    Observe next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for s in $\mathcal{S}$ **do**
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    end for$s \leftarrow s'$
  end while
end for

14.3.4 TD(1) and Monte-Carlo

When $\lambda = 1$, the sum of TD errors converges into the Monte-Carlo error. This means that TD(1) is roughly equivalent to every-visit Monte-Carlo where the error is accumulated online. Moreover,

- If the value function is only updated offline at end of episode, then the total update is exactly the same as Monte-Carlo.
- If the value function is updated online after every step, then TD(1) may have different total update to Monte-Carlo.

Chapter 15

Model-free control

Many Markov Decision Processes can't be modelled or, if the model is known, are too big to handle. To approach these problems, we can use model-free control algorithms (i.e., control algorithms that don't require a model). Model free problems can be divided in

- **On-policy** learning problems. On-policy learning learns about policy π from experience sampled from π.
- **Off-policy** learning problems. Off-policy learning learns about policy π from experience sampled from another policy $\bar{\pi}$.

15.1 On-policy control

15.1.1 Monte-Carlo control

The idea behind on-policy Monte-Carlo control is to use the policy iteration model. This means that we have to define how to perform

- **Policy improvement**. For policy improvement, we can't use greedy policy improvement (12.7) over $V()$ since it requires a model of the problem. Luckily, policy improvement over $Q(s, a)$ is model free
$$\pi'(s) = \arg\max_{a \in \mathcal{A}} Q(s, a) \tag{15.1}$$
we don't have to consider the probability $\pi(a|s)$
- **Policy evaluation**. For policy evaluation, we can use Monte-Carlo policy evaluation over the action-state function Q (since we use it for policy improvement). Namely, we want to compute Q^π.

The main problem with this approach is that using a deterministic action-value function Q, we will always choose the best action. This might be a problem since, if we keep choosing the best action, we might lose some better action which haven't been explored yet. This problem is known as exploration-exploitation problem and can be seen as the dilemma between keep doing actions that we know are good, or exploring new actions that are currently bad but that might lead to a bigger reward.

epsilon-greedy exploration

To handle the exploration-exploitation dilemma, we can use ε-greedy exploration which uses the hyper-parameter ε that sets the amount of exploration. In particular,

- With $\varepsilon = 0$ the agent is greedy and selects the best action.
- With $\varepsilon = 1$ the agent is random and always chooses an action at random.

Usually, at the beginning the agent should be able to explore as many actions as possible since it doesn't know what is the best action and, after a while, follow the best actions since it already knows the best action. ε-greedy exploration defines policy improvement as

$$\pi(s,a) = \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon & \text{if } a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \frac{\varepsilon}{m} & \text{otherwise} \end{cases} \tag{15.2}$$

where $\pi(s,a)$ is the probability that in state s the agent chooses action a. Basically, we are choosing

- A random action with probability ε.
- A greedy action with probability $1 - \varepsilon$.

The following theorem shows that ε-greedy exploration actually improves the policy π.

Theorem 15.1 (ε-greedy exploration improvement)**.** *For any ε-greedy policy π, the ε-greedy policy π' with respect to Q^π is an improvement.*

$$Q^\pi(s, \pi'(s)) = \sum_{a \in \mathcal{A}} \pi'(a|s) Q^\pi(s,a) \tag{15.3}$$

$$= \frac{\varepsilon}{m} \sum_{a \in \mathcal{A}} Q^\pi(s,a) + (1-\varepsilon) \max_{a \in \mathcal{A}} Q^\pi(s,a) \tag{15.4}$$

$$\geq \frac{\varepsilon}{m} \sum_{a \in \mathcal{A}} Q^\pi(s,a) + (1-\varepsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \frac{\varepsilon}{m}}{1-\varepsilon} Q^\pi(s,a) \quad = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s,a) \tag{15.5}$$

$$= V^\pi(s) \tag{15.6}$$

Therefore, from policy improvement Theorem 12.1, $V^{\pi'(s)} \geq V^\pi(s)$.

GLIE

GLIE is an important property of a classification algorithm.

Theorem 15.2 (Greedy in the Limit of Infinite Exploration)**.** *An algorithm is Greedy in the Limit of Infinite Exploration if*

- *All state-action pairs are explored infinitely many times.*

$$\lim_{k \to \infty} N_k(s,a) = \infty \tag{15.7}$$

- *The policy converges on a greedy policy.*

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = arg\max_{a'\in\mathcal{A}} Q_k(s', a')) \tag{15.8}$$

Since GLIE is so important, we can define a GLIE version of Monte-Carlo. In particular, GLIE Monte-Carlo works as follows:

1. Sample the k-th episode using policy $\pi : \{s_1, s_1, r_2, \dots, s_T\} \sim \pi$.
2. For each state s_t and action a_t in the episode

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

3. Improve policy based on the new action-value function setting

$$\varepsilon \leftarrow \frac{1}{k}$$

and

$$\pi \leftarrow \varepsilon - \text{greedy}(Q)$$

The GLIE Monte-Carlo algorithm converges thanks to the following theorem.

Theorem 15.3 (GLIE Monte-Carlo convergence)**.** *GLIE Monte-Carlo control converges to the optimal action-value function.*

$$Q(s, a) \to Q^*(s, a) \tag{15.9}$$

Time scales

In Monte-Carlo control there are three main parameters to set

- Behavioural time scale governed by parameter γ.
- Sampling in the estimation of the Q-function, governed by the learning rate α.
- Exploration governed by ε.

A general rule for giving a value to these parameters is

$$1 - \gamma \gg \alpha \gg \varepsilon$$

Initially, one could set $1 - \gamma \approx \alpha \approx \varepsilon$. Then we can decrease ε faster than α. Otherwise, if we have a finite number of episodes, we can approximate the parameters as

- $\alpha \sim 1 - \frac{m}{M}$
- $\varepsilon \sim \left(1 - \frac{m}{M}\right)^2$

Algorithm 8 The Monte Carlo control algorithm.

Initialize:
for s in $\mathcal{S}$ **do**
  $\pi(s) \leftarrow$ random policy
  for a in $\mathcal{A}$ **do**
    $Q(s,a) \leftarrow$ random initialisation
    $Returns(s,a) \leftarrow \{\}$
  end for
end for
while $True$ **do**
  Choose $s_0 \in \mathcal{S}$, $a_0 \in \mathcal{A}(s_0)$ randomly such that all pairs have probability > 0
  $e \leftarrow s_0, a_0, r_1, \ldots, s_{T-1}, a_{T_1}, r_T$ ▷ Generate an episode from s_0, a_0 following $\pi(s)$
  $G \leftarrow 0$
  for t in $T-1, T-2, \ldots, 0$ **do** ▷ Policy evaluation
    $G \leftarrow \gamma G + R_{t+1}$
    if (s_t, a_t) not in $s_0, a_0, \ldots, s_{t-1}, a_{t-1}$ **then**
      $Returns(s,a) \leftarrow Returns(s,a) + G$ ▷ Append G to the returns
      $Q(s_t, a_t) \leftarrow average(Returns(s_t, a_t))$
      $\pi(s_t) \leftarrow \arg\max_a Q(s_t, a)$
    end if
  end for
end while

15.1.2 Temporal Difference control and SARSA

Temporal Difference estimation has many advantages over Monte-Carlo, one of which is low variance. This means that it has sense to use Temporal Difference for estimating the value in the policy evaluation phase of policy iteration. Basically, on-policy temporal difference control uses policy iteration where

- Policy evaluation is done with Temporal Difference.
- Policy improvement is done using ε-greedy policy improvement.

SARSA

One way to evaluate a policy π using Temporal Difference is by using SARSA. In particular, SARSA updates the action-value function Q using the following equation

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + Q(s',a') - Q(s,a)\big) \tag{15.10}$$

More precisely, the SARSA policy evaluation algorithm is shown in Algorithm

As always, we have a theorem that tells us that SARSA converges to the actual action-value function.

Theorem 15.4 (SARSA convergence)**.** *SARSA converges to the optimal action value function*

$$Q(s,a) \to Q^*(s,a) \tag{15.11}$$

Algorithm 9 The SARSA policy evaluation algorithm.

$Q(s, a) \leftarrow$ random initialisation
while True **do**
  $s \leftarrow$ initial state for the episode.
  Choose a from s using policy derived using ε-greedy policy.
  while s is terminal **do**
    Take action a
    Observe r
    Observe s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while
end while

under the following conditions

- *GLIE sequence of policies $\pi_t(s, a)$ (Theorem 15.2).*
- *Robbins-Monro sequence of step-sizes α_t*

$$\sum_{t=1}^{\infty} a_t = \infty$$
$$\sum_{t=1}^{\infty} a_t^2 < \infty$$

A formalisation of the SARSA algorithm is shown in Algorithm 9. As we can see, there is no explicitly policy improvement step as the General Policy Iteration framework (i.e., the generalisation of the Policy Improvement technique we saw when analysing dynamic programming techniques) would suggest. The reason for this is that the policy improvement is inferred from the fact that we are using the action value function $Q(s, a)$. From this function we can take the best greedy action for each state (i.e., the one with the biggest value). The only thing we have to decide is the value of ε, i.e., the probability of choosing a non-optimal action.

SARSA with eligibility traces

As for Temporal Difference, we can apply eligibility traces to SARSA, too. In this case we talk about SARSA(λ) and we can apply both forward and backward view algorithms. As for TD(λ) we can define intermediate steps between Monte-Carlo and Time Difference on-policy control. In this case, the eligibility rule assign an eligibility to each couple state s, action a.

$$e_t(s, a) = \gamma e_{t-1}(s, a) + \mathbf{1}(s_t = s, a_t = a) \tag{15.12}$$

A complete version of the algorithm is shown in Algorithm 10. Note that, depending on the value of λ can represent the Monte-Carlo ($\lambda = 1$) or Time Difference ($\lambda = 0$) algorithm.

Algorithm 10 The SARSA(λ) algorithm.

$Q(s,a) \leftarrow$ arbitrarily initialisation
while True **do**
  for (s,a) in $\mathcal{S} \times \mathcal{A}$ **do** $e(s,a) \leftarrow 0$
  end for
  while s not terminal **do**
    Take action a
    Observe r
    Observe s'
    Choose a' from s' using policy derived from Q (e.g., ε–greedy).
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for (s,a) in $\mathcal{S} \times \mathcal{A}$ **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    end for$s \leftarrow s'$ $a \leftarrow a'$
  end while
end while

15.2 Off-policy learning

Off-policy learning is based on the idea of learning about policy $\pi(a|s)$ using another policy $\bar{\pi}(a|s)$. More precisely:

- $\pi(a|s)$ is called **target policy**. The target policy is what we want to learn.
- $\bar{\pi}(a|s)$ is called **behaviour policy**. The behaviour policy is what we use to learn the target policy and interacts with the environment (i.e., it's the one used for choosing the best action).

The main advantages of off-policy learning are:

- We can use previous policies to learn a new policy.
- We can learn an optimal policy while following an exploratory policy.
- We can learn about multiple policies while following one policy.

15.2.1 Importance sampling

Important sampling estimates the expectation $\mathbb{E}$ of a different distribution (i.e., $\pi(a|s)$) with respect to the distribution used to draw samples (i.e., $\bar{\pi}(a|s)$). Important sampling requires less samples than Monte-Carlo but, in our case, it increases variance. In particular,

$$\mathbb{E}_{x\sim P}[f(x)] = \sum P(x)f(x) \tag{15.13}$$

$$= \sum Q(x)\frac{P(x)}{Q(x)}f(x) \tag{15.14}$$

$$= \mathbb{E}_{x\sim Q}\left[\frac{P(x)}{Q(x)}f(x)\right] \tag{15.15}$$

where, in our case

- f is the return v_t.
- x is the sequence of states s_t.
- P is the probability that sequence of states using the target policy (P) or under the behaviour policy (Q).
- $\frac{P(x)}{Q(x)}$ is the importance weight.

15.2.2 Importance sampling for off-policy Monte-Carlo

Important sampling can be applied also to Monte-Carlo. This means that we have to define a formula for v_t for updating $Q(s,a)$. From the definition in importance sampling, we have to weight the return v_t using the target policy π and the behaviour policy $\bar{\pi}$ for every episode in the episode. The return function is therefore

$$v_t^{\mu} = \frac{\pi(a_1|s_1)}{\bar{\pi}(a_1|s_1)} \frac{\pi(a_2|s_2)}{\bar{\pi}(a_2|s_2)} \cdots \frac{\pi(a_T|s_T)}{\bar{\pi}(a_T|s_T)} \tag{15.16}$$

Thanks to Equation 15.16 we can write the action-value function update as

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\big(v_t^{\mu} - Q(s_t,a_t)\big) \tag{15.17}$$

Note that this technique can't be used if $\bar{\pi}$ is zero and π is not zero. A more formal algorithm is shown in Algorithm 11

Algorithm 11 Monte-Carlo importance sampling.

for (s,a) in $\mathcal{S} \times \mathcal{A}$ **do**
  $Q(s,a) \leftarrow$ arbitrary initialisation
  $N(s,a) \leftarrow 0$
  $D(s,a) \leftarrow 0$
end for$\pi \leftarrow$ arbitrary deterministic policy
while do
  $\mathcal{E} \leftarrow \hat{\pi}$ an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, r_{T-1}, a_{T-1}, r_T, s_T$ using a policy
  $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
  for (s,a) in $\mathcal{E}$ after τ **do**
    $t \leftarrow$ time of the first occurrence of (s,a)
    $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\bar{\pi}(s_k,a_k)}$
    $N(s,a) \leftarrow N(s,a) + wR_t$
    $D(s,a) \leftarrow D(s,a) + w$
    $Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
  end for
  for s in $\mathcal{S}$ **do** $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a)$
  end for
end while

15.2.3 Important sampling for off-policy SARSA

Important sampling can also be applied to SARSA. This is particularly useful because importance sampling has an high variance, and SARSA can reduce it (since we know that SARSA is characterised by a low variance).

Since we are talking about SARSA (which we remember coming from Temporal Difference) we have to define the Temporal Difference target $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$. The target can be computed, remembering 15.15, considering the ratio between the target policy and the behavior policy, this time considering only the next event. What we get is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_{t+1} + \gamma \frac{\pi(a_{t+1}|s_{t+1})}{\hat{\pi}(a_{t+1}|s_{t+1})} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big) \tag{15.18}$$

15.2.4 Q-learning

Q-learning allows to learn an optimal policy $\pi = \pi^*$ from experience sampled from a behaviour policy $\bar{\pi}$. Note that, the behavior policy $\bar{\pi}$ has to be stochastic because it can't have null values.

Q-learning can be seen as an empirical version of the value-iteration and its update equation for the Q function is

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)\big) \tag{15.19}$$

Formally, the algorithm is shown in Algorithm 12

Algorithm 12 The Q-learning algorithm.

```
for (s, a) in S × A do
    Q(s, a) ← arbitrary initialisation
end for
while True do
    s ← starting state
    while s not terminal do
        Choose a from s using policy derived from Q (e.g., using ε-greedy)
        Choose action a
        Observe reward r
        Observe next state s'
        Q(s, a) ← Q(s, a) + α(r + γ max_{a'∈A} Q(s', a') − Q(s, a))
        s ← s'
    end while
end while
```

Q-learning converges but it only requires the Morrow property (check Theorem 15.11).
Let us now compare SARSA and Q-learning:

- Q-learning learns using the random policy but it can't learn using a deterministic policy.
- SARSA is aware of the exploration phase, hence it learns a policy that depends on the exploration. This means that we need to change ε progressively.

Chapter 16

Multi Armed Bandit

16.1 Introduction

Multi Armed Bandit is a framework used for reinforcement learning problem that addresses the exploration-exploitation problem. The basic idea is that, we want to explore as many opportunities and gather as many information as possible and to find the best policy. Still we don't want to spend too much time exploring non-rewarding actions. Depending on how much the agent wants to sacrifice immediate reward (i.e., following an action we know is good, at least for now), it can consider

- An infinite time horizon which allows to gather as much information as possible.
- a finite time horizon which allows to minimise the short-term loss due to uncertainty. Basically, the agent doesn't want to explore actions that are currently not convenient (i.e., have bad reward).

More formally, we don't know what the value function $Q(a|s)$ of each action is and we want to compute it online (i.e., improving the estimation with every new event we experience).

> **Definition 16.1** (Arm)**.** *In Multi Armed Bandit problems, we call arm one of the possible actions we can do.*

Clinical trial example To make things clearer, let us consider the following example. Say we want to test different drugs to understand which is more effective. We want to test as many drugs as possible on many patients and in different conditions (e.g., dosage, physical conditions, and so on). Basically we want to explore all different opportunities. Still, we don't want to give to patients some drugs that it looks like aren't working. Basically, we want to exploit drugs that initially look promising.

16.1.1 Reward

In a Multi Armed Bandit problem, the reward can be

- **Deterministic**. Each arm returns a single deterministic value.

- **Stochastic**. Each arm returns a value drawn from a distribution which is stationary over time (i.e., the distribution doesn't change with time).
- **Adversarial**. Each arm returns a value chosen by an adversary. The adversary chooses the reward at each round and knows the algorithm we are using to solve the problem.

16.2 Stochastic Multi Armed Bandit

16.2.1 Formalisation

Formally, a Multi Armed Bandit problem can be described as a specific case of a Markov Decision Process.

Definition 16.2 (Multi Armed Bandit problem)**.** *A Multi Armed Bandit problem is a tuple*

$$\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu^0 \rangle \tag{16.1}$$

where

- $\mathcal{S}$ *is the set of states. A MAB uses one state only, hence*

$$\mathcal{S} = \{s\}$$

- $\mathcal{A}$ *is the set of possible actions (i.e., the arms).*

$$\mathcal{A} = \{a_1, \dots, a_N\}$$

- P *is the state transition probability matrix that returns, given an action* a *and a state* s*, the probability to go to state* s'*. Since we have only one state, the transition matrix is*

$$P(s|a_i, s) = 1 \quad \forall a_i \in \mathcal{A} \tag{16.2}$$

- R *is the reward function. Since we have only one state, the reward function depends only on the action performed.*

$$R(s, a_i) = R(a_i)$$

- γ *is the discount factor. In MAB problems we want to consider only a finite time horizon, hence*

$$\gamma = 1 \tag{16.3}$$

- μ^0 *is the set of initial probabilities. Since there is only one state, we start from that state with probability 1.*

$$\mu_0(s) = 1$$

Since a MAB problem has only one state, many of the elements of a MDP are useless. More concisely, a Multi Armed Bandit can be defined as follows.

Definition 16.3 (Multi Armed Bandit)**.** *A Multi Armed Bandit problem is a tuple*

$$\langle \mathcal{A}, R \rangle \tag{16.4}$$

where

- $\mathcal{A}$ *is the set of possible arms (i.e., actions, choices the agent can do).*

$$\mathcal{A} = \{a_1, \dots, a_N\}$$

- R *is the set of unknown distributions* $\mathcal{R}(a_i)$ *of the rewards. The rewards* r *are drawn from* $\mathcal{R}(a_i)$*.*

$$r \sim R(a_i)$$

Note that, being draw from a probability distribution, rewards can take values in the interval $[0, 1]$*. The expected reward* $R(a_i)$ *is therefore*

$$R(a_i) = \mathbb{E}_{r \sim \mathcal{R}(a_i)}[r]$$

The interaction between a Multi Armed Bandit agent and the environment works as follows. At each round t

1. The agent selects an arm a_{i_t}.
2. The environment generates a stochastic reward $r_{a_{i_t},t}$ drawn from $\mathcal{R}(r_{a_{i_t}})$
3. The agent updates its information by means of history h_t (i.e., the pulled arm and the received reward).

The final goal of an MAB agent is to maximise the total reward (not the discounted one as we previously analysed) over a finite time horizon T.

$$v = \sum_{t=0}^{T} r_{a_{i_t},t} \tag{16.5}$$

where $r_{a_{i_t},t}$ is the reward (or better said, the realisation of the reward) obtained for pulling arm a_i (i.e., for performing action a_i) at time t. This quantity should possibly converge, for $T \to \infty$, to the arm with the largest expected reward.

16.2.2 Regret

Every reinforcement learning model we saw until now considered only the concept of reward and tried to maximise it. MAB provides a different approach, too. In particular an agent can also consider what it has lost when choosing a particular arm (and not what it has gained). This idea is represented by the regret L_T. Before introducing the regret, let us call R^* the expected reward of the optimal arm a^*

$$R^* = R(a^*) = \max_{a \in \mathcal{A}} R(a) = \max_{a \in \mathcal{A}} \mathbb{E}_{r \sim \mathcal{R}}[r] \tag{16.6}$$

Thanks to the optimal expected reward, at any time step t we compute the expected loss as the difference between the maximum reward possible R^* and the expected reward we get selecting action

$r_{a_i,t}$

$$R^* - R(a_{i,t}) = R^* - \mathbb{E}_{r \sim \mathcal{R}(a_{i,t})}[r] \tag{16.7}$$

Thanks to these values, we can compute the **expected pseudo regret**

Definition 16.4 (Expected Pseudo Regret)**.** *The expected pseudo regret L_T of a MAB agent is*

$$L_T = TR^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right] \tag{16.8}$$

It's important to remember two things about the expected pseudo regret:

- The expected value is taken with respect to the stochasticity of the reward and the randomness of the algorithm used to select the actions.
- The maximisation of the cumulative reward is equivalent to the minimisation of the cumulative regret.

Alternative definition

The regret L_T can be expressed also in another way. To introduce this new representation we have to introduce the **sub-optimality gap**

$$\Delta_i = R^* - R(a_i) \tag{16.9}$$

which returns the average difference between a generic arm a_i and the optimal arm a^*. Since R^* is constant, TR^* can be written as $\sum_{i=1}^{T} R^*$, hence starting from the expected pseudo regret (16.8) we can write

$$L_T = TR^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right] \tag{16.10}$$

$$= \sum_{t=1}^{T} R^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right] \tag{16.11}$$

$$= \sum_{t=1}^{T} R^* - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right] \tag{16.12}$$

$$= \mathbb{E}\left[\sum_{t=1}^{T} R^*\right] - \mathbb{E}\left[\sum_{t=1}^{T} R(a_{i_t})\right] \tag{16.13}$$

$$= \mathbb{E}\left[\sum_{t=1}^{T} R^* - R(a_{i_t})\right] \tag{16.14}$$

The sum of the terms $R^* - R(a_{i_t})$ can be written using the expected number of times $N_T(a_i)$ an arm a_i has been pulled after $T-1$ steps. What we get is

$$L_T = \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \mathbb{E}[N_T(a_i)] R^* - R(a_i) \tag{16.15}$$

$$= \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \mathbb{E}[N_T(a_i)] \Delta_i \tag{16.16}$$

Basically, we have obtained a new definition of regret in terms of the difference between the optimal expected reward and the expected reward of each arm.

$$L_T = \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \mathbb{E}[N_T(a_i)] \Delta_i \tag{16.17}$$

16.2.3 MAB lower bound

From Equation 16.17 we can see that the the more arms are similar in terms of reward (i.e., the more $R(a_i)$ are close, for different a_is), the more the problem is difficult because we can't find a definitely good arm. This result is better explained by the following theorem.

Theorem 16.1 (MAB lower bound, Lai and Robbins)**.** *Given a MAB stochastic problem, any algorithm satisfies*

$$\lim_{T \to \infty} L_T \geq \log T \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \frac{\Delta_i}{KL\big(\mathcal{R}(a_i), \mathcal{R}(a^*)\big)} \tag{16.18}$$

where $KL\big(\mathcal{R}(a_i), \mathcal{R}(a^*)\big)$ is the Kullback-Leibler divergence between the two Bernoulli distribution $\mathcal{R}(a_i)$, $\mathcal{R}(a^*)$.

16.2.4 Pure exploitation algorithm

The easiest way to solve a MAB stochastic problem is by using the pure exploitation algorithm. In this case the agent always chooses the action with the maximum expected reward. In short, the agent, at time t chooses

$$a_{i_t} = \arg\max_{a \in \mathcal{A}} \hat{R}_t(a)$$

where $\hat{R}_t(a)$ is the expected reward for an arm, computed considering all previous steps (i.e., until $t-1$)

$$\hat{R}_t(a) = \frac{1}{N_t(a)} \sum_{j=1}^{t-1} r_{a_{i_j}, j} \mathbf{1}\{a = a_{i_j}\} \tag{16.19}$$

and $\mathbf{1}\{a = a_{i_j}\}$ is the function that returns 1 if $a = a_{i_j}$. This algorithm might not converge to the optimal solution since the agent chooses only the best action. Moreover, we are not considering the uncertainty corresponding to the $\hat{R}_t(a)$ estimate. To solve this problem we have to add a bonus for exploration. In particular there exist two different approaches:

- **Frequentist**. The expected value of the rewards $R(a_1), \dots, R(a_N)$ are unknown parameters and a policy selects at each time step an arm based on the observation history.
- **Bayesian**. The expected value of the rewards $R(a_1), \dots, R(a_N)$ are random variables with prior distributions $f_1, \dots, f_N$. A policy selects at each time step an arm based on the observation history and on the provided priors.

The basic idea is to estimate the value of $R(a_i)$ (computed as the sum of $\hat{R}_t(a)$ and an adjusting term). In particular, we can compute an upper bound for the value of $\hat{R}_t(a)$ that defines how certain are we about the expected reward $\hat{R}_t(a)$. If the uncertainty is high (i.e., the upper and lower bounds are far apart), the algorithm should explore the action associated with the expected reward.

16.2.5 Upper Confidence Bound

The Upper Confidence Bound $U(a_i)$ is an upper value for the expected reward $R(a_i)$ that considers our estimate $\hat{R}_t(a_i)$ and a bound length $B_t(a_i)$

$$U(a_i) = \hat{R}_t(a_i) + B_t(a_i) \geq R(a_i) \tag{16.20}$$

which holds with probability bigger than $1 - \delta$ (with $\delta \in [0, 1]$).

The bound length $B_t(a_i)$ depends on how much information we have on the arm, i.e., the number of times $N_t(a_i)$ we pulled that arm. In particular:

- If $N_t(a_i)$ is small, $U(a_i)$ is large and the estimate value $\hat{R}_t(a_i)$ is uncertain.
- $N_t(a_i)$ is large, $U(a_i)$ is small and the estimate value $\hat{R}_t(a_i)$ is accurate.

To compute the upper bound, we can use the concentration inequality defined by the Höeffding bound

Definition 16.5 (Höeffding bound)**.** *Let $X_1, \dots, X_t$ be independent identically distributed random variables with support in $[0, 1]$ and identical mean $\mathbb{E}[X_i] = X$. Let $\hat{X}_t = \frac{\sum_{i=1}^{t} X_i}{t}$. Then*

$$\mathbb{P}(X > \bar{X}_t + u) \leq e^{-2tu^2} \tag{16.21}$$

The Höeffding bound can be applied to Equation 16.20 to obtain the probability that the actual expected reward $R(a_i)$ is bigger than the upper bound $U(a_i)$ (i.e., that $U(a_i)$ isn't actually an upper bound, since it's smaller than what it's approximating)

$$\mathbb{P}(R(a_i) \geq \hat{R}_t(a_i) + B_t(a_i)) \leq e^{-2N_t(a_i)B_t(a_i)^2} \tag{16.22}$$

Equation 16.22 can be used to practically compute the upper bound. In particular, we have to

1. Pick a probability p that the real value exceeds the bound.

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

2. Solve to find $B_t(a_i)$.

$$B_t(a_i) = \sqrt{\frac{-\log p}{2N_t(a_i)}}$$

3. Reduce the value of p over time (e.g., $p = t^{-4}$). Basically, we want the probability that we are wrong about the upper bound decrease with time.

$$B_t(a_i) = \sqrt{\frac{4 \log t}{2N_t(a_i)}} \tag{16.23}$$

4. Ensure to select the optimal action as the number of samples increases.

$$\lim_{t \to \infty} B_t(a_i) = 0 \Rightarrow \lim_{t \to \infty} U_t(a_i) = R(a_i)$$

UCB1

Now that we know how to compute the upper bound $U_t(a_i)$, thanks to the estimate $\hat{R}_t(a_i)$ (16.19) and $B_t(a_i)$ (16.23), we can introduce the algorithm to solve a MAB problem. The algorithm works as follows. For each time step t

1. Compute, for each $a_i \in \mathcal{A}$, the estimate of the reward $\hat{R}_t(a_i)$

$$\hat{R}(a) = \frac{\sum_{j=1}^{t-1} r_{a_{i,j},j} \mathbf{1}\{a = a_{i_j}\}}{N_t(a)}$$

2. Compute, for each $a_i \in \mathcal{A}$, the bound length $B_t(a_i)$

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

3. Play arm a_i that maximises the sum between $\hat{R}$ and B.

$$a_{i,t} = \arg\max_{a_i \in \mathcal{A}} \left\{ \hat{R}_t(a_i) + B_t(a_i) \right\}$$

The maximum upper bound for the expected regret we can get from UCB1 is given by the following theorem.

Theorem 16.2 (UCB1 upper bound, Auer-Cesa-Bianchi)**.** *At finite time T, the expected total regret of the UCB1 algorithm applied to a stochastic MAB problem is*

$$L_T \leq 8 \log T \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \frac{1}{\Delta_i} + \left(1 + \frac{\pi^2}{3}\right) \sum_{a_i \in \mathcal{A}: \Delta_i > 0} \Delta_i \tag{16.24}$$

16.2.6 Thompson sampling

Thompson sampling is a Bayesian technique for solving a MAB problem. The general algorithm is

1. Consider a Bayesian prior for each arm $f_1, \dots, f_N$. In particular, the prior should be initially distributed as a $Beta(1, 1)$ (i.e., the uniform distribution).
2. At each round t, sample from each one of the distributions $\hat{r}_1, \dots, \hat{r}_N$ of the rewards. Note that the rewards should have a Bernoulli distribution.
3. Pull the arm $a_{i,t}$ with the highest sampled value $i_t = \arg\max_i \hat{r}_i$.
4. Update the prior incorporating the new information. In particular, the prior should be updated as follows:
   - If a success occurs, we should set $f_i(t+1) = Beta(\alpha_t + 1, \beta_t)$.
   - If a failure occurs, we should set $f_i(t+1) = Beta(\alpha_t, \beta_t + 1)$.

As for other algorithms, we have a theorem that defines what the upper bound for the expected reward is.

Theorem 16.3 (Thompson sampling upper bound, Kaufmann and Munos)**.** *At time T, the expected total regret of the Thompson sampling algorithm applied to a stochastic MAB problem is*

$$L_T \leq \mathcal{O}(\sum_{a_i \in \mathcal{A}:\Delta_i > 0} \frac{\Delta_i}{KL\big(\mathcal{R}(a_i), \mathcal{R}(a^*)\big)}(\log T + \log\log T)) \tag{16.25}$$

16.3 Adversarial Multi Armed Bandit

An adversarial Multi Armed Bandit is defined as follows.

Definition 16.6 (Adversarial Multi Armed Bandit)**.** *An adversarial Multi Armed Bandit is a tuple A Multi Armed Bandit problem is a tuple*

$$\langle \mathcal{A}, R \rangle \tag{16.26}$$

where

- *$\mathcal{A}$ is the set of possible arms (i.e., actions, choices the agent can do).*

$$\mathcal{A} = \{a_1, \dots, a_N\}$$

- *$\mathcal{R}$ is the reward vector for which the realisation $r_{a_i,t}$ is decided by an adversary player at each turn.*

The principle of working of a Multi Armed Bandit agent is the following. At each time step t:

1. The agent selects a single arm a_{i_t}.
2. The adversary chooses reward $r_{a_i,t}$.
3. The agent gets reward $r_{a_{i_t},t}$.

The goal of the agent is to maximise the cumulative reward over a time horizon T

$$v = sum_{t=1}^{T} r_{a_{i_t},t}$$

In this setting we assume that the adversary is oblivious (or non-adaptive), i.e., it selects the rewards in advance knowing the agent's algorithm but not the actual action realisation.

16.3.1 Pseudo regret

For adversarial Multi Armed Bandit we can't compare the cumulated regret with the optimal one since the regret is chosen by an adversary and not by a random distribution. Given that we have an adversary we can't use a deterministic algorithm either because otherwise the adversary would know what we play every time. For this setting we should use **pseudo regret** instead.

Definition 16.7 (Pseudo regret). *The pseudo regret L_T for an adversarial MAB problem is*

$$L_T = \max_{a_i \in \mathcal{A}} \mathbb{E}\left[\sum_{t=1}^{T} r_{a_i,t}\right] - \mathbb{E}\left[\sum_{t=1}^{T} r_{a_{i_t},t}\right] \tag{16.27}$$

Basically, we are comparing the expected value of the best expected value of the best constant action with the expected value of the rewards we obtained. The following theorem gives a lower bound for the pseudo regret.

Theorem 16.4 (Pseudo regret lower bound). *Let sup be the supremum over all distribution of rewards such that, for $i \in \{1, \dots, N\}$ the rewards $r_{i,1}, \dots, r_{i,T}$ and $r_{i,j}$ are independent identically distributed, and let inf be the infimum over all forecasters. Then*

$$\textit{infsup}\ L_T \geq \frac{1}{\sqrt{20}}\sqrt{TN} \tag{16.28}$$

16.3.2 EXP3

EXP3 is an algorithm used for solving adversarial MAB problems. At each time t, an agent chooses an arm a_i, with probability

$$\pi_t(a_i) = (1-\beta)\frac{w_t(a_i)}{\sum_j w_t(a_j)} + \frac{\beta}{N} \tag{16.29}$$

where $w_t(a_i)$ is updated at every time step as follows

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

The following theorem defines the upped bound for EXP3 expected regret.

Theorem 16.5 (EXP3 upper bound). *At time t, the pseudo regret of EXP3 applied to an adversarial MAB problem with $\beta = \eta = \sqrt{\frac{N \log N}{T(e-1)}}$ is*

$$L_T \leq \mathcal{O}(\sqrt{TN \log N}) \tag{16.30}$$

where the expectation is taken with respect to both the random generalisation and the internal randomisation of the forecaster.

Appendix A

Probability

A.1 Bayes

The Bayes' rule allows to compute (or better to update) the probability of an event H (also called hypothesis), knowing that an event E happened.

$$p(H|E) = \frac{p(H)p(E|H)}{p(E)} \tag{A.1}$$

$$= \frac{p(H)p(E|H)}{p(H)p(E|H) + p(\neg H)p(E|\neg H)} \tag{A.2}$$

$$= \frac{p(H \wedge E)}{p(E)} \tag{A.3}$$

where we call

- $p(H)$ the **prior**.
- $p(H|E)$ the **posterior**.
- $p(E|H)$ the **likelihood**.

A.2 Joint distributions

Given two random variables A and B,

- If the variables are independent then the joint distribution is the product between the single distributions.

$$p(A, B) = p(B, A) = P(A) \cdot P(B) \tag{A.4}$$

- If the variables are dependent the joint distribution can be computed as

$$p(A, B) = p(B, A) \tag{A.5}$$

$$= p(A) \cdot p(B|A) \tag{A.6}$$

$$= p(B) \cdot p(A|B) \tag{A.7}$$

A.3 Conjugate prior

In Bayesian probability theory, if the posterior distribution $p(t|x)$ is in the same probability distribution family as the prior probability distribution $p(t)$, the prior and posterior are then called **conjugate distributions**, and the prior is called a conjugate prior for the likelihood function $p(x|t)$.

A.4 Gaussian distribution

X is a normal random variable of parameters μ and σ^2 if

$$f(x|\mu,\sigma^2) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^2}} \tag{A.8}$$

and we write

$$X \sim \mathcal{N}(\mu,\sigma^2) \tag{A.9}$$

Appendix B

Algebra

B.1 Notation

Let us introduce notation the notation we use.

Row vector A row vector with D elements is represented as

$$(v_0, \ldots, v_{D-1})$$

Column vector A column vector $\mathbf{v}$ with D elements is represented as

$$\mathbf{v} = (v_0, \ldots, v_{D-1})^T$$

Dimensions When describing a matrix, we will represent its dimensions as

$$R \times C$$

where

- R is the number of **rows** of the matrix.
- C is the number of **columns** of the matrix.

B.2 Positive and semi-positive definite matrix

B.2.1 Positive definite matrix

A matrix A is said to be positive definite, if

$$\mathbf{x}^T A \mathbf{x} > 0$$

for all possible vectors $\mathbf{x} \in \mathbb{R}^N \setminus \{0\}$.

B.2.2 Semi-positive definite matrix

A matrix A is said to be semi-positive definite, if

$$\mathbf{x}^T A\mathbf{x} \geq 0$$

for all possible vectors $\mathbf{x} \in \mathbb{R}^N \setminus \{0\}$.

B.3 Eigenvectors and eigenvalues

Given a square matrix $A \in \mathbb{R}^{N\times N}$, its eigenvalues $\lambda_1, \ldots, \lambda_N$ and corresponding eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_N$ are defined from the eigenvector equations:

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{B.1}$$

for each $i \in \{1, \ldots, N\}$. Intuitively, an eigenvector $\mathbf{v}_i$ is a direction in the space that is only stretched when the transformation A is applied. The corresponding eigenvalue λ_i is the factor this vector is stretched. Using the matricial formulation, we have that to find the generic pair $(\lambda, \mathbf{v})$ for which it is possible to solve the following:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{B.2}$$

$$(A - \lambda\mathbf{I}_N)\mathbf{v} = 0 \tag{B.3}$$

where $\mathbf{I}_N$ is the identity matrix of order N. The previous problem has a non-null solution only if the determinant of the matrix $A - \lambda\mathbf{I}_N$ is non-null.

Properties

The eigenvalues $\lambda_1, \ldots, \lambda_N$ of a matrix A satisfy the following properties:

- The rank of the matrix A is equal to the number of nonzero eigenvalues.
- The determinant of A is equal to the product of its eigenvalues.

$$|A| = \prod_{i=1}^{N} \lambda_i$$

- The trace of A is equal to the sum of its eigenvalues.

$$Tr(A) = \sum_{i=1}^{N} \lambda_i$$

- If A is positive definite, all eigenvalues are positive.

$$\lambda_i > 0 \quad \forall i$$

- If A is semi-positive definite, all its eigenvalues are non-negative.

$$\lambda_i \geq 0 \quad \forall i$$

B.4 Matrix properties

Here's a list of useful matrix properties:

- $$(A + B)^T = A^T + B^T \tag{B.4}$$
- $$(AB)^T = B^T A^T \tag{B.5}$$

Definitions

Accuracy, 45
Action-value function, 110
Adversarial Multi Armed Bandit, 151
Arm, 144

Bellman operator, 112, 113
Bellman optimality operator, 115
Bias, 47

Dichotomy, 72
Discounted reward, 107

Efficiently PAC-learnable, 71
Expected Pseudo Regret, 147

Fixed point, 112

Gaussian process, 89

History, 103
Homogeneous kernel, 78
Höeffding bound, 149

Kernel, 77

Machine learning, 2
Markov assumption, 105
Markov Decision Process, 105
Multi Armed Bandit, 146
Multi Armed Bandit problem, 145

Non-parametric algorithm, 7

Optimal action-value function, 113
Optimal state-value function, 113

PAC-learnable, 71
Parametric algorithm, 6
Policy, 108
Policy evaluation, 110

Radial basis function, 84

Shattering, 72
State-value function, 110
Stationary kernel, 78
Stationary stochastic Markovian policy, 109

Valid kernel, 77
Variance, 48
VC-Dimension, 73

Theorems and principles

ε-greedy exploration improvement, 137

EXP3 upper bound, 152

Gauss-Markov, 23
GLIE Monte-Carlo convergence, 138
Greedy in the Limit of Infinite Exploration, 137

MAB lower bound, Lai and Robbins, 148
Mercer, 79

No free lunch, 45
No free lunch corollary, 46

Optimal policy, 114

PAC learning in version spaces, 68
Perceptron convergence theorem, 35
Policy improvement, 120
Pseudo regret lower bound, 152

SARSA convergence, 139
Solution for linear programming, 124
Sutton hypothesis, 106

Thompson sampling upper bound, Kaufmann and Munos, 151

UCB1 upper bound, Auer-Cesa-Bianchi, 150

Value iteration convergence, 122
Value iteration error, 122

Equations

Bias variance decomposition, 47

Classification maximum likelihood, 37

Gaussian kernel, 81
Gaussian process distribution, 89
Gradient descend for linear models, 19

Homogeneous kernel, 79

k-fold loss, 57
Kernel, 77

Lasso complexity loss, 25
Leave One Out average loss, 57
Linear kernel, 78
Linear model for classification, 30
Linear model for regression, 13
Linear model for regression in matrix form, 14
Linear model for regression with basis functions, 16
Linear model for regression with basis functions in matrix form, 16
Linear regression model for kernel methods, 83
Logistic regression gradient, 39

Loss function for lasso, 25

MAB bound length, 149
Margin for Support Vector Machines, 95
Model for support vector machines, 93

Nadaraya-Watson model, 87
Naive Bayes model, 40
Non-Euclidean Gaussian kernel, 81

Optimal parameter vector for Ridge regression, 24
Optimal parameters for linear regression, 18

PAC bound for the error, 70
PAC bound for the number of samples, 70
PAC learning in version spaces, 68

Radial basis function, 84

Softmax transformation, 39
Stationary kernel, 78
Sum of Squared Errors, 17

Upper confidence bound, 149

Variance, 48