A short introduction to Machine Learning

Mateo Myftaraj[1]

May 2, 2022

[1]https://github.com/MatMyfta

Contents

I Basics **1**

1 Introduction **3**
1.1 When to use Machine Learning 4
1.2 Definition 6
1.2.1 Formally 6
1.3 Machine Learning vs Other Disciplines 7
1.3.1 Artificial Intelligence 7
1.3.2 Deep Learning 8
1.3.3 Data Mining 8
1.4 Back to machine learning 9

2 Data, Features, and Models **11**
2.1 The learning process 11
2.2 Data 12
2.2.1 Data Split 13
2.2.2 Data generating distribution 15
2.3 Feature 16
2.3.1 Characteristics of good features 17
2.4 Types of learning 17
2.4.1 Supervised learning 18
2.4.2 Unsupervised learning 21
2.4.3 Reinforcement learning 23
2.4.4 Other Learning Types 24

3 Generalization error, Models, Hypothesis space **27**
3.1 Task 27
3.2 Model and hypotesis space 28
3.2.1 The ideal target 30
3.2.2 The feasible target 30
3.2.3 The actual target 30
3.3 Error function 30
3.3.1 Overfitting 32
3.3.2 Underfitting 33
3.3.3 Estimate the generalization error 34
3.3.4 Improve the generalization 34

II Supervised Learning **37**

4 k-Nearest Neighbor **39**

4.1 Introduction . . . 39
4.2 Algorithm . . . 40
4.2.1 Euclidean distance . . . 40
4.2.2 Algorithm . . . 41
4.3 Decision boundaries . . . 41
4.4 Choosing k . . . 41
4.5 KNN in practice . . . 42
4.5.1 Recommender Systems . . . 42
4.5.2 Weighted k-Nearest Neighbor . . . 42

5 Linear Models **45**
5.1 Bias . . . 45
5.1.1 Define a line . . . 46
5.1.2 Classifying with linear models . . . 47
5.2 Online learning . . . 47
5.2.1 Tasks and applications . . . 47
5.3 Perceptron . . . 48
5.3.1 Convergence and number of iterations . . . 48
5.3.2 Algorithm . . . 49

6 Decision trees **51**
6.1 How decision trees work . . . 51
6.1.1 Inference . . . 52
6.2 Decision tree learning algorithm . . . 52
6.2.1 Growing a leaf . . . 53
6.2.2 Growing a node . . . 53
6.3 About split selection . . . 54
6.4 Leaf predictions . . . 54
6.5 Impurity measures for classification . . . 54
6.6 Impurity measures for regression . . . 55
6.7 Split functions . . . 55
6.7.1 Discrete nominal features . . . 55
6.7.2 Ordinal features . . . 56
6.7.3 Oblique . . . 56
6.8 Overfitting . . . 56
6.8.1 Pruning . . . 56
6.9 Random forests . . . 57

7 Multi-class classification **59**
7.1 Extension from binary . . . 59
7.1.1 k-nearest neighbours . . . 60
7.1.2 Decistion trees . . . 60
7.1.3 Perceptron . . . 60
7.2 One vs All . . . 60
7.3 All vs All . . . 61
7.4 OVA vs AVA . . . 62
7.5 Evaluation . . . 63
7.5.1 Confusion matrix . . . 63

8 Gradient Descent **65**

8.1 Notation . . . 65
8.2 The optimization framework for Linear Models . . . 66
8.3 Convex Surrogate Loss Functions . . . 66
8.4 Gradients, math review . . . 67
8.5 Optimization with gradient descent . . . 68
8.5.1 Algorithm . . . 68

9 Regularization **71**
9.1 Regularizers . . . 71
9.2 p-norm . . . 72
9.3 Minimizing with a regularizer . . . 72
9.3.1 Model-based machine learning . . . 73

10 Support Vector Machines **75**
10.1 Large margin classifiers . . . 75
10.1.1 Support vectors . . . 76
10.1.2 Maximizing the margin . . . 76
10.2 Soft margin classification . . . 77
10.3 Non linearly separable data . . . 78
10.3.1 Dual problem . . . 78
10.3.2 Kernel trick . . . 79

11 Ranking **81**
11.1 Multilabel classification . . . 82
11.2 Ranking problems . . . 82
11.2.1 Black box approach . . . 82
11.2.2 Preference function . . . 83
11.3 Algorithm . . . 83
11.4 Bipartite Ranking . . . 84
11.5 Weighted binary classification . . . 84
11.5.1 ω-Ranking . . . 85

12 Neural Networks **87**
12.1 From Perceptron to Deep Networks . . . 88
12.1.1 Backpropagation . . . 89
12.2 Feedforward Networks . . . 90
12.2.1 Training . . . 91
12.3 Backpropagation . . . 93
12.3.1 Step 1: Feedforward propagation . . . 94
12.3.2 Step 2: Compute error and train . . . 95
12.3.3 Step 3: Backpropagation . . . 95
12.4 Training a Neural Network . . . 95
12.4.1 Gradient Descent . . . 96
12.5 Convolutional Neural Networks . . . 98
12.5.1 Convolutional layers . . . 98
12.6 Other Neural Networks . . . 101
12.6.1 Recurrent Neural Networks . . . 102
12.6.2 Autoencoders . . . 103

III Unsupervised Learning 105

13 Unsupervised Learning 107
13.1 Tasks in depth . . . 107
13.1.1 Dimensionality Reduction . . . 107
13.1.2 Clustering . . . 108
13.1.3 Density Estimation . . . 109
13.2 Principal Component Analysis . . . 109
13.2.1 Variance Along Unit Direction . . . 111
13.2.2 Eigenvalue Decomposition . . . 111
13.2.3 PCA using Eigenvalue Decomosition . . . 112
13.2.4 PCA Using Singular Value Decomposition (SVD) . . . 113
13.2.5 Dimensionality Reduction Using PCA . . . 113
13.2.6 Alternative Interpretation . . . 113
13.2.7 Kernel PCA . . . 114
13.3 k-Means Clustering . . . 115
13.3.1 Optimization Algorithm . . . 115
13.3.2 Properties . . . 115

14 Clustering 117
14.1 k-Means . . . 117
14.1.1 Issues . . . 117
14.1.2 Properties . . . 118
14.2 Issues for Clustering . . . 119
14.2.1 Types of Clutering Algorithms . . . 119
14.3 EM Clustering . . . 120
14.3.1 Soft Clustering in EM Clustering . . . 120
14.3.2 Mixture of Gaussians . . . 121
14.3.3 Expectation Maximization . . . 122
14.4 Other Clustering Algorithms . . . 122
14.4.1 Spectral Clustering . . . 123
14.4.2 Hierarchical Clustering . . . 123

15 Deep Generative Models 125
15.1 Density Estimation . . . 125
15.2 Variational AutoEncoder (VAE) . . . 126
15.2.1 AutoEncoder . . . 126
15.2.2 AutoEncoder in Generative Models . . . 127
15.2.3 Variational Upper Bound . . . 128
15.2.4 Conditional VAE . . . 129
15.2.5 Issues with VAEs . . . 129
15.3 Generative Adversarial Networks (GAN) . . . 129
15.3.1 Objective . . . 130
15.3.2 Issues with GANs . . . 131
15.3.3 More GANs . . . 131

IV Reinforcement Learning 133

16 Reinforcement Learning 135

16.1 The Idea 135
16.2 Markov Decision Process 137
16.2.1 Definition 137
16.2.2 MDP Loop 138
16.2.3 Objective 138
16.2.4 Reinforcement Learning vs Supervised Learning . . 139
16.3 Value Based Methods 140
16.3.1 Q-Learning 140
16.3.2 Deep Q-Learning 143
16.4 Policy Gradient Methods 143
16.4.1 Objective Function 144
16.4.2 Reinforce Algorithm 145

Part I

Basics

1

Introduction

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience and by the use of data. It is a branch of computer science which focuses on the use of data and algorithms to imitate the way that humans learn, graudally improving its accuracy[1]. Machine learning is an important component of the growing field of data science, through the use of statistical methods, machine learning algorithms build a model based on sample data, known as *training data*, in order to make predictions or decisions without being explicitly programmed to do so.

Learning algorithms work on the basis that strategies, algorithms, and inferences that worked well in the past are likely to continue working well in the future. These inferences can be obvious, such as "since the sun rose every morning for the last 10,000 days, it will probably rise tomorrow morning as well". They can be nuanced, such as "$X\%$ of families have geographically separate species with color variants, so there is a $Y\%$ chance that undiscovered black swans exist".

Machine learning programs can perform tasks without being explicitly programmed to do so. It involves computers learning from data provided so that they carry out certain tasks. For simple tasks assigned to computers, it is possible to program algorithms telling the machine how to execute all steps required to solve the problem at hand; on the computer's part, no learning is needed. For more advanced tasks, it can be challenging for a human to manually create the needed algorithms. In practice, it can turn out to be more effective to help the machine develop its own algorithm, rather than having human programmers specify every needed step.

The discipline of machine learning employs various approaches to teach computers to accomplish tasks where no fully satisfactory algorithm is

[1] We will go much more in depth to the details of the term **accuracy** later.

available. In cases where vast numbers of potential answers exist, one approach is to label some of the correct answers as valid. This can then be used as training data for the computer to improve the algorithm(s) it uses to determine correct answers. For example, to train a system for the task of digital character recognition, the MNIST dataset of handwritten digits has often been used.

Machine Learning allows computers to acquire **knowledge**. Knowledge is acquired through **algorithms** by learning and inferring from **data**, and it is presented by a **model** that is used on future data.

1.1 When to use Machine Learning

Machine learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks. It is important to remember that ML is not a solution for every type of problem. There are certain cases where robust solutions can be developed without using ML techniques. For example, you don't need ML if you can determine a target value by using simple rules, computations, or predetermined steps that can be programmed without needing any data-driven learning. In brief, Machine Learning can be involved in the following cases:

- when human expertise does not exist;
- when humans cannot explain their expertise;
- when models must be customized;
- when models are based on huge amounts of data.

When human expertise does not exist

It may appear obvuois, when the human expertise lacks, then he cannot state to the machine the commands to execute. An examples is the Machine Learning techniques used for navigating to Mars: risk to human astronauts and interplanetary distance causing slow and limited communication drives scientists to pursue an autonomous approach to exploring distant planets, such as Mars. A portion of exploration of Mars has been conducted through the autonomous collection and analysis of Martian data by spacecraft such as the Mars rovers and the Mars Express Orbiter. The autonomy used on these Mars exploration spacecraft and on Earth to analyze data collected by these vehicles mainly consist of machine learning.

When humans cannot explain their expertise

In this case it may be possible to have the necessary human expertise, but it is not possible to explain them to a machine. In order to make this more concrete, imagine that you need to do an algorithm that *recognises*

when a user says some words, it is very simple for a human to handle this job because there is a lot of human expertise, but it is infeasible to write a program (without Machine Learning) that can categorically recognise words. Speech recognition is a sub field of machine learning, and it involves various methodologies and technologies which allow recognising what the user is saying; while in voice recognition the machine learns to determine who is speaking.

When models must be customized

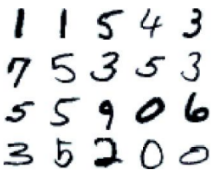
Sometimes we don't have a single solution, but we need to adapt the solution based to a moltitude variables. An example for this kind of problems is the personalized medicine. The purpose of personalized medicine is to select and deliver patient-specific treatments to achieve the best possible outcome. The challenge lies in identifying an optimum treatment as the number of possible predictors of good response like genetic and other biomarkers, and the option of treatments is increasing.

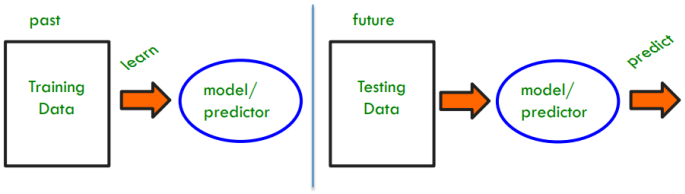
In addition to this, as most clinical trials are based upon average treatment effects, similar medicines become non-responsive for some patients and responsive for some other patients.

When models are based on huge amounts of data

As of 2021, 20 years have passed since the landmark completion of the draft human genome sequence. This milestone has led to the generation of an extraordinary amount of genomic data. Estimates predict that genomics research will generate between 2 and 40 exabytes of data within the next decade. DNA sequencing and other biological techniques will continue to increase the number and complexity of such data sets. This is why genomics researchers need AI/ML-based computational tools that can handle, extract and interpret the valuable information hidden within this large trove of data.

ML can be used for **recognizing** patterns, for example handwritten digits, facial and medical images; it can be used for **generating** patterns, for example for generating images or motion sequences; it can be used for recognizing anomalies, for example unusual credit card transactions or unusual patterns of sensor readings; or it can be used for **predictions**, for example it can be used to predict future stock prices or currency exchange rates, autonomous driving or to predict best moves in games.





1.2 Definition

Machine learning studies computer algorithms for learning to do stuff. We might, for instance, be interested in learning to complete a task, or to make accurate predictions, or to behave intelligently. The learning that is being done is always based on some sort of observations or data, such as examples, direct experience, or instruction. So in general, machine learning is about learning to do better in the future based on what was experienced in the past. Some more technical definitions have been provided during the development of machine learning:

- It is concerned with the automatic discovery of regularities in data through the use of computer algorithms and with the use of these regularities to take actions. — Chrisopher M. Bishop
- The goal of machine learning is to deveop methods that can automatically detect patterns in data, and then to use the uncovered patterns to predict future data or other outcomes of interest. — Kevin P. Murphy
- Machine learning is about predicting the future based on the past. — Hal Daume III
- A computer program is said to learn form **experience** E with resepct to some class of **tasks** T and performance **measure** P, if its performance at tasks in T, as measured by P, improves with experience E. — T. Mitchell

1.2.1 Formally

Mathematically speaking, machine learning is the study of algorithms that improve their performance P at some task T with experience E. A well-defined learning task is given by a triplet $\langle T, P, E \rangle$.

Example 1.2.0 — Tasks for machine learning applications

For handwritten word recognition, the learning task $\langle T, P, E \rangle$ is defined as follows:
T: Recognizing handwritten words;
P: Percentage of words correctly classified;
E: Database of human-labeled images of handwritten words.

For the spam email recognition, the learning task $\langle T, P, E \rangle$ is defined as follows:
T: Categorizing email messages as spam or legitimate;
P: Percentage of email messages correctly classified;
E: Database of emails, some with human-given labels.

1.3 Machine Learning vs Other Disciplines

Modern day machine learning has two objectives, one is to classify data based on models which have been developed, the other purpose is to make predictions for future outcomes based on these models. A hypothetical algorithm specific to classifying data may use computer vision of moles coupled with supervised learning in order to train it to classify the cancerous moles. Where as, a machine learning algorithm for stock trading may inform the trader of future potential predictions.

1.3.1 Artificial Intelligence

Artificial intelligence (AI) is intelligence demonstrated by machines, as opposed to natural intelligence displayed by animals including humans. Leading AI textbooks define the field as the study of "intelligent agents": any system that perceives its environment and takes actions that maximize its chance of achieving its goals.

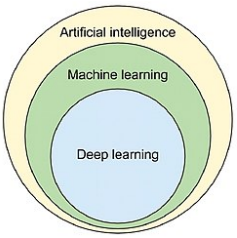


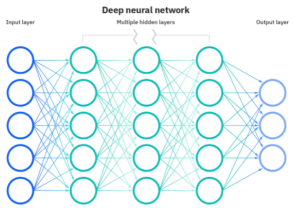
Figure 1.1: Machine Learning as subfield of AI

Some popular accounts use the term "artificial intelligence" to describe machines that mimic "cognitive" functions that humans associate with the human mind, such as "learning" and "problem solving", however, this definition is rejected by major AI researchers.

AI applications include advanced web search engines (e.g., Google), recommendation systems (used by YouTube, Amazon and Netflix), understanding human speech (such as Siri and Alexa), self-driving cars (e.g.,

Tesla), automated decision-making and competing at the highest level in strategic game systems (such as chess and Go). As machines become increasingly capable, tasks considered to require "intelligence" are often removed from the definition of AI, a phenomenon known as the AI effect. For instance, optical character recognition is frequently excluded from things considered to be AI, having become a routine technology.

1.3.2 Deep Learning



Deep Learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Artificial neural networks (ANNs) were inspired by information processing and distributed communication nodes in biological systems. ANNs have various differences from biological brains. Specifically, artificial neural networks tend to be static and symbolic, while the biological brain of most living organisms is dynamic (plastic) and analogue.

The adjective "deep" in deep learning refers to the use of multiple layers in the network. Early work shows that a linear perceptron[2] cannot be a universal classifier, but that a network with a nonpolynomial activation function with one hidden layer of unbounded width can. Deep learning is a modern variation which is concerned with an unbounded number of layers of bounded size, which permits pratical application and optimized implementation, while retaining theoretical universality under mild conditions. In deep learning the layers are also permitted to be heterogeneous and to deviate widely from biologically informed connectionist models, for the sake of efficiency, trainability and understandability, whence the "structured" part.

In deep learning, each level learns to transform its input data into a slightly more abstract and composite representation. In an image recognition application, the raw input may be a matrix of pixels; the first representational layer may abstract the pixels and encode edges; the second layer may compose and encode arrangements of edges; the third layer may encode a nose and eyes; and the fourth layer may recognize that the image contains a face. Importantly, a deep learning process can learn which features to optimally place in which level on its own. This does not eliminate the need for hand-tuning; for example, varying numbers of layers and layer sizes can provide different degrees of abstraction.

1.3.3 Data Mining

Data mining is a process of searching, extracting and analyzing (that may include) discovering various types of text graphic patterns (as calli-

[2]**Perceptron:** is an algorithm for supervised learning of binary classifiers — a function which can decide whether or not an input belongs to some specific class.

graphic for example), language and literary figures, stylistics, in large amounts of textual or mixed visual and textual data sets, that also involvs methods at the intersection of machine learning, formal linguistics analyses as textual statistics, and database systems.

The actual data mining task is the semi-automatic or automatic analysis of large quantities of textual databases to extract previously not well known or entirely unknown, interesting or surprising language or information patterns such as groups of textual data records, but also unusual records (sometimes computer detected as anomaly), but also associations or dependencies (rule of pattern association and sequential pattern mining). This usually involves using database techniques such as using of spatial indices. These textual or information patterns can then be seen as summarizing of the input data, and may be used in further analysis or, for example, in machine learning for analysis involving predictions.

1.4 Back to machine learning

The general set up of predicting the future based on the past is at the core of most machine learning. The objects that our algorithm will make predictions about are **examples**. Let's consider for example the Netflix recommender system: when you watch a movie and state that you liked (or disliked) it, Netflix takes this as an example, and the like or dislike as a label that will make the algorithm learn from this example.

To make this concrete, Figure 1.2 shows the general framework of induction. We are given **training data** on which our algorithm is expected to learn. This training data is the examples that Alice observes, or the historical rating data for the recommender system. Based on this training data, our learning algorithm induces a function f that will map a new example to a corresponding prediction.

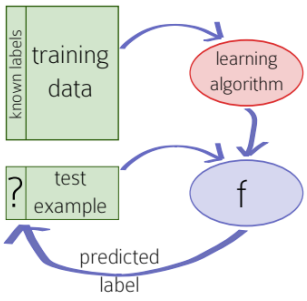


Figure 1.2

We want our algorithm to be able to make lots of predictions, so we refer to the collection of examples on which we will evaluate our algorithm as the **test set**. The test set is a closely guarded secret: it is the final exam on which our learning algorithm is being testes.

The goal of inductive machine learning is to take some training data and use it to induce a function f. This function f will be evaluated on the test data. The machine learning algorithm has succeeded if its performance on the test data is high.

Exercises

1. When is Machine Learning recommended and when it is not? Give some example.
2. Give some example of Machine Learning algorithms on real applications.
3. Give the formal definition of Machine Learning and provide some example of task.
4. Provide the definition of Artificial Intelligence and Deep Learning enphasizing the differences from Machine Learning.
5. What are the examples?
6. How are the training data and the test data used?

2

Data, Features, and Models

> *"The goal of machine learning is to develop methods that can automatically detect patterns in data, and then to use the uncovered patterns to predict future data or other outcomes of interest."*
>
> – Kevin P. Murphy

At a basic level, machine learning is about predicting the future based on the past. For instance, you might wish to predict how much a user Alice will like a movie that she hasn't seen, based on her ratings of movies that she has seen. This prediction could be based on many factors of the movies; in general, this means making informed guesses about some unobserved property of some object, based on observed properties of that object.

2.1 The learning process

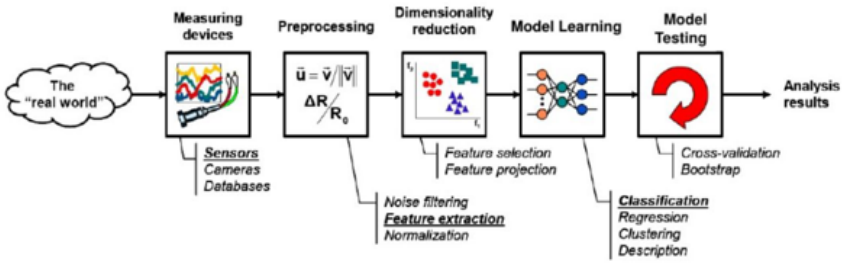


Figure 2.1: Learning process scheme

From the data we acquire knowledge, which is represented by a model, the model is used on future data.

In Figure 2.1 we have a simplified pipeline of the learning process, divided by stages.

The **measurement** can come from different sensors or database, these data are obtained in different ways, and then they are preprocessed.

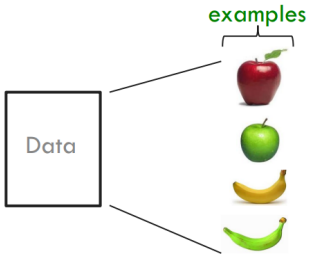
The **preprocessing** depends on the application, for example it can be noise filtering for measurements done when working with sensonrs. We need to transform our data in a set of vectore, called *feature*, this operation is called *feature extraction*. It is common, in many application, to do a normalization, which can be achieved in different ways.

Usually, feature extracted from the data are not directly fed into the model, but there is a selection (or projection) of the vectors to **reduce their dimensionality** and maintaining only useful information. Then, the machine learning algorithm is applied, it can differ depending on the different task of the applcation. Once we have outputted our model from the chosen algorithm, the testing phase is performed.

2.2 Data

Data are individual facts, statistics, or items of information, often numeric. In a more technical sense, data are a set of values of qualitative or quantitative variables about one or more persons or objects, while a *datum* is a single value of a single variable.

Data are measured, collected, reported, and analyzed, and used to create data visualizations such as graphs, tables or images. Data as a general concept refers to the fact that some existing information or knowledge is represented or coded in some form suitable for better usage or processing.



We want to have a single way to treat our information and the same approach to represent the data. For each example in my training set, we have to translate these examples in terms of feature. Feature is usually a set of numbers, one example will be represented by a vector of cardinality N.

Raw data is a collection of numbers or characters before it has been *cleaned* and corrected by researchers. Data processing commonly occurs by stages, and the processed data from one stage may be considered the raw data of the next stage.

Data is the most important part of all Data Analytics, Machine Learning, Artificial Intelligence. Without data, we can't train any model and all

modern research and automation will go in vain.

Information is data that has been interpreted and manipulated and has now some meaningful inference for the user.

Knowledge is the combination of inferred information, experiences, learning, and insights. Results in awareness or concept building for an individual organization. Data can be split into:

- **Training data**: the part of data we use to train our model. This is the data that the model actually sees (both input and output) and learns from.
- **Validation data**: the part of data that is used to do a frequent evaulation of the mode, fit on the training dataset along with improving involved hyperparameters (initially set parameters before the model begins learning). This data plays its part when the model is actually training.
- **Testing data**: once our model is completely trained, testing data provides an unbiased evaluation. When we feed in the inputs of testing data, our model will predict some values. After prediction, we evaluate our model by comparing it with the actual output present in the testing data. This is how we evaluate and see how much our model has learned from the experiences feed in as training data, set at the time of training.

2.2.1 Data Split

Data is the information about the problem to solve in the form of a distribution, for classification and regression: $p_{\text{data}} \in \Delta(X \times Y)$, for density estimation, clustering and dimensionality reduction: $p_{\text{data}} \in \Delta(X)$. The data distribution p_{data} is typically unknown, but we can sample from it.

In machine learning, a common task is the study and construction of algorithms that can learn from and make predictions on data. Such algorithms function by making data-driven predictions or decisions, through building a mathematical model from input data. These input data used to build the model are usually divided in multiple data sets.

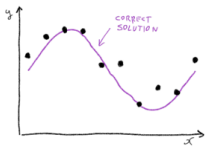
Training set

The model is initially fit on a **training data set**, which is a set of example used to fit the parameters of the model. The model is trained on the training set using a supervised learning method. In practice, the training set often consists of pairs of an input vector and the corresponding output vector, where the answer key is commonly denoted as the *target*. The current model is run with the training set and produces a result, which is then compared with the target, for each input vector in the training set. Based on the result of the comparison and the specific learning algorithm being used, the parameters of the model are adjusted. The model fitting can include both variable selection and parameter estimation.

A training set is a set of examples used during the learning process and is used to fit the parameters of, for example, a classifier.

For classification tasks, a supervised learning algorithm looks at the training set to determina, or learn, the optimal combinations of variables that will generate a good predictive model. The goal is to produce a trained (fitted) model that generalizes well to new, unknown data. The fitted model is evaluated using new examples from the held-out datasets to estimate the model's accuracy in classifying new data. To reduce the risk of issues such as over-fitting, the examples in the validation and test datasets should not be used to train the model.

Example 2.2.0 — Training set example



Given the data $D_n = \{(x_1, y_1), ..., (x_n, y_n)\}$ generated from $\sin(2\pi x)$+noise.

The training set is made of pair, and is represented by $n = 10$ black dots. In the left figure we can see a graphical representtion of the training set. This data could serve for a task of regression.

Validation test

A **validation set** is a date set of examples used to tune the hyperparameters[1] of a classifier. It should follow the same probability distribution as the training set.

In order to avoid overfitting, when any classification parameter needs to be adjusted, it is necessary to have a validation set in addition to the training and test set. For example, if the most suitable classifier for the problem is sought, the training data set is used to train the different candidate classifiers, the validation data set is used to compare their performances and decide which one to take and, finally, the test set is used to obtain the performance characteristics such as accuracy, sensitivity, specificity, and so on. The validation set functions as a hybrid: it is training data used for testing, but neither as part of the low-level training nor as part of the final testing.

Test set

The **test data set** is a data set used to provide an unbiased evaluation of a final model fit on the training set. If the data in the test set has

[1] **Hyperparameter:** parameter whose value is used to control the learning proces. An example for artificial neural networks includes the number of hidden units in each layer.

never been used in training, the test set is also called **holdout set**.

A test set is a data set that is independent of the training set, but that follows the same probability distribution. If a model fit to the training set also fits the test set well, minimal overfitting has taken place. A better fitting of the training data set as opposed to the test set usually points to over-fitting.

A test set is therefore a set of examples used only to assess the performance of a fully specified classifier. To do this, the final model is used to predict classifications of examples in the test set. Those predictions are compared to the examples' true classifications to assess the model's accuracy.

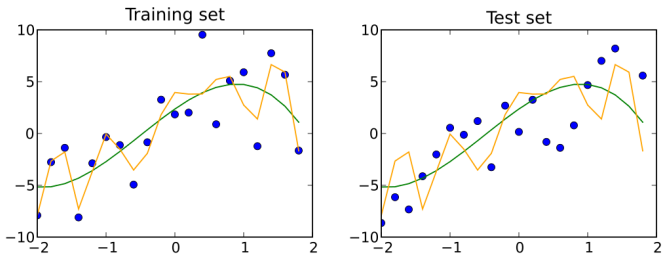


Figure 2.2

Figure 2.3: A training set and a test set from the same statistical population are shown as points. Two predictive models are fit to the training data. Both fitted models are plotted with both the training and test sets.

2.2.2 Data generating distribution

Our assumption is that learning problems are characterized by some unknown probablity distribution D over an input/output pairs $(x, y) \in X \times Y$.

The training and test data are generated by a probability distribution overdatasets called the **data-generating process**. We typically make a set of assumptions known collectively as the i.i.d. assumptions. These assumptions are that the examples in each dataset are independent from each other, and thatthe training set and test set are identically distributed, drawn from the same probability distribution as each other.

This assumption enables us to describe the data-generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every testexample. We call that shared underlying distribution the **data-generating distribution**, denoted p_data. This probabilistic framework and the i.i.d. assumptions enables us to mathematically study the relationship between training error and test error.

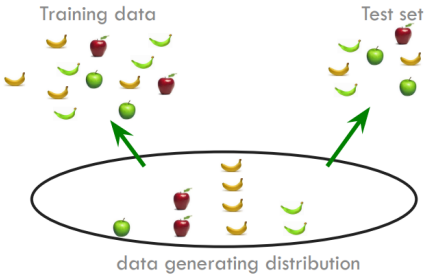
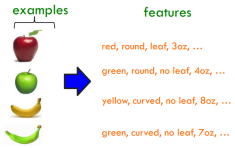


Figure 2.4

Figure 2.5: Example of a data generating distribution.

2.3 Feature

In machine learning and pattern recognition, a **feature** is an individual measurable property or characteristic of a phenomenon. Choosing informative, discriminating and independent features is a crucial element of effective algorithms in pattern recognition, classification and regression. Feature are usually numeric, but structural feature such as strings and graphs are used in syntactic pattern recognition. The concept of feature is related to that of explanatory variable used in statistical techniques such as linear regression.



One of the most important aspect of machine learning model is identifying the features which will help create a great model, the model that performs well on unseen data. The initial set of raw features can be redundant and too large to be managed. Therefore, a preliminary step in many applications of machine learning and pattern recognition consists of selecting a subset of features, or constructing a new and reduced set of features to facilitate learning, and to improve generalization and interpretability. Extracting or selecting features is a combination of art and science; developing systems to do so is known as feature engineering. It requires the experimentation of multiple possibilities and the combination of automated techniques with the intuition and knowledge of the domain expert. Automating this process is feature learning, where a machine not only uses features for learning, but learns the features itself.

A **feature vector** is an n-dimensional vector of numerical features that represent some object. Many algorithms in machine learning require a numerical representation of objects, since such representations facilitate processing and statistical analysis. Feature vectors are often combined with weights using a dot product in order to construct a linear predictor function that is used to determine a score for making a prediction.

The vector spaces associated with these vector is often called **feature**

space. In order to reduce the dimensionality of the feature space, a number of dimensionality reduction techniques can be employed.

Higher-level features can be obtained from already available features and added to the feature vector. This process is referred to as feature construction. Feature construction is the application of a set of constructive operators to a set of existing features resulting in construction of new features. Feature construction has long been considered a powerful tool for increasing both accuracy and understanding of structure, particularly in high-dimensional problems.

2.3.1 Characteristics of good features

A great feature must satisfy the following criteria, that are the characteristics of good features:

- Features must be found in most of the data samples: Great features represent unique characteritistics which can be applied across different types of data samples and are not limited to just one data sample. For example, can the "red" color of apple act as a feature? Not really. Because apple can be found in different colors. It might have happened that the sample of apples that was taken for evaluation contained apple of just "red" color. If not found, we may end up creating models having high bias.
- Features must be unique and may not be found prevalent with other (different) forms: Great features are the ones which is unique to apple and should not be applicable for other fruits. The toughness characteristic of apple such as "hard to teeth" may not be good feature. This is because a guava can also be explained using this feature.
- Features in reality: There can be features which can be accidental in nature and is not a feature at all when considering the population. For example, in a particular sample of data, a particular kind of feature can be found to be prevalent. However, when multiple data samples are taken, the feature goes missing.

2.4 Types of learning

Machine learning approeaches are traditionally divided into three broad categories, depending on the nature of the *signal* or *feedback* available to the learning system. Machine Learning has found its applications in almost every business sector. There are several algorithms used in machine learning that help you build complex models. Each of these algorithms in machine learning can be classified into a certain category.

There are primarily three types of machine learning:

- Supervised Learning;

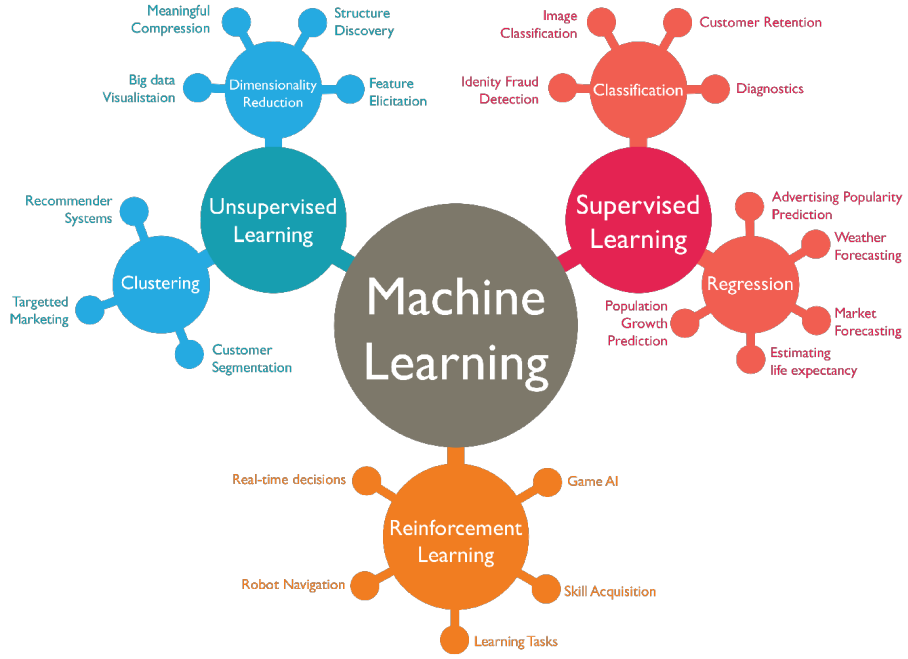
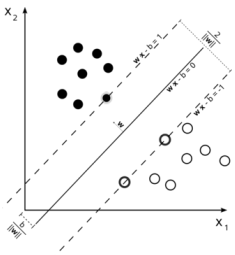


Figure 2.6: Types of learning map

- Unsupervisde Learning;
- Reinforcement Learning.

2.4.1 Supervised learning



Supervised learning algorithms build a mathematical model of a set of data that contains both the inputs and the desired outputs. The training data consists of a set of training examples. Each training example has one or more inputs and the desired output, also known as a supervisory signal.

Through iterative optimization of an objective function, supervised learning algorithms learn a function that can be used to predict the ouput associated with new inputs. An optimal function will allow the algorithm to correctly determine the output for inputs that were not a part of the training data. An algorithm that improves the accuracy of its outputs or predictions over time is said to have learned to perform that task.

Supervised learning uses a training set to teach models to yield the desired output. This training dataset includes inputs and correct outputs, which allow the model to learn over time. The algorithm measures its

accuracy through the loss function[2], adjusting until the error has been sufficiently minimized.

We can resume the supervised learning in the following definition: Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs. It infers a function from labeled training data consisting of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value.

Classification

Classification is the problem of identifying which of a set of categories an observation belongs to. An algorithm that implements classification, especially in a concrete implementation, is known as a *classifier*. The term also refers to the mathematical function, implemented by a classification algorithm, that maps input data to category.

Classification is a process of categorizing a given set of data into classes, It can be performed on both structured or unstructured data. The process starts with predicting the class of given data points. The classes are often referred to as target, label or categories.

The classification predictive modeling is the task of approximating the mapping function from input variables to discrete output variables. The main goal is to identify which class/category the new data will fall into.

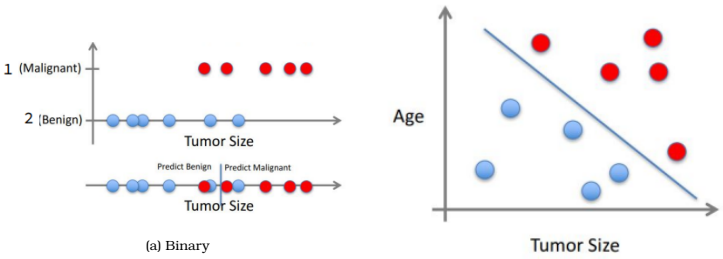
We can divide the classification problem into two subproblems, that are binary classification and multiclass classification:

- **Binary classification:** It is a type of classification with two outcomes, for eg – either true or false. Given a training set $T = \{(x_1, y_1), ..., (x_m, y_m)\}$. Learn a function f to predict y given x. y is categorical, $d = 1$.
$$f : \mathbb{R}^d \to \{1, 2, ..., k\}$$
- **Multiclass classification:** The classification with more than two classes, in multi-class classification each sample is assigned to one and only one label or target.. Given a training set $T = \{(x_1, y_1), ..., (x_m, y_m)\}$. Learn a function f to predict y given x. x is multidimensional (multiple features).
$$f : \mathbb{R}^d \to \{1, 2, ..., k\}$$

Regression

Regression analysis is a set of statistical processes for estimating the relationships between a dependent variable and one or more independent variables. The most common form of regression analysis is linear regression, in which one finds the line that most closely fits the data according to a specific mathematical criterion.

[2]We will talk more about loss functions in the next chapters.



(a) Binary

(b) Multiclass

Figure 2.7: Classification example

The following article provides an outline for Regression in Machine Learning. Regression means to predict the value using the input data. Regression models are used to predict a continuous value. It is mostly used to find the relationship between the variables and forecasting. Regression models differ based on the kind of relationship between dependent and independent variables.

Given a training set $T = \{(x_1, y_1), ..., (x_m, y_m)\}$. Learn a function f to predict y given x. y is real-valued $d = 1$.

$$f : \mathbb{R}^d \to \mathbb{R} \tag{2.1}$$

Linear regression is employed in varied ways in which a number of them are listed as:

- Sales prognostication
- Risk analysis
- Housing applications
- Finance applications

Ranking

Ranking is the data transformation in which numerical or ordinal values are replaced by their rank when the data are sorted. Training

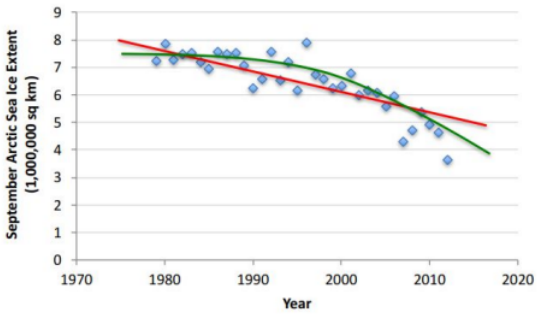


Figure 2.8: Regression

data consists of lists of items with some partial order specified between items in each list. This order is typically induced by giving a numerical or ordinal score or a binary judgment (e.g. "relevant" or "not relevant") for each item. The goal of constructing the ranking model is to rank new, unseen lists in a similar way to rankings in the training data.

For example, the ordinal data *hot*, *cold*, and *warm* would be replaced by 3, 1, 2.

Example 2.4.0 —

Training data consists of queries and documents matching them together with relevance degree of each match. It may be prepared manually by human assessors (or raters, as Google calls them), who check results for some queries and determine relevance of each result. It is not feasible to check the relevance of all documents, and so typically a technique called pooling is used — only the top few documents, retrieved by some existing ranking models are checked. This technique may introduce selection bias. Alternatively, training data may be derived automatically by analyzing clickthrough logs (i.e. search results which got clicks from users), query chains, or such search engines' features as Google's (since-replaced) SearchWiki. Clickthrough logs can be biased by the tendency of users to click on the top search results on the assumption that they are already well-ranked.

2.4.2 Unsupervised learning

Unsupervised learning algorithms take a set of data that contains only inputs, and find structure in the data, like grouping or clustering of data points. The algorithms, therefore, learn from test data that has not been labeled, classified or categorized.

Unsupervised learning uses machine learning algorithms to analyze and cluster unlabeled datasets. These algorithms discover hidden patterns or data groupings without the need for human intervention. Its ability to discover similarities and differences in information make it the ideal solution for exploratory data analysis, cross-selling strategies, customer segmentation, and image recognition.

Instead of responding to feedback, unsupervised learning algorithms identify commonalities in the data and react based on the presence or absence of such commonalities in each new piece of data.

Clustering

Cluster analysis is the assignment of a set of observations into subsets (called *clusters*) so that observations within the same cluster are similar

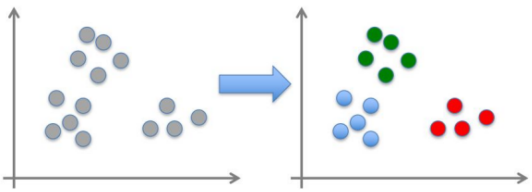


Figure 2.9: Clustering

according to one or more predesignated criteria, while observations drawn from different clusters are dissimilar.

Cluster analysis itself is not one specific algorithm, but the general task to be solved. It can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances between cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a multi-objective optimization problem. The appropriate clustering algorithm and parameter settings (including parameters such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

Formally, given $T = \{x_1, ..., x_m\}$ without labels, the output is the hidden structure behind the x's, that is the cluster.

Anomaly detection

Anomaly detection is the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data. In data analysis, anomaly detection (also referred to as outlier detection) is generally understood to be the identification of rare items, events or observations which deviate significantly from the majority of the data. Such examples may arouse suspicions of being generated by a different mechanism, or appear inconsistent with the data.

Typically the anomalous items will translate to some kind of problem such as bank fraud, a structural defect, medical problems or errors in a text. Anomalies are also referred to as outliers, novelties, noise, deviations and exceptions.

Anomaly detection is applicable in a very large number and variety of domains, and is an important subarea of unsupervised machine learning. As such it has applications in cyber-security intrusion detection, fraud

detection, fault detection, system health monitoring, event detection in sensor networks, detecting ecosystem disturbances, defect detection in images using machine vision, medical diagnosis and law enforcement.

Dimensionality reduction

Dimensionality reduction is a process of reducing the number of random variables under consideration by obtaining a set of principal variables. It is a process of reducing the dimension of the feature set, also called *number of features*. Most of the dimensionality reduction techniques can be considered as either feature elimination or extraction.

Dimensionality reduction, or dimension reduction, is the transformation of data from a high-dimensional space into a low-dimensional space so that the low-dimensional representation retains some meaningful properties of the original data, ideally close to its intrinsic dimension. Working in high-dimensional spaces can be undesirable for many reasons; raw data are often sparse as a consequence of the curse of dimensionality, and analyzing the data is usually computationally intractable (hard to control or deal with). Dimensionality reduction is common in fields that deal with large numbers of observations and/or large numbers of variables, such as signal processing, speech recognition, neuroinformatics, and bioinformatics.

Dimensionality reduction can be used for noise reduction, data visualization, cluster analysis, or as an intermediate step to facilitate other analyses.

2.4.3 Reinforcement learning

Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward.

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward. Although the designer sets the reward policy-that is, the rules of the game-he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint machine's creativity. In contrast to human beings, artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.

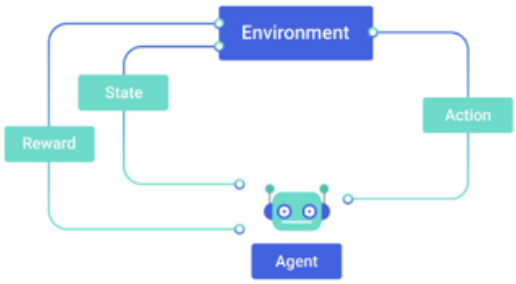


Figure 2.10: Reinforcement learning flow chart

Training the models that control autonomous cars is an excellent example of a potential application of reinforcement learning. In an ideal situation, the computer should get no instructions on driving the car. The programmer would avoid hard-wiring anything connected with the task and allow the machine to learn from its own errors. In a perfect situation, the only hard-wired element would be the reward function.

2.4.4 Other Learning Types

- semi-supervised learning, - active learning, - online vs offline learning, - generative vs discriminative, - parametric vs non-parametric.

Exercises

1. Describe the learning process and each of its step.
2. What is data? How can be split? How is it used in Machine Learning?
3. Provide the definitions of Information and Knowledge.
4. What is a feature? What is a feature vector? What do they represent?
5. Provide the definitions of the three types of learning.
6. Provide the definitions of classification, regression, and ranking.

   ! What is multiclass classification?
7. Provide the definitions of clustering, anomaly detection, and dimensionality reduction.

   ! What is dimensionality reduction? What is it useful for? What are its benefits?
8. What is Reinforcement Learning? Provide a basic flow chart.

   ! What is the difference between supervised learning and reinforcement learning?

3

Generalization error, Models, Hypothesis space

> *"Machine learning is the science of getting computers to act without being explicitly programmed."*
>
> – A. Samuel

Machine learning is a very general and useful framework, but won't always work. In order to better understand when it will and when it will not work, it is useful to formalize the learning problem more.

3.1 Task

Tasks represent the type of prediction being made to solve a problem on some data. We can identify a task with the set of functions that can potentially solve it. In general, it consists of functions assigning each input $x \in X$ an output $y \in Y$.

$$f : X \to Y \qquad F_{\text{task}} \subset Y^X$$

The nature of X, Y and F_{task} depends on the type of task.

Classification task The task for classification is to find a function $f \in Y^X$ assigning each input $x \in X$ a discrete label.

$$f(x) \in Y = \{c_1, ..., c_k\}$$

Common classification applicaton are face recognition, character recognition, spam detection, medical diagnosis (from symptoms to illnesses), and biometrics.

Regression task The task for regression is to find a function $f(x) \in Y$ assigning each input a *countinuous* label. Example of applications for regression task are in the field of economics/finance (predict the value of a stock), epidemiology, car/plane navigation (angle of the steering wheel, acceleration, ...), temporal trends (weather over time)..

Density estimation Density estimation is the construction of an estimate, based on observed data, of an unobservable underlying probability density function. The unobservable density function is thought of as the density according to which a large population is distributed.

The task for density estimation is to find a probability distribution $f \in \Delta(X)$[1] that fits the data $x \in X$.

Clustering task The task for clustering is to find a function $f \in \mathbb{N}^X$ that assigns each input $x \in X$ a cluster index $f(x) \in \mathbb{N}$. All points mapped to the same index form a cluster.

Some applications for clustering tasks are social network analysis and genomics (group individuals by genetic similarity).

Dimensionality reduction task The task for dimensionality reduction is to find a function $f \in Y^X$ mapping each (high dimensional) input $x \in X$ to a lower dimensional embedding $f(x) \in Y$, where $\dim(Y) \ll \dim(X)$, and $Y = \mathbb{R}^2$.

3.2 Model and hypotesis space

A **model** in machine learning is the output of a machine learning algorithm run on data. A model represents what was learned by a machine learning algorithm.

The model is the thing that is saved after running a machine learning algorithm on training data and represents the rules, numbers, and any other algorithm-specific data structures required to make predictions.

[1] $\Delta(X)$ is the set of all the probability distributions.

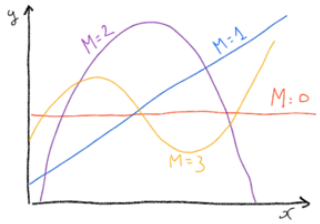
Example 3.2.0 —

- The linear regression algorithm results in a model comprised of a vector of coefficients with specific values.
- The decision tree algorithm results in a model comprised of a tree of if-then statements with specific values.
- The neural network/backpropagation/gradient descent algorithm together results in a model comprised of a graph structure with vectors or matrices of weigths with specific values.

A machine learning model is more challenging for a beginner because there is not a clear analogy with other algorithms in computer science. The best analogy is to think of the machine learning model as a program, comprised of both data and a procedure for using the data to make a prediction. It is the implementation of a function $f \in F_{\text{task}}$ that can be computed. The function f is an abstract concept, and the model is the implementation.

We assume that the set of all possible model is a subset $H \subset F_{\text{task}}$, called *hypotesis space*.

Example 3.2.0 — Polynomial curve fitting



Data are the dots in the graph, and we want to learn the function. One way is to find the set of functions f and parametrize them with paremeter w, multiplying with associated x to the power of j. This corresponds to a polynomial of different degrees.
Model:

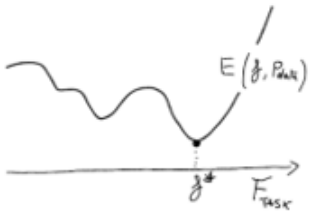
$$f_w(x) = \sum_{j=0}^{M} w_j x^j$$

Hypothesis space:

$$H_M = \{f_w : w \in \mathbb{R}^M\}$$

where H_M is the hypotesis space for fixed $w \in \mathbb{N}$.

3.2.1 The ideal target

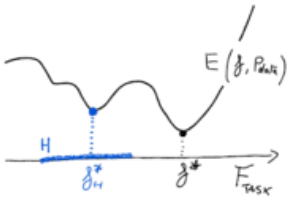


We want to minimize a generalization error function $E(f; p_{\text{data}})$. The error function determines how well a solution $f \in F_{\text{task}}$ fits some given data and guides the selection of the best solution in F_{task}:

$$f^* \in arg \min_{f \in F_{\text{task}}} E(f; p_{\text{data}})$$

An error function tells how well my model fits some given data. Allows us to find the best f^* in the space of F_{task}. Since this search space is too large and we do not have access to p_{data}, we cannot complete this search, so we need an implementation for the error function.

3.2.2 The feasible target

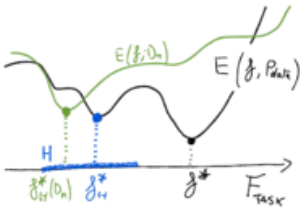


We need to restrict the focus on finding functions that can be implemented and evaluated in a tractable way. Thus we define a model and an hypotesis space $H \subset F_{\text{task}}$ and seek a solution within that space.

$$f_H^* \in arg \min_{f \in H} E(f; p_{\text{data}})$$

We have restricted our search space, however we still may have some trouble computing the search, because p_{data} is unknown.

3.2.3 The actual target



We need to revisit our minimization problem. Instead of using p_{data} we need to work on a data sample, i.e. a training set $D_n = \{z_1, ..., z_n\}$ where $z_i = (x_i, y_i) \in X \times Y$, and $z_i \sim p_{\text{data}}$.

$$f_H^*(D_n) \in arg \min_{f \in H} E(f; D_n)$$

This function is called training error and it does not match with the generalization error.

3.3 Error function

Machine learning can be thought of as an optimization problem, where there is an objective function that needs to be either maximized or

minimized, and the best solution is the model that achieves either the highest or lowest score respectively.

Typically in machine learning problems, we seek to minimize the error between the predicted value vs the actual value. The word *error* represents the penalty of failing to achieve the expected output. If the loss is calculated for a single training example, it is called or **error function**. If the same loss is averaged across the entire training sample, the loss is called cost function.

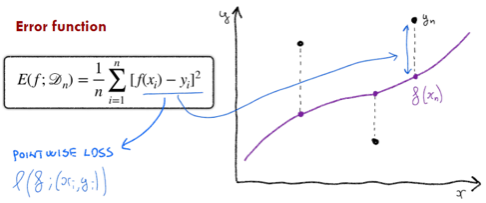
Error function vary with the type of problem we are trying to solve. Regression problems that attempt to predict a continuous value have one set of loss functions, while the classification problems where the algorithm attempts to classify the training example into one of the target classes have another set of error/cost function.

Typically, the generalization and training error function can be written in terms of a pointwise loss $l(f; z)$ measuring the error incurred by f on the training example z.

$$E(f; p_{\text{data}}) = E_{z \sim p_{\text{data}}}[l(f; z)] \tag{3.1}$$

$$E(f; D_n) = \frac{1}{n} \sum_{i=1}^{n} l(f; z_i) \tag{3.2}$$

Example 3.3.0 — Polynomial curve fitting



The pointwise loss is computed on each sample x_i separately. In this case it makes sense to have the square term, in this way we have the so called *square loss*. Given our training data, we compute y_i and our function f. Our objective is:

$$f^*_{H_M}(D_n) \in arg \min_{f \in H_M} E(f; D_n)$$

The objective is equivalent to f_{w^*}, where w^* is defined as follows:

$$w^* \in arg \min_{w \in \mathbb{R}^M} \frac{1}{n}\sum_{i=1}^{n}[f_w(x_i) - y_i]^2$$

This requires to solve a linear system of equations.

Our learning algorithm solves the optimization problem targeting $f^*_H(D_n)$, but might end up in a different result.

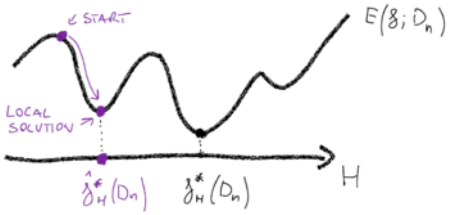


Figure 3.1

Let us put all the charts together, and build a recap chart for the $f^*(D_n)$ function, showing the training error and the generalization error.

After performing our search, we obtain our learning output, this could introduce two potential problems, overfitting and underfitting.

3.3.1 Overfitting

Overfitting is the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional

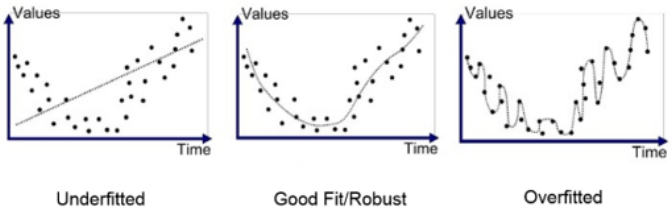


Figure 3.2: Graphical representation of curve fitting

data or predict future observations reliably.

An overfitted model is a statistical model that contains more parameters than can be justified by the data. The essence of overfitting is to have unknowningly extracted some of the residual variatione (i.e., the noise) as if the variation represented underlying model structure.

Overfitting occurs when the learned function $\hat{f}_H^*$, becomes sensitive to the noise in the sample. As the result, the function will perform well on the training set but not perform well on other data from the joint probability distribution of x and y. Thus, the more overfitting occurs, the larger the generalization error.

3.3.2 Underfitting

Underfitting occurs when a machine learning algorithm cannot adequately capture the underlying structure of the data. It occurs when the model or algorithm shows low variance but high bias. It is often a result of an excessively simple model which is not able to process the complexity of the problem. This results in a model which is not suitable to handle all the signal and is therefore orced to take some signal as noise.

If instead a model is capable to handle the signal but anyways takes a part of it as noise as well, it is also considered to be underfitted. The latter case can happen if the loss function of a model includes a penalty which is too high in that specific case.

An underfitted model would ignore some important replicable structure in the data and thus fail to identify effects that were actually supported by the data. In this case, bias in the parameter estimators is often substantial, and the sampling variance is underestimated, both factors resulting in poor confidence interval coverage. Underfitted models tend to miss important treatment effects in experimental settings.

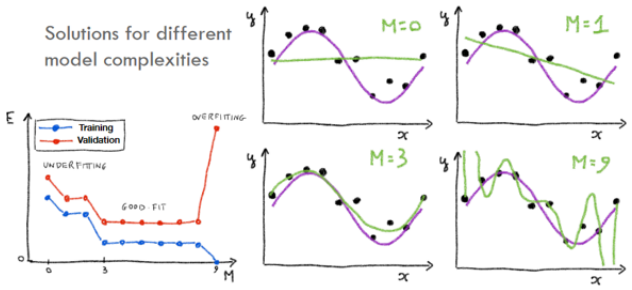
3.3.3 Estimate the generalization error

The generalization error cannot be computed for p_{data}, because it's unknown. The only possible way to access generalization error is by

sampling. We use a data sample, and we assume that the probability distribution is the same for the training set, for the validation set, and for the test set.

We use the training set to find the best model, then the validation set to find the best hyperparameters. Once we have done this operation, we may think of having a way to estimate the generalization error in order to understand if the model will work well on the test data.

Example 3.3.0 — Polynomial curve fitting



With the first model in Figure 3.3.3, there is an underfitting, high error, low performance in training and validation set. On the other side, with the last model there is overfitting, we approximate with almost 0 errors on training data, while performing poorly on unknown test data represented by smooth curve.

3.3.4 Improve the generalization

In order to improve generalization well, one needs to avoid both underfitting of the training data and overfitting of the training data.

There are a number of approeaches for improving generalization, one can:

- Avoid to obtain the minimum on training error;
- Reduce the model capacity;
- Change the objective with a regularization term;
- Inject noise in the learning algorithm to smooth out the data points;
- Stop the learning algorithm before the convergence to avoid the model to learn too well on training data.

Regularization

The regularization is the modification of the training error function with a term $\Omega(f)$ that typically penalizes complex solutions.

$$E_{\text{reg}}(f; D_n) = E(f; D_n) + \lambda_n \Omega(f)$$

We are adding another function Ω to force the algorithm to learn a model with low complexity. It penalizes complex solution to find a simpler one, which can find a model that generalizes better (so to find a better f^*). The λ_n is a trade-off parameter.

Example 3.3.0 — Polynomial curve fitting

We regularize by penalizing polynomials with large coefficients:

$$E_{\text{reg}}(f; D_n) = \frac{1}{n}\sum_{i=1}^{n}[f_w(x_i) - y_i]^2 + \frac{\lambda}{n}||w||^2$$

$$||w||^2 = \sum_i = w_i^2$$

We invoke norm regularization and set lambda hyperparameters by considering performance on validation set. The lambda parameter regulates the trade-off between underfitting and overfitting

There are other approaches one can attempt, for example:

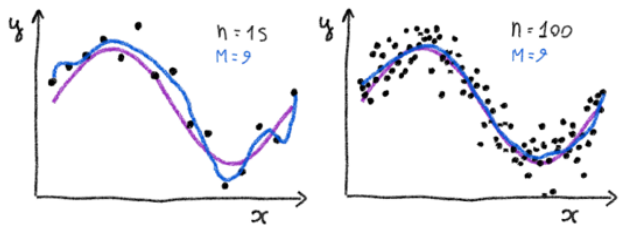
- Increase the amount of data;
- Add more training samples;
- Augment the training set with transformations, when the provious is not possible;
- Combine prediction from multiple, decorrelated models to have preditions that are independent, and merging them we will probably get better labels. This technique is called *ensembling*.

Example 3.3.0 — Polynomial curve fitting

Let's see the modification of the generalization in respect to the data size.

$$E(f; D_n) \to E(f; p_{\text{data}}) \qquad \text{as } n \to \infty$$

A graphical representation is provided in Figure **??**.



Part II

Supervised Learning

4

k-Nearest Neighbor

After having introduced all the basics of machine learning, we start talking about the specific algorithm in order to address at the beginning of supervised classification problem. The first algorithm we present is one of the simplest one, and it is the **K-Nearest Neighbor**.

4.1 Introduction

K-Nearest Neighbor classifier is one of the introductory supervised classifiers, which every data science learner should be aware of. This algorithm was first used for a pattern classification task which was first used by Fix & Hodges in 1951. To be similar the name was given as KNN classifier. KNN aims for pattern recognition tasks.

K-Nearest Neighbor also known as KNN is a supervised learning algorithm that can be used for regression as well as classification problems. Generally, it is used for classification problems in machine learning.

In **KNN classification**, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.

In **KNN regression** the output is the property value for the object. This value is the average of the values of k nearest neighbors.

Both for classification and regression, a useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor

a weight of $\frac{1}{d}$, where d is the distance to the neighbor.

4.2 Algorithm

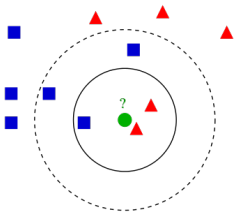


Figure 4.1

The training examples are vectors in a multidimensional feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples.

In the classification phase, k is a user-defined constant, and an unlabeled vector is classified by assigning the label which is most frequent amongh the k training samples nearest to that query point.

In Figure 4.1 we have an example of KNN classification. The test sample (the green dot) should be classified either to blue squares or to red triangles. if $k = 3$ (solid line circle) it is assigned to the red triangles because there are two triangles and only one square. If $k = 5$ (dashed line circle) it is assigned to the blue squares.

To classify an example d you have to find the k nearest neighbors of d, and then choose as the label the majority label within the k nearest neighbors. But how do we measure the *nearest*?

4.2.1 Euclidean distance

The most common choice, is to compute the **euclidean distance** between the test sample and his neighbors.

In two dimensions, given two points $a(a_1, a_2)$ and $b(b_1, b_2)$, the euclidean distance is computed as follows:

$$D(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$$

In n-dimensions, we compute the distance in an analogous manner. Given two points $a(a_1, a_2, ..., a_n)$ and $b(b_1, b_2, ..., b_n)$, the euclidean distance is computed as follows:

$$D(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2} \tag{4.1}$$

$$= \sqrt{\sum_{d=1}^{D} (a_d - b_d)^2} \tag{4.2}$$

Measuring distance (or similarity) is a domain-specific problem and there are many, many differen variations.

4.2.2 Algorithm

Algorithm 1: kNN-classification($data$, $query$, k, $distance_{fn}$)

$neighborDistance \leftarrow [\,]$;
foreach $(index, example) \in data$ **do**
  $distance \leftarrow distance_{fn}(example.pop(), query)$;
  $neigborDistance.append((distance, index))$;
end
$sortedND \leftarrow sort(neighborDistance)$;
$kNNDistances \leftarrow sortedND[k]$; `//Get first k neighbors`
$kNNLabels \leftarrow [kNNDistances.index]$; `//Get labels of first k neighbors`
return *kNNDistances, Counter(labels).mostCommon()*

4.3 Decision boundaries

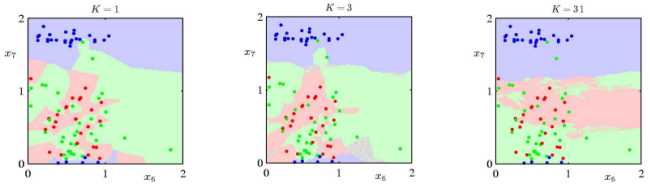
The decision boundaries are areas in the features space where the classification of a point (example) changes.

Nearest neighbor rules implicitly compute the decision boundary. It is also possible to compute the decision boundary explicitly, and to do so efficiently, so that the computational complexity is a function of the boundary complexity.

Decision boundaries are useful ways to visualize the complexity of a learned model. Intuitively, a learned model with a decision boundary that is really jagged (like the coastline of Norway) is really complex and prone to overfitting. A learned model with a decision boundary that is really simple (like the bounary between Arizona and Utah) is potentially underfit.

In kNN, once you have chosen k, kNN gives locally defined decision boundaries between classes, that depend on the choice of k.

4.4 Choosing k



The best choice of k depends upon the data; generally, larger values of k reduces effect of the noise on the classification, but make boundaries between classes less distinct. A good k can be selected by various heuristic techniques. The special case where the class is predicted to be the class of the closest training sample (i.e. when $k = 1$ is called the nearest neighbor algorithm.

The accuracy of the kNN algorithm can be severely degraded by the presence of noisy or irrelevant features, or if the feature scales are not consistent with their importance. A particularly popular approach is the use of evolutionary algorithms to optimize feature scaling. Another popular approach is to scale fetures by the mutual information of the training data with the training classes.

In binary classification problems, it is helpful to choose k to be an odd number as this avoids tied votes. One popular way of choosing the empirically optimal k in this setting is via bootstrap method.

4.5 KNN in practice

KNN's main disadvantage of becoming significantly slower as the volume of data increases makes it an impractical choice in environments where predictions need to be made rapidly. Moreover, there are faster algorithms that can produce more accurate classification and regression results.

However, provided you have sufficient computing resources to speedily handle the data you are using to make predictions, KNN can still be useful in solving problems that have solutions that depend on identifying similar objects.

4.5.1 Recommender Systems

At scale, this would look like recommending products on Amazon, article on Medium, movies on Netflix, or videos on YouTube. Although, we can be certain they all use more efficient means of making recommendations due to the enormous volume of data they process.

4.5.2 Weighted k-Nearest Neighbor

The intuition behind weighted kNN, is to give more weight to the points which are nearby and less weight to the points which are farther away. Any function can be used as a kernel funcion for the weighted kNN classifier whose value decreases as the distance increases.

The k-nearest neighbor classifier can be viewed as assigning the k nearest neighbors a weight $\frac{1}{k}$ and all others 0 weights. This can be generalised to weighted nearest neighbour classifiers. That is, where the

ith nearest neighbour is assigned a weight w_{ni}m with $\sum_{i=1}^{n} w_{ni} = 1$.

Algorithm

- Let $L = \{(x_i, y_i), i = 1, 2, ..., n\}$ be a training set of observations x_i with iven class y_i, and let x be a new observation (query point) whose class label y has to be predicted.
- Compute $d(xi, x)$ for $i = 1, 2, ..., n$, the distance between the query point and every other point in the training set.
- Select $D' \subset D$, the set of k nearest training data points to the query points.
- Predict the class of the query point, using distance-weighted voting. The v represents the class labels. Use the following formula:

$$y' = argmax_v \sum_{(x_i, y_i) \in D_z} w_i \times I(v = y_i) \tag{4.3}$$

5

Linear Models

We now move over to discuss another simple model in Machine Learning, the class of **Linear Models**. Linear Models leads us to the introduction of **perceptrons**.

Some machine learning approaches make strong assumptions about the data: if the assumptions are true it can often lead to better performance, but if the assumptions are false the approach can fail miserably. Other approaches don't make many assumptions about the data, this can allow us to learn from more varied data, but they are more prone to overfitting and generaly require more training data.

Linear models generate a formula to create a best-fit line to predict unknown values. Linear Models are considered not as predictive as newer algorithm classes, but they can be trained relatively quicky and are generally more straightforward to interpret, which can be a big plus.

5.1 Bias

The **bias** of a model is how strong the model assumptions are. Low-bias classifiers make minimal assumptions about the data (kNN and decision trees are generally considered low bias), high-bias classifiers make strong assumptions about the data.

A strong high-bias assumption is **linear separability**: in two dimenstions, can separate classes by a line, in higher dimensions, need hyperplanes.

A linear model is a model that assumes the data is linearly separable.

5.1.1 Define a line

Any pair of values (w_1, w_2) defines a line through the origin: $w_1 f_1 + w_2 f_2 = 0$.

Example 5.1.0 —

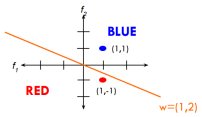


Figure 5.1

For example, in Figure **??** we have the line $1f_1 + 2_2 = 0$. This equation means that we have our training data in a two-dimensional space, we have only two features (f_1 and f_2), and the line that separate our training points is the orange line.
We can see the parameters $(w_1 = 1, w_2 = 2)$ as the vector perpendicular to the orange line. In fact, the orange line is perpendicular to the vector that starts from the origin $O(0, 0)$, and goes to the point $w(1, 2)$.
How can we classify mathematically the points based on a line? If we got two example data $B(1, 1)$ and $R(1, -1)$, we have the following results:

$$(1, 1) : 1 * 1 + 2 * 1 = 3$$
$$(1, -1) : 1 * 1 + 2 * (-1) = -1$$

The sign of the result indicates which side of the line is occupied by the example in the space.

We can move our line from the origin, if we use the following equation, that uses a parameter a in order to represent any line in a two dimensional space:

$$w_1 f_1 + w_2 f_2 = a$$

In this case we have a line intersects at a in the horizontal axis.

A linear model in n-dimensional space (i.e. n features) is defined by $n + 1$ weights. In two dimesions, we have a line

$$w_1 f_1 + w_2 f_2 + b = 0 \qquad \text{where } b = -a \tag{5.1}$$

In three dimensions, we have a plane

$$w_1 f_1 + w_2 f_2 + w_3 f_3 + b = 0$$

In n dimensions, we have a **hyperplane**

$$b + \sum_{i=1}^{n} w_i f_i = 0 \tag{5.2}$$

5.1.2 Classifying with linear models

Havin define the notion of line and the notion of hyperplane, we can show that a linear model, and in particular an hyperplane, can be used for classification by simply looking at the sign of the following equation:

$$b + \sum_{i=1}^{n} w_i f_i \tag{5.3}$$

if the result of equation 5.3 is greater than 0, we have a positive example that belongs to the positive class; otherwise, if the result is lower than 0, we have a negative example that belongs to the negative class.

5.2 Online learning

Online learning is a method of machine learning in which data becomes available in a sequential order ad is used to update the best predictor for future data at each step, as opposed to batch learning techniques which generate the best predictor by learning on the etire training data set at once.

Online learning is able to overcome the drawbacks of batch learning in that the predictive model can be updated intantly for any new data instances. Thus, online learning algorithms are far more efficient and scalable for large-scale machine learning tasks in real-world data analytics applications where data are not only large in size, but also arriving at a high velocity.

5.2.1 Tasks and applications

Online learning algorithms can be derived for supervised learning tasks. One of the most common tasks is classification, aiming to predict the categories for a new data instance belongs to, on the basis of observing past training data instances whose category labels are given. For example, a commonly studied task in online learning is **online binary classification** (e.g., spam email filtering) which only involves two categories ("spam" vs "benign" emails); other types of supervised classification tasks include multi-class classification, multi-label classification, and multiple-instance classification, etc.

In addition to classification tasks, another common supervised learning task is **regression analysis**, which refers to the learning process for estimating the relationships among variables (typically between a dependent variable and one or more independent variables). Online learning techniques are naturally applied for regression analysis tasks, e.g., time series analysis in financial markets where data instances naturally arrive in a sequential way. Besides, another application for online learning with financial time series data is **online portfolio section** where an

online learner aims to find a good (e.g., profitable and low-risk) strategy for making a sequence of decisions for portfolio selection.

5.3 Perceptron

The **perceptron** is an algorithm for supervised learning of binary classifiers. A binary classifier is a function which can decide wheter or not an input, represented by a vectior of numbers, belongs to some specific class.

In the modern sense, the perceptron is an algorithm for learning a binary classifier called a threshold function: a function that maps its input x to an output value $f(x)$:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0, \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where w is a vector of real-valued weights, $w \cdot x$ is the dot product $\sum_{i=1}^{m} w_i x_i$, where m is the number of inputs to the perceptron, and b is the bias.

The value of $f(x)$ is used to classify x as either a positive or a negative instance, in the case of a binary classification problem. If b is negative, then the weighted combination of inputs must produce a positive value greater thant $|b|$ in order to push the classifier neuron over the 0 threshold.

Spatially, the bias alters the position of the decision boundary. The perceptron learning algorithm does not terminate if the learning set is not linearly separable. If the vectors are not linearly separable learning will never reach a point where all vectors are classified properly.

5.3.1 Convergence and number of iterations

We will have convergence when the data are linearly separable, but when data is not linearly separable, we have to set up a number of iterations, otherwise the algorithm will fall in an infinite loop.

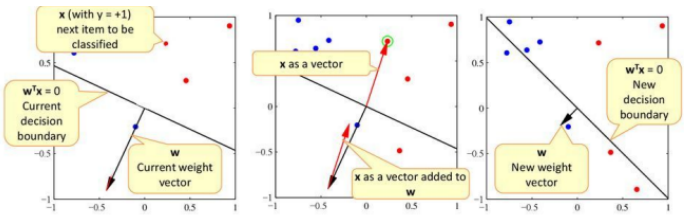
The training instances are linearly separable if, given two classes of examples, there exists a hyperplane that will separate the two classes.

The number of iterations can be used to prevent the overfitting of the perceptron algorithm.

Algorithm 2: Perceptron learning

```
while convergence not reached do
    foreach training example f_i, label do
        prediction ← b + Σ_{i=1}^{n} w_i f_i ;        //check if it is correct
        if not correct then
            foreach w_i do
                w_i ← w_i + f_i · label ;             //update all the weights
            end
            b ← b + label;
        end
    end
end
```

5.3.2 Algorithm



The perceptron algorithm guarantees that finds **some** line that separates the data, but we have no guarantees that the perceptron have found a special line. This will be especially important when we will discuss the difference between the perceptron and the Support Vector Machine.

6

Decision trees

Decision tree learning is one of the predictive modelling approaches used in machine learning. It uses a decisione tree as a predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). Tree models where the target variable can take a discrete set of values are called **classification trees**; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values are called **regression trees**. Decision trees are among the most popular machine learning algorithms given their intelligibility and simplicity.

The concept of a decision tree is to select splits that decrease the impurity of class distribution in the resulting subsets of instances. At the same time, the domination of one or more classes over the other classes is increased. This way, you can find a subset that contains only instances of one class after a few splits.

For decision trees, binary trees are referred because they can be generalized better than trees wmith high-cardinality attributes.

6.1 How decision trees work

Decision tree growing is done by creating a decision tree from a data set. Splits are selected, and class labels are assigned to leaves when no further splits are required or possible.

A decision tree takes an input $x \in X$, the growing starts from a signle root node, and routes it through its nodes

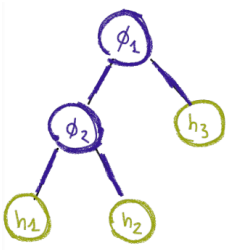


Figure 6.1

until it reaches a leaf node. Each non-terminal node $Node(\phi, t_L, t_r)$ holds a routing function $\phi \in \{L, R\}^X$, a left child t_L and a right child t_R. When x reaches the node it will go to the left child t_L or the right child t_R depending on the value of $\phi(x) \in \{L, R\}$. When you do a split, each of the created descendant nodes corresponds to the applicable subset of the training data set. Further splits of these nodes result in new nodes that correspond to smaller subsets of data sets, and so on. Nodes that are not split further become leaves, each leaf node $Leaf(h)$ holds a prediction function $h \in F_{\text{task}}$ (typically a constant), depending on the task we want to solve it can be $h \in Y^X$ (classification or regression) or $h \in \Delta(X)$ (density estimation) or other. Once x reaches the leaf, the final prediction is given by $h(x)$.

6.1.1 Inference

f_t is the function returning the prediction for input $x \in X$ according to the decision tree t. It is recursively defined as

$$f_t(x) = \begin{cases} h(x) & \text{if } t = Leaf(h) \\ f_{t_{\phi(x)}}(x) & \text{if } t = Node(\phi, t_L, t_R) \end{cases} \tag{6.1}$$

Namely, given an input sample x, if the decision tree t is a leaf, then do the prediction $h(x)$; otherwise, if the decision tree t is a node, then call recursively the prediction function f on the left decision tree node t_L or right decision tree node t_R, depending on the value of the function $\phi(x)$.

6.2 Decision tree learning algorithm

Given a training set $D_n = \{z_1, ..., z_n\}$, the set T of all the possible decision trees, find f_{t^*}, that corresponds to the optimal tree, where

$$t^* \in arg \min_{t \in T} E(f_t; D_n) \tag{6.2}$$

The optimization problem is easy if we don't impose constraints (e.g. most compact tree), a solution can be found using a simple, greedy strategy. Otherwise, the complexity class could be NP-hard. Hereafter, we will assume:

$$E(f_t; D) = \frac{1}{|D|} \sum_{z \in D} l(f; z) \tag{6.3}$$

That means that the empirical error of our training set can be computed by the normalized sum of the pair-wise losses.

Fix a set of leaf predictions $H_{\text{leaf}} \subset F_{\text{task}}$ (e.g., constant functions), that correspond to the number of classes. Fix a set of possible split functions $\Phi \subset \{L, R\}^X$. Define the tree-growing strategy that recursively partitions

the training set and decides whether to grow leaves or non-terminal nodes. Finally, we stop when the final criteria is reached, and all the examples of the current training set are associated to the same class.

6.2.1 Growing a leaf

Growing a leaf means to find a way to know if the current training set are associated to the same class. Formally, given $D = \{z_1, ..., z_n\}$ the training set reaching the node. The optimal leaf predictor, that minimizes the local error, can be computed as:

$$h_D^* \in arg \min_{h \in H_{\text{leaf}}} E(h; D) \tag{6.4}$$

The optimal error value is also called **impurity measure**, and tells us what is the probability of misclassifying an observation. Note that the lower the impurity, the better the split.

$$I(D) = E(h_D^*; D) \tag{6.5}$$

If some criterion is met we grow a leaf $Leaf(h_D^*)$, for example:

- Purity $I(D) < E$;
- Minimum cardinality $|D| < k$;
- Others...

6.2.2 Growing a node

Given the set of all the possible routing functions Φ, and the set of training samples D; if no stopping criterion is met, find the optimal split function, by minimizing the impurity function of our training set.

$$\phi_D^* \in arg \min_{\phi \in \Phi} I_\phi(D) \tag{6.6}$$

The impurity $I_\phi(D)$ of a split function ϕ given a training set D is computed in terms of the impurity of the split data:

$$I_\phi(D) = \sum_{d \in \{L,R\}} \frac{|D_d^\phi|}{|D|} I(D_d^\phi) \tag{6.7}$$

$$D_d^\phi = \{(x, y) \in D; \phi(x) = d\} \tag{6.8}$$

Note that the impurity of a split function is the lowest training error that can be attained by a tree consisting of a root and two leaves.

Now, we finally grow a node $Node(\phi^*, t_L, t_R$, where ϕ^* is the optimal split. t_L and t_R are obtained by recursively applying the learning algorithm to the associated training set splits.

In a nutshell:

$$Grow(D) = \begin{cases} Leaf(h_D^*) & \text{if stopping criterion met} \\ Node(\phi_D^*, Grow(D_L^*), Grow(D_R^*)) & \text{otherwise} \end{cases} \tag{6.9}$$

6.3 About split selection

Sometimes the selection of the best split function is not given in terms of minimizing the impurity of the split, but in the equivalent maximization of information gain, which is given by:

$$\Delta_\phi(D) = I(D) - I_\phi(D) \tag{6.10}$$

Information gain is non-negative, i.e. $\Delta_\phi(D) \geq 0$ for any $\phi \in \{L, R\}^X$ and any training set $D \subset X \times Y$. So, the impurity will never increase for any randomly chosen split.

6.4 Leaf predictions

The leaf rediction provides a solution to a simplified problem involving only data reaching the leaf. This solution can be an arbitrary function $h \in F_{\text{task}}$ but in practice we restrict it to a subset H_{leaf} of simple ones.

The simplest predictor one can construct is a function returning a constant (e.g. a prefixed class label). The set of all possible constant functions can be written as:

$$H_{\text{leaf}} = \bigcup_{y \in Y} \{y\}^X \tag{6.11}$$

6.5 Impurity measures for classification

In the case of classification, we have an output space $Y = \{c_1, ..., c_k\}$ and a training set $D \subset X \times Y$, that is the cartesian product of the input space and output space. Let $D^y = \{(x, y') \in D : y = y'\}$ denote the subset of training samples in D with class label y.

Consider the following error function:

$$E(f; D) = \frac{1}{|D|} \sum_{z \in D} l(f; z)$$

If $l(f; (x, y)) = 1_{f(x) \neq y}$, meaning that it is 1 if the condition $f(x) \neq y$ is verified, and the set of all the leaves $H_{\text{leaf}} = \cup_{y \in Y} \{y\}^X$, then the impurity measure is the **classification error**:

$$I(D) = 1 - max_{y \in Y} \frac{|D^Y|}{|D|} \tag{6.12}$$

If $H_{\text{leaf}} = \cup_{\pi \in \Delta(Y)} \{\pi\}^X$ and $l(f; (x, y)) = \sum_{c \in Y} [f_c(x) - 1_{c=y}]^2$, then the

impurity measure is the **Gini impurity**:

$$I(D) = 1 - \sum_{y \in Y} \left(\frac{|D^y|}{|D|} \right)^2 \tag{6.13}$$

If $l(f;(x,y)) = -\log f_y(x)$ and $H_{\text{leaf}} = \cup_{\pi \in \Delta(Y)} \{\pi\}^X$ (constant label distribution as leaf prediction) then the impurity measure is the **entropy**:

$$I(D) = -\sum_{y \in Y} \frac{|D^y|}{|D|} \log \frac{|D^y|}{|D|} \tag{6.14}$$

6.6 Impurity measures for regression

In the case of regression, we have the output space $Y \subset (R)^d$, and $D \subset X \times Y$. If $l(f;(x,y)) = ||f(x) - y||_2$, and $H_{\text{leaf}} = \cup_{y \in Y} \{y\}^X$, then the impurity measure is the **variance** associated to all the points in the training set:

$$I(D) = \frac{1}{|D|} \sum_{(x,y) \in D} ||x - \mu_D||^2 \tag{6.15}$$

where $\mu_D = \frac{1}{|D|} \sum_{(x,y) \in D} x$ is the *mean* of all the points in the training set.

6.7 Split functions

A data point $x \in X$ could be d-dimensional with each dimension taking heterogeneous types of values (discrete, continuous) and have a total ordering or not (ordinal or nominal).

The split function $\phi \in \{L, R\}^X$ determines wheter a data point $x \in X$ should move left ($\phi(x) = L$) or right ($\phi(x) = R$). The possible split functions are restricted to some predefined set $\Phi \subset \{L, R\}^X$ depending on the nature of the feature space. The prototypical split function for a d-dimensional input first selects one dimension and then applies a 1-dimensional splitting criterion.

6.7.1 Discrete nominal features

Assume discrete nominal features taking values in K (e.g. colors, marital status, etc.). The split function can be implemented given a partition of K into K_R and K_L, associated to different partitions of the tree.

$$\phi(x) = \begin{cases} L & \text{if } x \in K_L \\ R & \text{if } x \in K_R \end{cases} \tag{6.16}$$

Finding the optimal split requires testing $2^{|K|-1} - 1$ bi-partitions.

6.7.2 Ordinal features

Assume ordinal features, i.e. can be sorted (numbers, t-shirt sizes, etc.), taking values in K. The split function can be implemented given a threshold $r \in K$:

$$\phi(x) = \begin{cases} L & \text{if } x \leq r \\ R & \text{if } x > r \end{cases} \tag{6.17}$$

If $|K| \leq |D|$, finding the optimal split requires testing $|K| - 1$ thresholds. Otherwise, it requires sorting the input values in D, where D is the training set reaching the node, and testing $|D| - 1$ thresholds.

6.7.3 Oblique

Sometimes it is convenient to split using multiple features at once. Such split functions work with continuous features and are called **oblique**, because can generate oblique decision boundaries. If $x \in \mathbb{R}^d$ then the split function can be implemented given $w \in \mathbb{R}^d$ and $r \in \mathbb{R}$:

$$\phi(x) = \begin{cases} L & \text{if } w^T x \leq r \\ R & \text{otherwise} \end{cases} \tag{6.18}$$

6.8 Overfitting

Decision trees are **non-parametric models** with a structure that is determine by the data. As a result, they are flexible and can wasily fit the training set, but with high risk of **overfitting**.

Standard techniques to improve generalization apply also to decision trees:

- early stopping;
- regularization;
- data augmentations;
- complexity regularization;
- ensembling.

6.8.1 Pruning

Pruning is a data compression technique that reduces the size of decision trees by remoing sections in the tree that are non-critical and redundant to classify instances. Pruning reduces the complexity of the final classifier, and hence improves predictive accuracy by the reduction of overfitting.

A common strategy is to grow the tree until each node contains a small number of instances then use pruning to remove nodes that do not provide additional information.

Pruning should reduce the size of a learning tree without reducing predictive accuracy as measured by a cross-validation set. There are many techniques for tree pruning that differ in the measurement that is used to optimize performance.

6.9 Random forests

Random forests are an ensemble learning method for classification, regression, and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned. Random forests correct for decision trees' habit of overfitting to their training set. They generally outperform decision trees, but their accuracy is lower than gradient boosted trees. However, data characteristics can affect their performance.

If we bild many trees, all of which work weel and overfit in different ways, we can reduce the amount of overfitting by averaging their results. This reduction in overfitting, while retaining the predictive power of the trees, can be shown using rigorous mathematics.

To implement this strategy, we need to build many decision trees. Each tree should do an acceptable job of predicting the target, and should also be different from the other trees. Ranomd forests get their name from injecting randomness into the tree building to ensure each tree is different.

Split functions are optimized on randomly sampled features, or are samped completely at random. This helps obtaining decorrelated decision trees. The final prediction of the forest is obtained by averaging the prediction of each tree in the ensemble $Q = \{t_1, ..., t_T\}$.

$$f_Q(x) = \frac{1}{T} \sum_{j=1}^{T} f_t(x) \tag{6.19}$$

While random forests often achieve higher accuracy than a single decision tree, they sacrifice the intrinsic interpretability present in decision trees. They are very powerful, often work well without heavy tuning of the parameters, and don't require scaling of the data. Essentially, they share all of the benefits of decision trees, while making up for some of their deficiencies.

One reason to still use decision trees is if you need a compact representation of the decision making process. It is basically impossible to interpret tens or hundreds of trees in detail, and trrees in random forests tend to be deeper than decision trees.

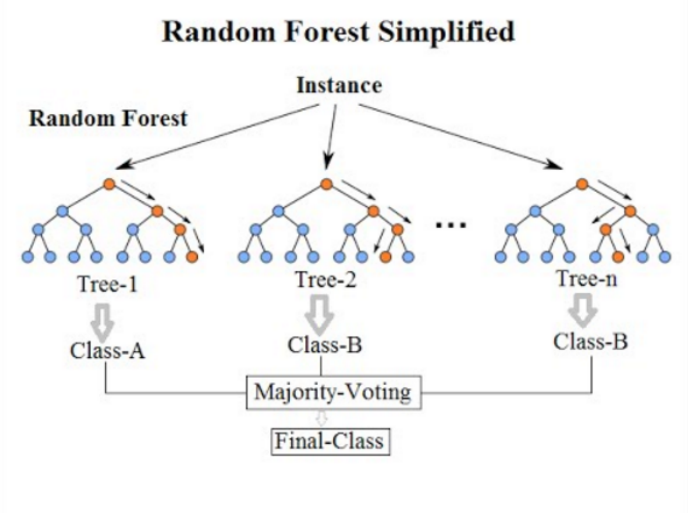


Figure 6.2: Random forest classification example

One application example of the random forest is a notable application in computer vision: the **kinect**.

7

Multi-class classification

Multiclass classification is the problem of classifying instances into one of three or more classes. While many classification algorithms naturally permit the use of more than two classes, some are by nature binary algorithms; these can, however, be turned into multinomial classifiers by a variety of strategies.

Most of the real-world applications involve multiclass classification, for example:

- document classification,
- handwriting recognition,
- face recognition,
- sentiment analysis,
- autonomous vehicles,
- emotion recognition.

We will see that it's relatively easy to think of a binary classifier as a *black box*, which can be reused for solving these more complex problems. This is a very useful abstraction, since it allows us to reuse knowledge, rather than having to build new learning models and algorithms from scratch.

Formally, given an input space X and a number of classes K, an unknow distribution D over $X \times [K]$, and a training set D' sampled from D. The multiclass classification computes a function f minimizing $\mathbb{E}_{(x,y)\sim D}[f(x) \neq y]$.

7.1 Extension from binary

This section discusses strategies of extending the existing binary classifiers to solve multi-class classification problems.

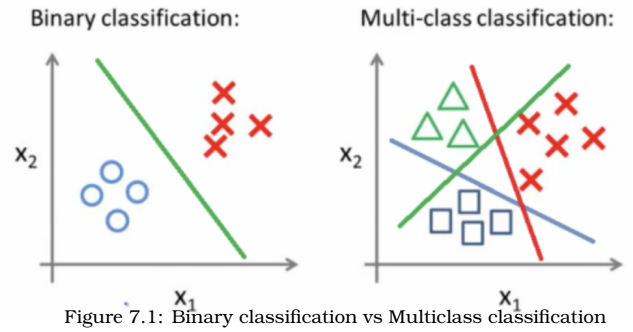


Figure 7.1: Binary classification vs Multiclass classification

7.1.1 k-nearest neighbours

k-nearest neighbors is considered among the oldest non-parametric classification algorithms. To classify an unknown example, the distance from that example to every other training example is measured. The k smallest distances are identified, and the most represented class by these k nearest neighbours is considered the output class labe.

7.1.2 Decistion trees

Decision tree learning is a powerful classification technique. The tree tries to infer a split of the training data based on the values of the available features to produce a good generalization. The algorithm can naturally handle binary or multiclass classification problems. The leaf nodes can refer to any of the K classes concerned.

7.1.3 Perceptron

It is hard to separate three or more classes with just one line, so the perceptron algorithm need some changes to fit the multiclass classification problem.

Using a binary classifier as a black box, can reduce the task of multicalss classification to the binary case? There are multiple ways to decompose the multiclass prediction into multiple binary decisions. Two approaches are:

- One-versus-all,
- All-versus-all.

7.2 One vs All

A very common approach is the **one versus all** technique. To perform OVA, you train K-many binary classifiers $f_1, ..., f_K$. Each classifier sees all of the training data. Classifier f_i receives all examples labeled class i

as positives and all other examples as negatives.

At test time, whichever classifier predicts positive wins, with ties broken randomly.

Algorithm 3: OneVersusAllTrain($D^{\text{multiclass}}$, $BinaryTrain$)

for $i \leftarrow 1, ..., K$ **do**
  $D^{\text{bin}} \leftarrow$ relabel $D^{multiclass}$ so class i is positive, and class $\neg i$ is negative;
  $f_i \leftarrow BinaryTrain(D^{\text{bin}})$;
end
return $f_1, ..., f_K$

In the testing procedure, the prediction of the ith classifier is added to the overall score for class i. Thus, if the prediction is positive, class i gets a vote; if the prediction is negative, every one else (implicitly) gets a vote.

Algorithm 4: OneVersusAllTest($f_1, ..., f_K$, x)

$score \leftarrow \langle 0, 0, ..., 0 \rangle$;
for $i \leftarrow 1, ..., K$ **do**
  $y \leftarrow f_i(x)$;
  $score_i \leftarrow score_i + y$;
end
return $argmax_k\ score_k$

OVA is quite natural and easy to implement. It also works very well in practice, so long as you do a god job choosing a good binary classification algorithm tuning its hyperparameters well.

Its weakness is that it can be somewhat brittle. Intuitively, it is not particularly robus to errors in the underlying classifiers. If one classifier makes a mistake, it is possible that the intire prediction is erroneous. In fact, it is entirely possible that *none* of the K classifier predicts positive (which is actually the worst scenario)!

7.3 All vs All

To develop alternative approeaches, a useful way to think about turning multiclass classification problems into binary classification problems is to think of them like torunaments. You have K teams entering a tournament, but unfortunately the sport they are playing only allows two to compete at a time.

One natural approach is to have every team compete against every other team. The team that wins the majority of its matches is declared the

winner. This is the **all versus all** approach.

The most natural way to think about it is as training $\binom{K}{2}$ classifiers. Say f_{ij} for $1 \leq i < j \leq K$ is the classifier that pits class i against class j. This classifier receives all of the class i examples as positive and all of the class j examples as negative. When a test point arrives, it is run through all f_{ij} classifiers. Every time f_{ij} predicts positive, class i gets a point; otherwise, class j gets a point. After running all $\binom{K}{2}$ classifiers, the class with the most votes wins.

Algorithm 5: AllVersusAllTrain($D^{\text{multiclass}}$, $BinaryTrain$)

$f_{ij} \leftarrow \emptyset, \forall 1 \leq i < j \leq K$;
for $i \leftarrow 1, ..., K-1$ **do**
  $D^{\text{pos}} \leftarrow$ all $x \in D^{\text{multiclass}}$ labeled i;
  for $j \leftarrow i+1, ..., K$ **do**
    $D^{\text{neg}} \leftarrow$ all $x \in D^{\text{pos}}$ labeled j;
    $D^{\text{bin}} \leftarrow \{(x, +1) : x \in D^{\text{pos}}\} \cup \{(x, -1) : x \in D^{\text{neg}}\}$;
    $f_{ij} \leftarrow BinaryTrain(D^{\text{bin}})$;
  end
end
return *all* f_{ij}

Algorithm 6: AllVersusAllTest(all f_{ij}, x)

$score \leftarrow \langle 0, 0, ..., 0 \rangle$;
for $i \leftarrow 1, ..., K-1$ **do**
  for $j \leftarrow i+1, ..., K$ **do**
    $y \leftarrow f_{ij}(x)$;
    $score_i \leftarrow score_i + y$;
    $score_j \leftarrow score_j - y$;
  end
end
return $argmax_k\ score_k$

7.4 OVA vs AVA

In terms of train time, AVA learns more classifiers, however, they are trained on much smaller data, and this tends to make it faster if the labels are equally balanced.

In terms of test time, AVA has more classifiers, so often is slower.

Furthermore, AVA trains on more balanced data sets, and test with more classifiers, therefore has more chances for errors.

If using a binary classifier, the most common thing to do is OVA. Other-

	box	clap	wave	jog	Run	Walk
Box	100	0	0	0	0	0
Clap	0	94	6	0	0	0
Wave	0	1	99	0	0	0
Jog	0	0	0	91	7	2
Run	0	0	0	10	89	1
Walk	0	0	0	0	6	94

Figure 7.2: Confusion matrix example

wise, use a classifier that allows for multiple labels, for example decision trees and k-NN work reasonably well, but other more sophisticated methods work better.

7.5 Evaluation

Selecting the best metrics for evaluating the performance of a given classifier on a certain dataset is guided by a number of consideration including the class-balance and expected outcomes. One particular performance measure may evaluate a classifier rom a single perspective and often fail to measure others. Consequently, there is no unified metric to measure the generalized performance of a classifier.

Two methods, **micro-averaging**, and **macro-averaging** are used to extract a single number for each of the precision, recall and other metrics across multiple classes.

A macro-average calculates the metric autonomously for each class to calculate the average.

In contrast, the micro-average calculates average metric from the aggregate contributions of all classes. Micro-average is used in unbalanced datasets as this method takes the frequency of each class into consideration. The micro average precision, recall, and accuracy scores are mathematically equivalent.

7.5.1 Confusion matrix

A confusion matrix shows the combination of the actual and predicted classes. Each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class. It is a good measure of whether models can account for the overlap in class properties and understand which classes are most easily confused.

8

Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient of the funciont at the current point, because this is the direction of steepest descent. Conversely, stepping in the direction of the gradient will lead to a local maximum of that function; the preocedure is then known as **gradient ascent**.

8.1 Notation

Let's introduce some convenient notation. The indicator function is a function for turning `true` and `false` answers into numbers or counts:

$$1[x] = \begin{cases} 1 & \text{if } x = \texttt{true} \\ 0 & \text{if } x = \texttt{false} \end{cases} \tag{8.1}$$

We use a **vector notation**. We represent an example $f_1, f_2, ..., f_m$ as a single vector, $\boldsymbol{x}$. Similarly, we can represent the weight $w_1, w_2, ..., w_m$ as a single vector, $\boldsymbol{w}$.

The dot-product between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is defined as:

$$a \cdot b = \sum_{j=1}^{m} a_j b_j \tag{8.2}$$

8.2 The optimization framework for Linear Models

We have already seen the perceptron as a way of finding a weight vector $\boldsymbol{w}$ and bias b that do a good job of separating positive training examles from negative training examples. The goal of the perceptron was to find a separating hyperplane for some training data set. Not all data sets are linearly separable. In the case that your training data *isn't* linearly separable, you might want to find the hyperplane that makes the *fewest errors* on the training data. We can write this down as a formal mathematics optimization problem as follows:

$$\min_{w,b} \sum_{n} 1[y_n(\boldsymbol{w} \cdot \boldsymbol{x} + b) > 0] \tag{8.3}$$

The objective function is the thing we are trying to minimize. In this case, the objective function is simply the **error rate** of the linear classifier parametrized by $\boldsymbol{w}$, b.

We know that the perceptron algorithm is guaranteed to find parameters for this model if the data is linearly separable. In other words, if the optimum of 8.3 is zero, then the perceptron will efficiently find parameters for this model.

8.3 Convex Surrogate Loss Functions

There are two equivalent definitions of a convex function. The first is that it's second derivative is always non-negative. The second definition is that any chord of the function lies above it.

Convex functions are nice because they are easy to minimize. This leads to the idea of **convex surrogate loss functions**. Since zero/one loss is hard to optimize, we want to optimize something else, instead. Since convex functions are easy to optimize, we wanto to approximate zero/one loss with a convex function This approximating function will be called a **surrogate loss**. The surrogate losses we construct will always be *upper bounds* on the true loss function: this guarantees that if you minimize the surrogate loss, you are also pushing down the real loss.

There are four common surrogate loss functions, each with their own

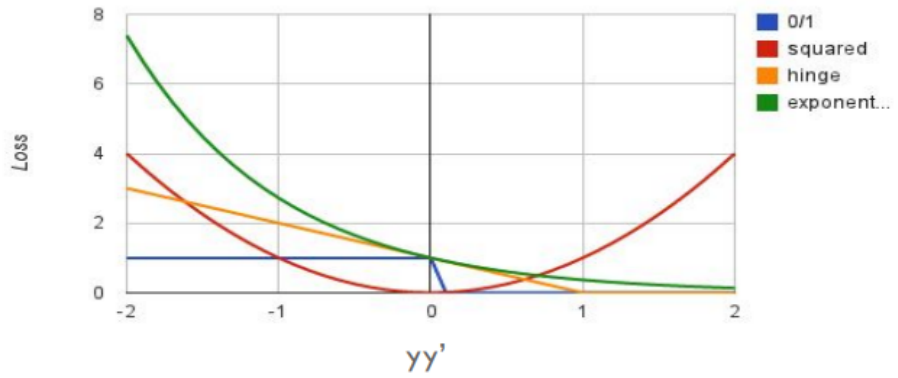


Figure 8.1: Surrogate loss functions

properties:

$$\text{Zero/one} \quad l^{(0/1)}(y, \hat{y}) = 1[y\hat{y} \leq 0] \tag{8.4}$$

$$\text{Hinge} \quad l^{(\text{hin})}(y, \hat{y}) = \max\{0, 1 - y\hat{y}\} \tag{8.5}$$

$$\text{Logistic:} \quad l^{(\log)}(y, \hat{y}) = \frac{1}{\log 2} \log(1 + exp[-y\hat{y}]) \tag{8.6}$$

$$\text{Exponential:} \quad l^{(\exp)}(y, \hat{y}) = exp[-y\hat{y}] \tag{8.7}$$

$$\text{Squared:} \quad l^{(\text{sqr})}(y, \hat{y}) = (y - \hat{y})^2 \tag{8.8}$$

There are two big differences in these loss functions. The first is how *upset* they get by erroneous predictions. In the case of hinge and logistic loss, the growth of the function as $\hat{y}$ goes negative is linear. For squared and exponential loss, it is super-linear. This means that exponential loss would rather get a few examples a little wrong than one example really wrong. The other difference is how they deal with very confident correct predictions. Once $y\hat{y} > 1$, hinge loss does not care any more, byt logistic and exponential loss still think you can do better. On the other hand, squared loss thinks it's just as bad to predict $+3$ on a positive example as it is to predict -1 on a positive example.

8.4 Gradients, math review

A gradient is a multidimensional generalization of a derivative. Suppose you have a function $f : \mathbb{R}^D \to \mathbb{R}$ that takes a vector $\boldsymbol{x} = \langle x_1, x_2, ..., x_D \rangle$ as input and produces a scalar value as output.

You can differentiate this function according to any one of the inputs; for instance, you can compute $\frac{\delta f}{\delta x_5}$ to get the derivative with respect to the fifth input. The **gradient** of f is just the vector consisssting of the derivative f with respect to each of its input coordinates independently, and is denoted ∇f, or, when the input to f is ambiguous, $\nabla_x f$. This is

defined as:

$$\nabla_x f = \langle \frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, ..., \frac{\delta f}{\delta x_D} \rangle \tag{8.9}$$

For example, consider the function $f(x_1, x_2, x_3) = x_1^3 + 5x_1x_2 - 3x_2x_3^2$. The gradient is:

$$\nabla_x f = \langle 3x_1^2 + 5x_2, 5x_1 - 3x_3^2, -6x_2x_3 \rangle \tag{8.10}$$

Note that if $f : \mathbb{R}^{\rightarrow}\mathbb{R}$, thent $\nabla f : \mathbb{R}^D \rightarrow \mathbb{R}^D$. If you evaluate $\nabla f(x)$, this will give you the gradient at $\boldsymbol{x}$, a vector in $\mathbb{R}^D$. This vector can be interpreted as the direction of **steepest ascent**: namely, if you were to traver an infinitesimal amount in the direction of the gradient, you would go uphill the most.

8.5 Optimization with gradient descent

Suppose you are trying to find the maximum of a function $f(\boldsymbol{x})$. The optimizer maintains a current estimate of the parameter of interest, $\boldsymbol{x}$. At each step, it measures the **gradient** of the function it is trying to optimize. This measurement occurs at the current location, $\boldsymbol{x}$. Call the gradient $\boldsymbol{g}$. It then takes a step in the direction of the gradient, where the size of the step is controlled by a parameter η. The complete step is $\boldsymbol{x} \leftarrow \boldsymbol{x} + \eta \boldsymbol{g}$. This is the basic idea of **gradient ascent**.

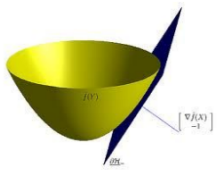


Figure 8.2: Convex function

The opposite of gradient ascent is **gradient descent**. All of our learning problems will be framed as *minimization* problems (trying to reach the bottom of a ditch, rather than the top of a hill). Therefore, descent is the primary approach you will use. One of the major conditions for gradient ascent being able to find the true, **global minimum**, of its objective function is convexity. Without convexity, all is lost.

8.5.1 Algorithm

Algorithm 7: GradientDescent(F, K, $\eta_1, ..., \eta_K$)

$\boldsymbol{z}^{(0)} = \langle 0, 0, ..., 0 \rangle$;
for $k \leftarrow 1, ..., K$ **do**
  $\boldsymbol{g}^{(k)} \leftarrow \nabla_{\boldsymbol{z}} F|_{\boldsymbol{z}^{(k-1)}}$;
  $\boldsymbol{z}^{(k)} \leftarrow \boldsymbol{z}^{(k-1)} - \eta^{(k-1)} \boldsymbol{g}^{(k)}$;
end
return $\boldsymbol{z}^{(K)}$

The function takes as argument the function F to be minimized, the number of iterations K to run and a sequence of learning rates $\eta_1, ..., \eta_K$.

The only real work you need to do to apply a gradient descent method is be able to compute derivatives. Suppose we choose the exponential loss function. We are interested on how much we want to move in the error direction at each iteration: namely, we are interested on the value of $\frac{\delta}{\delta w_i} l(z)$.

$$\frac{\delta l}{\delta w_j} = \frac{\delta}{\delta w_j} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) \tag{8.11}$$

$$= \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) \frac{\delta}{\delta w_j} - y_i(w \cdot x_i + b) \tag{8.12}$$

Now, in order to make it more clear, let's expand the operations for the member $-\frac{\delta}{\delta w_i} y_i(w \cdot x_i + b)$.

$$\begin{aligned} -\frac{\delta l}{\delta w_j} - y_j(w \cdot x_j + b) &= -\frac{\delta}{\delta w_j} y_j(\sum_{j=1}^{m} w_j x_{ij} + b) \\ &= -\frac{\delta}{\delta w_j} y_j(w_1 x_{i1} + w_2 x_{i2} + ... + w_m x_{im} + b) \\ &= -\frac{\delta}{\delta w_j} (y_j w_1 x_{i1} + y_j w_2 x_{i2} + ... + y_j w_m x_{im} + y_j b) \\ &= -y_j x_{ij} \end{aligned}$$

Let's now take this result and return to expand the equation 8.11.

$$\frac{\delta}{\delta w_j} = \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) \frac{\delta}{\delta w_j} - y_i(w \cdot x_i + b) \tag{8.13}$$

$$= \sum_{i=1}^{n} -y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) \tag{8.14}$$

That means that for each example x_i, the exponential update rule is:

$$w_j = w_j + \eta y_i x_{ij} exp(-y_i(w \cdot x_i + b)) \tag{8.15}$$

If you remember from 2 perceptron algorithm, there is a similarity: $w_i \leftarrow w_i + f_i * label$. In practice: $w_j \leftarrow w_j + x_{ij} y_i c$, where $c = \eta exp(-y_i(w \cdot x_i + b))$.

9

Regularization

In our learning objectice, we had a erm correspond to the zero/one loss on the training data, plus a **regularizer** whose goal was to ensure that the learned function did not overfit. If you replace to zero/one loss with a surrogate loss, you obtain the following objective:

$$\min_{w,b} \sum_n l(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) + \lambda R(\boldsymbol{w}, b) \tag{9.1}$$

From the discussion of surrogate loss function, we would like to ensure that R is convex. Otherwise, we will be back to the point where optimization becomes difficult. Beyond that, a common desire is that the components of the weight vecotr should be small. This is a form of **inductive bias**.

9.1 Regularizers

If w_i is reasonably small, this is unlikely to have much of an effect on the classification decision. On the other hand, if w_i is large, this could have a large effect. Another way to say the same thing is to look at the derivative of the predictions as a function of w_i. The derivative of $\boldsymbol{w} \cdot \boldsymbol{x} + b$ with respect to w_i is:

$$\frac{\delta[\boldsymbol{w} \cdot \boldsymbol{x} + b]}{\delta w_i} = \frac{\delta[\sum_d w_d x_d + b]}{\delta w_i} = x_i \tag{9.2}$$

Interpreting the derivative as the rate of change, we can see that the range of change of the prediction funcion is proportional to the individual weights. So if you want the function to change slowly, you want to ensure that the weights stay small.

One way to accomplish this is to simply use the norm of the weight

vector. Namely $R^{(\text{norm})}(\boldsymbol{w}, b) = ||\boldsymbol{w}|| = \sqrt{\sum_d w_d^2}$. This function is convex and smooth, which makes it easy to minimize.

An alternative to using the sum of squared weights is to use the sum of absolute weights: $R^{(\text{abs})}(\boldsymbol{w}, b) = \sum_d |w_d|$.

9.2 p-norm

The line of thinking leads to the general concept of p-norms. This is a family of norms that all have the same generalflavor. We write $||\boldsymbol{w}||_p$ to denote the p-norm of $\boldsymbol{w}$.

$$||\boldsymbol{w}||_p = R(\boldsymbol{w}, b) = \sqrt[p]{\sum_{w_j} |w_j|^p} \tag{9.3}$$

You can check that the 2-norm exactly corresponds to the usual Euclidean norm, and the 1-norm corresponds to the absolut regularizer described above.

When p-norms are used to regularize weight vectors, the interting aspect is how they trade-off multiple features. To see the behavior of p-norms in two dimensions, we can plot their **contour**.

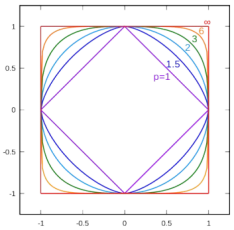


Figure 9.1: p-norms visualized

Figure **??** shows the contours for the same p-norms in two dimensions. Each line denotes two-dimensional vectors to which this norm assignes a total value of 1. By changing the value of p, you can interpolate between a square, down to a circle.

In general, smaller values of p prefer sparser vectors. You can see this by noticing that the contours of small p-norms stretch out along the axes.

9.3 Minimizing with a regularizer

We know how to solve convex minimization problems using gradient descent.

$$\min_{\boldsymbol{w},b} \sum_{i=1}^{n} loss(yy')$$

If we can ensure that the loss with the regularizer is convex, then we could still use gradient descent to solve convex minimization problems:

$$\min_{\boldsymbol{w},b} \sum_{i=1}^{n} loss(yy') + \lambda R(\boldsymbol{w}) \tag{9.4}$$

Equation 9.4 is convex as long as both the loss function and the regularizer are convex.

9.3.1 Model-based machine learning

There are the three step for the model-based machine learning:

- Pick a model;

$$b + \sum_{j=1}^{n} w_j f_j = 0 \tag{9.5}$$

- Pick a criteria to optimize, namely, pick the objective function;

$$\sum_{i=1}^{n} exp(-y_i(\boldsymbol{w} \cdot x_i + b)) + \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{9.6}$$

- Develop a learning algorithm;

$$\min_{\boldsymbol{w},b} \sum_{i=1}^{n} exp(-y_i(\boldsymbol{w} \cdot x_i + b)) + \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{9.7}$$

Note that the loss function penalizes examples where the prediction is different than the label; and regularizer penalizes large weights. This function is convex, thus is allowing us to use gradient descent.

$$\frac{\delta}{\delta w_j}\text{objective} = \frac{\delta}{\delta w_j} \sum_{i=1}^{n} exp(-y_i(\boldsymbol{w} \cdot x_i + b)) + \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{9.8}$$

$$= -\sum_{i=1}^{n} y_i x_{ij} exp(-y_i(\boldsymbol{w} \cdot x_i + b)) + \lambda w_j \tag{9.9}$$

$$\tag{9.10}$$

That means, that the gradient descent algorithm moves a small amount in a chosen dimension towards decreasing loss using the derivative.

$$w_j = w_j - \eta \frac{\delta}{\delta w_j}(l(\boldsymbol{w}) + R(\boldsymbol{w}, b)) \tag{9.11}$$

$$w_j = w_j + \eta \sum_{i=1}^{n} y_i x_{ij} exp(-y_i(\boldsymbol{w} \cdot x_i + b)) - \eta\lambda w_j \tag{9.12}$$

10

Support Vector Machines

Let's start by thinking back to the original goal of linear classifiers: to find a hyperplane that separates the positive training examples from the negative ones. Figure **??** shows some data and three potential hyperplane that separates the red training examples from the blue one. Which one do you like the best? Most likely you chose the green hyperplane. And most likely you chose it because it was furthest away from the closest training points.

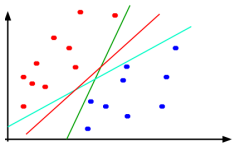


Figure 10.1: p-norms visualized

In other words, it has a large **margin**. The desire for hyperplanes with large margins is a perfect example of an inductive bias.

Following this line of thinking leads us to the **Support Vector Machine** (SVM). this is simply a way of setting up an optimization problem that attempts to find a separating hyperplane with as large margin as possible.

10.1 Large margin classifiers

The **margin** of a single data point is defined to be the distance from the data point to a decision boundary. Note that there are many distances and decision boundaries that may be appropriate for certain datasets and goals.

A **margin classifier** is a classifier which is able to give an associated distance from the decision boundary for each example. For instance, if a linear classifier is used, the distance of an example from the separating hyperplane is the margin of that example.

There are many huperplanes that might classify the data. One reason-

able choice as the best hyperplane is the one that represents the largest margin between the two classes. So we choose the hyperplane so that the distance from it to the nearest data point on each side is maximized. If such a hyperplane exists, it is known as the **maximum margin hyperplane** and the linear classifier it defines is known as a **maximum margin classifier**, or equivalently, the perceptron of optimal stability.

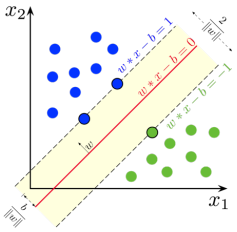


Figure 10.2: Maximum-margin hyperplane and margins for an SVM trained with samples from two classes.

10.1.1 Support vectors

Support vectors are the elements of the training set that would change the position of the dividing hyperplane if removed. They are critical elements of the training set.

The problem of finding the optimal hyperplane is an optimization problem and can be solved by optimization techniques. For n dimensions, there will be at least $n+1$ support vectors. We can see in Figure **??** that there are 3 support vectors, 2 are of the blue class, and 1 of the green class.

10.1.2 Maximizing the margin

Given the hyperplane in Figure **??**, we have the margins described by the equation $\boldsymbol{w} \cdot \boldsymbol{x} - b = a$, where $a = 1$ for the blue margin, and $a = -1$ for the green margin.

The measure for the margin is defined by

$$\frac{\boldsymbol{w} \cdot x_i + b}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||} \tag{10.1}$$

Our goal is now to maximize the margin, namely, to select the hyperplane with the largest margin where the points are classified correctly and outside the margin.

Let's setup the problem as a constrained optimization problem, subject to $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 \forall i$

$$\max_{a,b} margin(\boldsymbol{w}, b) = \max_{w,b} \frac{1}{||\boldsymbol{w}||} \tag{10.2}$$

$$= \min_{w,b} ||\boldsymbol{w}|| \tag{10.3}$$

Maximizing the margin is equivalent to minimize the norm of the weights (subject to separating constraints). The minimization criterion wants $\boldsymbol{w}$ to be as small as possible, while the constraints make sure that the data is separable.

The **support vector machine problem** is a version of a **quadratic optimization problem**, which wants to maximize/minimize a quadratic function subject to a set of linear constraints.

$$\min_{\boldsymbol{w},b} ||\boldsymbol{w}||^2 \tag{10.4}$$

10.2 Soft margin classification

For the very high dimensional problems common in classification, sometimes the data are linearly separable. But in general case they are not, and even if they are, we might prefer a solution that better separates the bulk of the data while ignoring a few weird noise.

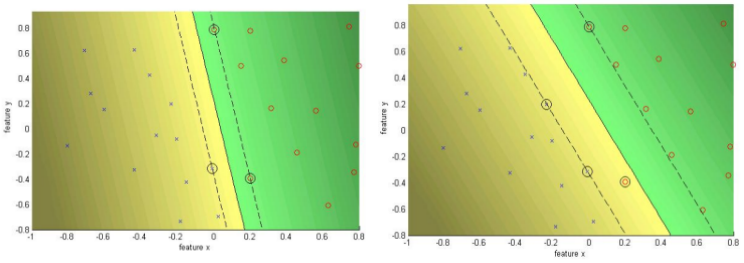
If the training set D is not linearly separable, the standard approeach is to allow the fat decision margin to make a few mistakes (some points - outliers or noisy example - are inside or on the wrong side of the margin). We then pay a cost for each misclassified example, which depends on how far it is from meeting the marin requirements. To implement this, we introduce *slack variables* ζ_i. A non-zero value for ζ_i allows $\boldsymbol{x}_i$ to not meet the margin requirement at a cost proportional to the value of ζ_i.

The formulation of the SVM optimization problem with slack variables is:

Find $\boldsymbol{w}$, b and $\zeta_i \geq 0$ such that:

- $\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_i \zeta_i$ is minimized
- and $\forall\{(\boldsymbol{x}_i, y_i)\}$, $y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \zeta_i$

The optimization problem is then a trading off how fat it can make the margin versus how many points have to be moved around to allow this margin. the margin can be less than 1 for a point $\boldsymbol{x}_i$ by setting $\zeta_i > 0$, but then one pays a penalty of $C\zeta_i$ in the minimization for having done that. The som of the ζ_i gives an upper bound on the number of training errors.



(a) $C = \infty$ hard margin

(b) $C = 10$ soft margin

Figure 10.3: Hard margin vs soft margin example.

Soft-margin SVMs minimize training error traded off against margin.

The parameter C is a *regularization term*, which proivdes a way to control overfitting: small C allows constraints to be easily ignored to large margin; large C makes constraints hard to ignore $\rightarrow$ narrow margin. If $C = \infty$ enforces all constraints: hard margin.

The slack variable can have two different values:

$$\zeta_i = \begin{cases} 0 & \text{if } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 \\ 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) & \text{otherwise} \end{cases} \tag{10.5}$$

$$= \max(0, 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)) \tag{10.6}$$

$$= \max(0, 1 - yy') \tag{10.7}$$

Note that the last equality is the hinge loss, that is actually our loss, so we can rewrite our optimization problem as follows

$$\min_{\boldsymbol{w},b} ||\boldsymbol{w}||^2 + C \sum_i \max(0, 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b)) \tag{10.8}$$

That looks like the expression:

$$arg \min_{\boldsymbol{w},b} \sum_i^n l(yy') + \lambda R(\boldsymbol{w}b) \tag{10.9}$$

10.3 Non linearly separable data

We have gone though the understanding of **Support Vectors** and **Margins**. Then used the concepts to build the objective functions for soft margin classifier. We have have also learned the objective function, that is defined as **Primal Problem**.

However our final goal is to solve non-linear SVM, where primal problem is not helpful.

10.3.1 Dual problem

In mathematical optimization theory, duality means that optimization problems may be viewed from either of two perspectives, the primal problem or the dual problem. The solution to the dual problem provides a lower bound to the solution of the primal (minimization) problem.

Quadratic optimization problems are a well-known class of mathematical programming problems for which several algorithms exist. One possible solution involves constructing a dual problem where a Lagrange multiplier α_i is associated with every inequality constraint in the primal problem:

Find $\alpha_1, ..., \alpha_n$ such that $\sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$ is maximized and

- $\sum_i \alpha_i y_i = 0$

- $\alpha_i \geq 0 \forall \alpha_i$

Given a solution $\alpha_1 ... \alpha_n$ to the dual problem, the solution to the primal is:

$$\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i$$
$$b = y_k - \sum_i \alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x}_k \text{ for any } \alpha_k > 0$$

Each non-zero α_i indicates that the corresponding $\boldsymbol{x}_i$ is a support vector. Then the classifying function is:

$$f(\boldsymbol{x}) = \sum_i \alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + b \tag{10.10}$$

The solution relies on an inner product between the test point $\boldsymbol{x}$ and the support vectors $\boldsymbol{x}_i$. Also solving the optimization problem involves the computing of the inner products between all training points.

Dual problem is similar in the non separable case, but notice the constraints.

Find $\alpha_1 ... \alpha_N$ such that $\sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$ is maximized and

- $\sum_i \alpha_i y_i = 0$
- $0 \geq \alpha_i \geq C$ for all α_i

Again, $\boldsymbol{x}_i$ with non-zero α_i represent support vectors.

10.3.2 Kernel trick

Suppose now that we would like to learn a nonlinear classification rule which corresponds to a linear classification rule for the transformed data points $\phi(\boldsymbol{x}_i)$. The linear classifier relies on inner product between vectors

$$K(\boldsymbol{x}_i, \boldsymbol{y}_j) = \boldsymbol{x}_i^T \boldsymbol{y}_j \tag{10.11}$$

If every datapoint is mapped into high-dimensional space via some transformation $\Phi : \boldsymbol{x} \to \phi(\boldsymbol{x})$, the inner product becomes

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j) \tag{10.12}$$

A kernel function is a function that is equivalent to an inner product in some feature space. It implicitly maps data to a high-dimensional space (without the need to compute each $\phi(\boldsymbol{x})$ explicitly).

For some function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ checking that $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$ can be cumbersome.

Mercer's theorem

- Every semi-positive definite symmetric function is a kernel.

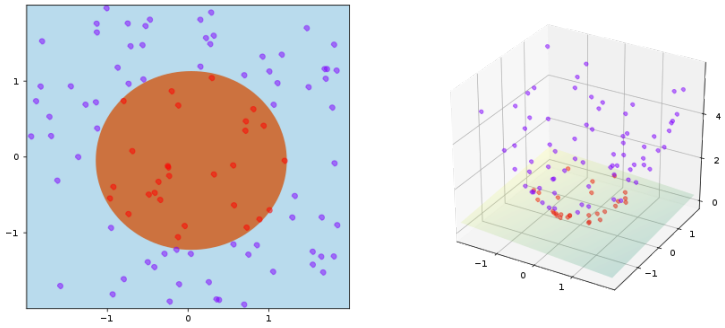


Figure 10.4: A training example of SVM with kernel given by $\phi((a,b)) = (a,b,a^2+b^2)$

- Semi-positive definite symmetric functions correspond to a semi-positive definite symmetric Gram matrix

$$K = \begin{bmatrix} K(\boldsymbol{x}_1,\boldsymbol{x}_1) & K(\boldsymbol{x}_1,\boldsymbol{x}_2) & ... & K(\boldsymbol{x}_1,\boldsymbol{x}_n) \\ K(\boldsymbol{x}_2,\boldsymbol{x}_1) & K(\boldsymbol{x}_2,\boldsymbol{x}_2) & ... & K(\boldsymbol{x}_2,\boldsymbol{x}_n) \\ ... & ... & ... & ... \\ K(\boldsymbol{x}_n,\boldsymbol{x}_1) & K(\boldsymbol{x}_n,\boldsymbol{x}_2) & ... & K(\boldsymbol{x}_n,\boldsymbol{x}_n) \end{bmatrix} \tag{10.13}$$

Kernels

[topsep=0pt, partopsep=0pt]

- Linear: $K(\boldsymbol{x}_i,\boldsymbol{x}_j) = \boldsymbol{x}_i^T\boldsymbol{x}_j$
- Polynomial of power p: $K(\boldsymbol{x}_i,\boldsymbol{x}_j) = (1+\boldsymbol{x}_i^T\boldsymbol{x}_j)^p$
- Gaussian (radial-basis function): $K(\boldsymbol{x}_i,\boldsymbol{x}_j) = e^{\frac{||\boldsymbol{x}_i-\boldsymbol{x}_j||^2}{2\sigma^2}}$
  - Mapping $\Phi : \boldsymbol{x} \to \phi(\boldsymbol{x})$, where $\phi(\boldsymbol{x})$ is infinite-dimensional: every point is mapped to a function (Gaussian); combination of functions for support vectors is the separator.
- Higher-dimensional space still has *intrinsic* dimensionality d (the mapping is *onto*), but linear separators in it correspond to non-linear separators in original space.

11

Ranking

Ranking is the application of machine learning, typically supervised or reinforcement learning, in the construction of ranking models for information retrieval systems. Training data consists of lists of items with some partial order speicified between items in each list. This order is typically induced by giving a numerical or ordinal score or a binary judgement for each item.

The ranking model purposes to rank, i.e. producing a permutation of items in new, unseen lists in a similar way to rankings in the training data.

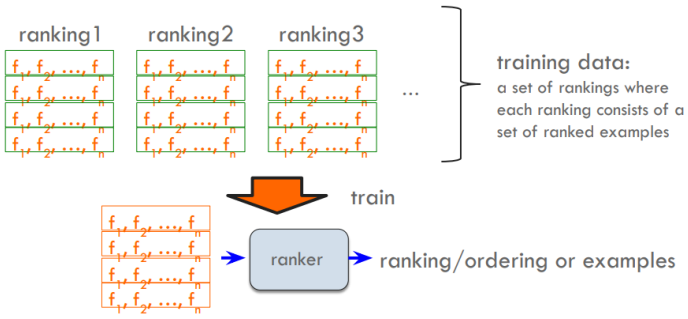


Figure 11.1: Ranking problems general schema.

Traditional machine learning solves a prediction problem on a single instance at a time. The aim of traiditional machine learning is to come up with a class or a single numerical score for that instance. Instead, ranking solves a ranking problem on a list of items. The aim of ranking is to come up with optimal ordering of those items. Ranking doesn't care much about the exact score that each item gets, but cares more about the relative ordering among the items.

11.1 Multilabel classification

In machine learning, **multilabel classification** and the strongly related problem of **multi-output classification** are varients of the classification problem where multiple labels may be assigned to each instance. Multilabel classification is a generalization of multiclass classification, which is the single-label problem of categorizing instances into precisely one of more than two classes; in the multi-label problem there is no constraint on how many of the classes the instance can be assigned to.

In multiclass classification each example has one label and exactly one label. In multilabel classification each example has **zero or more labels**.

Formally, multilabel classification is the problem of finding a model that maps inputs x to binary vectors y (assigning a value of 0 or 1 for each element - label - in y).

Multilabel applications examples are:

- Video surveillance - reidentification;
- Scene analysis - visual localization;
- Image annotation;
- Document topics;
- Medical diagnosis.

11.2 Ranking problems

Suppose we start a new web search company. Like other search engines, a userinputs a query and a set of documents is retrieved. Our goal is to rank the resulting documents based on relevance to the query. The ranking problem is to take a collection of items and sort them according to some notion of preference. One of the trickiest parts of doing ranking through learning is to properly define the loss function.

11.2.1 Black box approach

Given a generic binary classifier, we can use it to solve our new problem. We need to train a classifier to decide if the first input is better than the

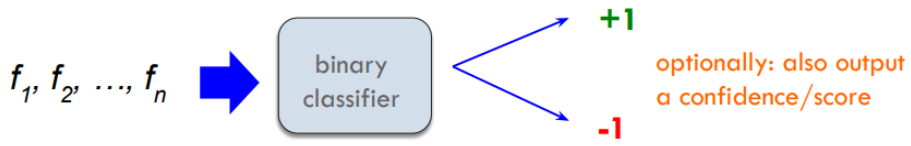


Figure 11.2: Black box approach to ranking.

second: we have to consider all the possible pairings of the examples in

a ranking and label them as positive if the first example is higher ranked, otherwise we rank them as negative.

The problem here is that our binary classifier takes only one example as input. How can we perform the following classification?

$$a_1, a_2, ..., a_n; b_1, b_2, ..., b_n \rightarrow f'_1, f'_2, ...f'_n \tag{11.1}$$

We need features that compare the two examples a_i and b_i, this new vector is called **combined feature vector**. There are many approaches that depend on domain and on the classifier, two common approaches to do this are:

- compare by difference:
$$f'_i = a_i - b_i$$
- compare the first to the second example:
$$f'_i = \begin{cases} 1 & \text{if } a_i > b_i \\ 0 & \text{otherwise} \end{cases}$$

11.2.2 Preference function

For each query, we are also given a collection of documents, together with a desired ranking over those documents. We'll assume that we have N-many queries and for each query we have M-many documents. The goal is to train a binary classifier to predict a **preference function**. Given a query q and two documents d_i and d_j, the classifier should predict whether d_i should be preferred to d_j with respect to the query q.

11.3 Algorithm

Algorithm 8: NaiveRankTrain($RankingData$, BinaryTrain)

```
D ← [ ];
for n ← 1 to N do
    for i, j ← 1 to M and i ≠ j do
        if i is preferred to j on query n then
            D ← D.append(x_nij, +1);
        else
            if j is preferred to i on query n then
                D ← D.append(x_nij, −1);
            end
        end
    end
end
return BinaryTrain(D)
```

Algorithm 9: NaiveRankTest(f, $\hat{x}$)

$score \leftarrow \langle 0, 0, ..., 0 \rangle$;
for $i, j \leftarrow 1$ *to* M *and* $i \neq j$ **do**
  $y \leftarrow f(\hat{x}_{ij})$;
  $score_i \leftarrow score_i + y$;
  $score_j \leftarrow score_j - y$;
end
return $score$.*sort()*

11.4 Bipartite Ranking

Bipartite ranking algorithms are useful for bipartite ranking problems. A **Bipartite ranking problem** is one in which you are trying to predict a binary response, for instance "is this document relevant or not?".

The only goal is to ensure that all the relevant documents are ahead of all the irrelevant documents. There is no notion that one relevant document is more relevan than another.

For non-bipartite ranking problems, more sophisticated methods can be used.

11.5 Weighted binary classification

For non-bipartite ranking problems, we can do better. First, when the preferences thath we get at training time are more nuanced than "relevant or not", we can incorporate these preferences at training time. Effectively, we want to give a higher weight to binary problems that are very different in terms of preference than others. Second, rather than producing a list of scores and then calling an arbitrary sorting algorithm, we can actually use the preference function as the sorting function inside our own implementation of quicksort.

Define a ranking as a function σ that maps the objects we are ranking to the desired position in the list $1, 2, ..., M$. If $\sigma_u < \sigma_v$ then u is preferred to v. Given data with observed rankings σ, our goal is to learn to predict rankings for new objects, $\hat{\sigma}$. We define Σ_M as the set of all ranking functions over M objects. We also wish to express the fact that making a mistake on some pairs is worse than making a mistake on others - this will be encoded in a cost function ω, where $\omega(i, j)$ is the cost for accidentally putting something in position j when it should have gone in position i. To be a valid cost function, ω should be *symmetric*, *monotonic*: if $i < j < k$ then $\omega(i, j) \leq \omega(i, k)$, *satisfy the triangle inequality*: $\omega(i, j) + \omega(j, k) \geq \omega(i, k)$.

11.5.1 ω-Ranking

Given an input space $\mathcal{X}$, an unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \Sigma_M$, and a training set D sampled from $\mathcal{D}$.
We want to compute a function $f : \mathcal{X} \to \Sigma_M$ minimizing:

$$\mathbb{E}_{(\boldsymbol{x},\sigma)\sim\mathcal{D}} \left[\sum_{u \neq v} [\sigma_u < \sigma_v][\hat{\sigma}_v < \hat{\sigma}_u] \omega(\sigma_u, \sigma_v) \right] \tag{11.2}$$

where $\hat{\sigma} = f(\boldsymbol{x})$.

In this definition, the only complex aspect is the loss function. This loss sums over all pairs of objects u and v, but the predicted ranking $\hat{\sigma}$ prefers v to u, then you incur a cost of $\omega(\sigma_u, \sigma_v)$.

Depending on the problem you care about, you can set ω to many "standard" options. If $\omega(i, j) = 1$ whenever $i \neq j$, then you achieve the Kemeny distance measure, which simly counts the number of airwise misordered items. We may only care about getting the top K predictions correct. In this case, we define:

$$\omega(i, j) = \begin{cases} 1 & \text{if } \min\{i, j\} \leq K \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{11.3}$$

In this case, only errors in the top K elements are penalized. Swapping items 55 and 56 is irrelevent (for $K < 55$).

Algorithm 10: RankTrain(D^{rank}, ω, BinaryTrain)

```
D^bin ← [ ];
for (x, σ) ∈ D^rank do
    for u ≠ v do
        y ← sign(σ_v − σ_u));
        w ← ω(σ_u − σ_v);
        D^bin ← D^bin ⊕ (y, w, x_uv);
    end
end
return BinaryTrain(D^bin)
```

At test time, instead of predicting scores and then sorting hte list, as in the previous algorithm, we run the quicksort algorithm, using the learned function as a comparison function. In practice at teach step a pivot p is chosen. Every object u is compared to p using the learned function and sorted to left or right. The difference between this algorithm and quicksort is that the comparison function is *probabilistic*. If f outputs probabilities, for instance it predicts that u has an 80% probability of being better than p, then it puts it on the left with 80% probability and on the right with 20% probability.

Algorithm 11: RankTest(f, $\hat{x}$, obj)

```
if obj contains 0 or 1 elements then
    return obj
else
    p ← randomly chosen object in obj;
    left ← [ ];
    right ← [ ];
    for u ∈ obj \ {p} do
        ŷ ← f(x_up);
        if uniform random variable < ŷ then
            left ← left ⊕ u;
        else
            right ← right ⊕ u;
        end
    end
    left ← RankTest(f, x̂, left);
    right ← RankTest(f, x̂, right);
end
return left ⊕ ⟨p⟩ ⊕ right
```

This algorithm is better than the naive algorithm in at least two ways. First, it only makes $O(M \log_2 M)$ calls to f rather than $O(M^2)$ calls in the naive case. Second, it achieves a better error bound.

12

Neural Networks

Artificial neural networks are computing systems inspired by the biological neural networks that constitute animal brains. An ANN is based on a collection of connected units or nodes called **artificial neurons**, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives a signal then processes it and can signal neurons connected to it. The signal at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called **edges**. Neurons and edges typically have a **weight** that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer, to the last layer, possibly after traversing the layers multiple times.

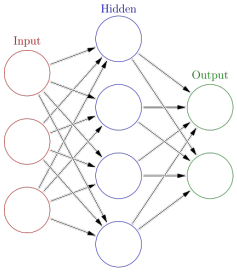


Figure 12.1: An artificial neural network as an interconnected group of nodes.

The idea of neural networks began unsurprisingly as a model of how neurons in the brain function, termed 'connectionism' and used connected circuits to simulate intelligent behaviour .In 1943, portrayed with a simple electrical circuit by neurophysiologist Warren McCulloch and mathematician Walter Pitts. Donald Hebb took the idea further in his book, The Organization of Behaviour (1949), proposing that neural pathways strengthen over each successive use, especially between neurons that tend to fire at the same time thus beginning the long journey towards quantifying the complex processes of the brain.

Around this time, Frank Rosenblatt, a psychologist at Cornell, was working on understanding the comparatively simpler decision systems present in the eye of a fly, which underlie and determine its flee response. In an attempt to understand and quantify this process, he proposed the idea of a Perceptron in 1958, calling it **Mark I Perceptron**. It was a system with a simple input output relationship, modeled on a McCulloch-Pitts neuron, proposed in 1943 by Warren S. McCulloch, a neuroscientist, and Walter Pitts, a logician to explain the complex decision processes in a brain using a linear threshold gate. A McCulloch-Pitts neuron takes in inputs, takes a weighted sum and returns '0' if the result is below threshold and '1' otherwise.

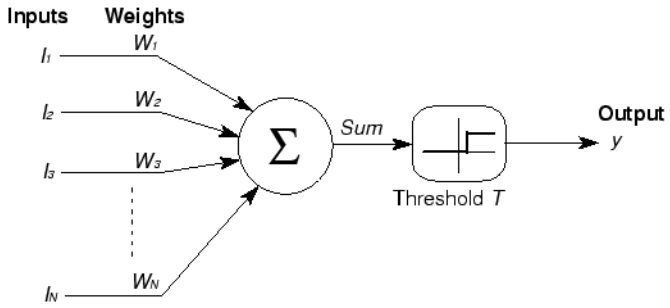


Figure 12.2: A McCulloch-Pitts neuron.

A major drawback? This perceptron could only learn to separate linearly separable classes, making the simple but non-linear exclusive-or circuit an insurmountable barrier.

12.1 From Perceptron to Deep Networks

In 1969 it was formally proved that perceptron cannot be used for non-linearly separable problems. In particular, it has been proved that the **XOR** (exclusive or) cannot be solved with the perceptron. We can realize the XOR as a function that map all pairs of input x_1 and x_2 to an output y with the following scheme:

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

Table 12.1: XOR truth table

Despite the messy and somewhat dis-satisfactory advent of the use of Machine Learning to quantify decision systems apart from the brain,

today's artificial neural networks are nothing more than **several layers of these perceptrons**.

Multi-Layer Perceptron

The idea of the multi-layer perceptron is to densely connect artifical neurons to realize compositions of non-linear functions. THe information is propagated from the inputs to the outputs. No cycles between outputs and inputs (DAG). Compute one or more non-linear function. Computation is carried out by composition of some number of algebraic functions implemented by the connections, weights and biases of the hidden and output layers. Hidden layers compute intermediate representations.

One of the problems that arose was with the impractically long runtimes required for running these networks given that this was the 60s apart from its inability to learn simple boolean exclusive-or circuits.

Multi-layer networks use a variety of learning techniques, the most popular being backpropagation. Here, the output values are compared with the correct answer to compute the value of some predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small. In this case, one would say that the network has learned a certain target function.

12.1.1 Backpropagation

Backpropagation, a method devised by researchers since the 60's and continuously developed on well into the AI winter, was an intuition based method that attributed reducing significance to each event as one went farther back in the chain of events. Backpropagation along with Gradient Descent forms the backbone and powerhouse of neural networks. While Gradient Descent constantly updates and moves the weights and bias towards the minimum of the cost function, backpropagation evaluates the gradient of the cost w.r.t. weights and biases, the magnitude and direction of which is used by gradient descent to evaluate the size and direction of the corrections to weights and bias parameters.

In a nutshell

1. Forward propagation: sum inputs, produce activations, feed-forward.
2. Error estimation.
3. Back propagate the error signal and use it to update weights.

The main idea is that given training samples $T = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, adjust the weights of the network Θ such that a cost function is mini-

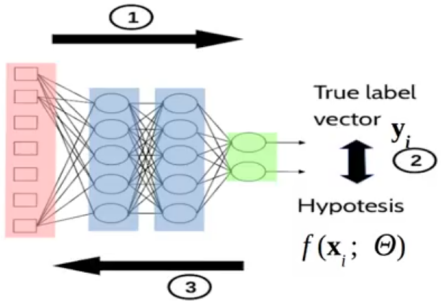


Figure 12.3: Backpropagation scheme.

mized

$$\min_{\Theta} \sum_{i} L(y_i, f(x_i; \Theta)) \tag{12.1}$$

Choose the loss function (e.g. L2), update the weights of each layer with gradient descend, and use backpropagation of the error signal to compute the gradient efficiently.

12.2 Feedforward Networks

A feedforward neural network is an artificial netowkr wherein connections between the nodes do not form a cycle. The feedforward neural network was the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction — forward — from the input nodes, through the hidden nodes (if any) and to the output nodes.

Feedforward neural networks are the quintessential deep learning models. The goal of a feedforward network is to approximate some function $f^* : \mathcal{X} \to \mathcal{Y}$. For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\Theta})$ and learns the value of the parameters $\boldsymbol{\Theta}$ that result in the best function approximation.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$.

These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called **first layer** of the network, $f^{(2)}$ is the **second layer** of the network, and so on. The overall length of the chain gives the **depth** of the model. The final layer is the **output layer** of the model.

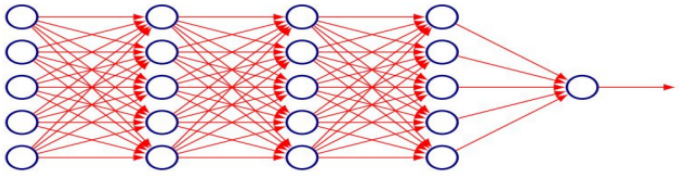


Figure 12.4: The function f is a composition of multiple functions.

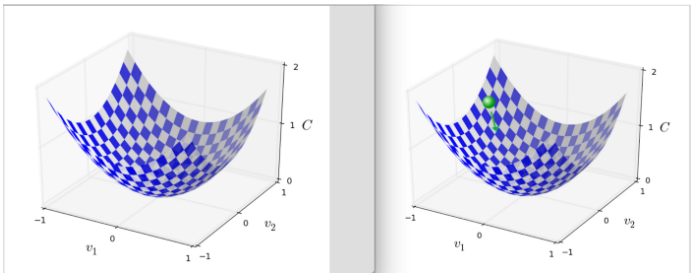


Figure 12.5: Ball diagram depicting the visualization of how gradient based learning occur.

12.2.1 Training

During the training, we drive $f(\boldsymbol{x};\Theta)$ to match $f^*(\boldsymbol{x})$. The training data provides us with noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points.

The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithms must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers, they are called **hidden layers**.

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters.

In order to apply gradient descent to train our neural network, we need to specify our model, that means that we need to specify the input and

output layers are made, and also the cost function.

Cost Function

The cost function, for any other machine learning algorithm, is the loss function that measures the discrepancy between the prediction of the model given the current parameters and the ground truth labels.

It is common for neural network, when we have classification problems, to apply cost function that is called **cross-entropy loss**. It is common for neural networks to transform the outputs (scores/logits) to probability, then we can think that a good cost function can be this:

$$\mathcal{L}_i = -\sum_k y_k \log(S(l_k)) = -log(S(l)) \tag{12.2}$$

That is the cross-entropy loss. The choice of the loss is related to the choice of the input unit. In this case we have decised to put a normalization operator that is the softmax loss $S(l_i) = \frac{e^{l_i}}{\sum_k e^{l_l}}$, where l_i are the scores. Linear layer produces unnormalized log probabilities, but it is desirable to produce normalized probabilities in the output layer, that's the reason why we use the softmax.

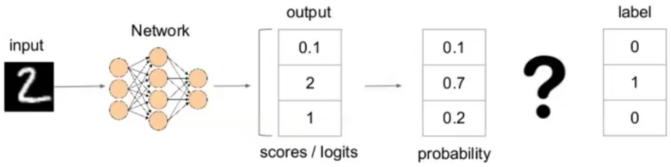


Figure 12.6

Activation function

The activation functions are implemented inside the **hidden units**, because here we compute affine transformation $z = W^T x + b$, and the non-linearity is applied that we define as $h(z)$. There are a lot of choices for h, so we have to choose it wisely.

The design of the hidden units is an active area of research. However the most popular is the **ReLU**, where $h(x) = \max(0, x)$. Here, the gradient is 0 or 1, similar to the linear units, easy to optimize. In fact it gives 1 if the gradient unit is active.

Potential problems with the ReLU activation function are that it is non-differentiable at zero, however, it is differentiable anywhere else, and the value of the derivate at zero can be arbitrarily chose to be 0 or 1; it is not zero-centered and it is unbounded; furthermore, the bigger problem is the **dying ReLU problem**: neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. In this state, no gradients flow backward through the neuron, and so the neuron becomes stuck in a perpetually inactive state and *dies*. This is a

form of the vanishing gradient problem. In some cases, large numbers of neurons in a network can become stuck in dead states, effectively decreasing the model capacity. This problem typically arises when the learning rate is set too high.

In order to mitigate these problems, there are some variants,for example: the Leaky ReLU, with $y_i = a_i x_i$; the Randomized Leaky ReLU, with $y_{ji} = a_{ji} x_{ji}$.

Other variation of activation function are classical squashing type of non-linearity, for example the **Sigmoid**, with $h(x) = \frac{1}{1+e^{-x}}$ and the **Hyperbolic tangent**, with $h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

Architecture

How do we decide the depth and the width of a neural network? There are not many theoretical advice for that, the only relevant result is that 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units).

The theorem also holds for other non-linear functions. The implication of this theorem is that regardless of function we are trying to learn, we know a large MLP can represent this function.

However, we are not guaranteed that our training algorithm will be able to learn that function.

12.3 Backpropagation

Backpropagationis a widely used algorithm for training feedforward neural networks. In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a signle input-output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes it feasible to use gradient methods for training multilayer networks, updating weights to minimize loss; gradient descent, or variants, are commonly used. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this is an example of dynamic programming.

The idea of backpropagation is a flow starting from the input layers and goes on through the hidden layers until they reach the output layer. Then the computed prediction of the network is compared with the ground truth and the error signal is computed. The error signal is

backpropagated to the network in order to compute the gradient and to update all the weights associated to this connection.

There are three main steps:

1. **Feedforward propagation**: accept input x, pass through intermediate stages and obtain output;
2. Use the computed output to compute a **scalar cost** depending on the loss function;
3. **Backpropagation** allows information to flow backwards from cost to compute the gradient.

In backpropagation we use **gradient descent**: we need error derivatives for all the weights in the net.

$$w_{11}^{(1)} := w_{11}^{(1)} - \eta \frac{\delta L}{\delta w_{11}^{(1)}} \tag{12.3}$$

From the training data we do not know what the hidden units should do, but we can compute hwo fast the error changes as we change a hidden activity.

Each hidden unit can affect many output units and have separate effeccts on error, then we combine these effects. We can compute the **error derivatives for hidden units efficiently**. Once we have error derivatives for hidden activities, it is easy to get error derivatives for weights going in.

12.3.1 Step 1: Feedforward propagation

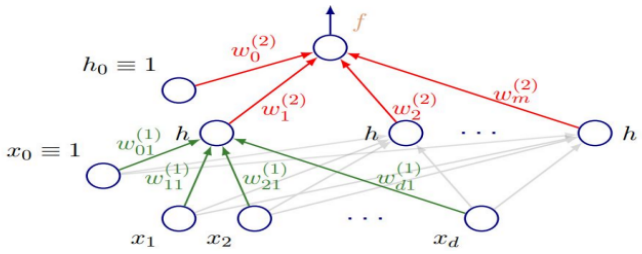


Figure 12.7: Feedforward operation.

In this step we go from our input to the output, with respect to Figure 12.7

$$\hat{y}(\boldsymbol{x}; \boldsymbol{w}) = f\left(\sum_{j=1}^{m} w_j^{(2)} h\left(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_0^{(2)}\right) \tag{12.4}$$

12.3.2 Step 2: Compute error and train

The error of the network on a training set is

$$L(X; \boldsymbol{w}) = \sum_{i=1}^{N} \frac{1}{2}(y_i - \hat{y}(\boldsymbol{x}_i; \boldsymbol{w}))^2 \tag{12.5}$$

There is no closed-form solution. To train, we need to apply the gradient descent.

It means that we need to evaluate the derivative of L on a single example. We can consider a simple linear model for output $\hat{y} = \sum_j w_j xij$:

$$\frac{\delta L(\boldsymbol{x}_i)}{\delta w_j} = (\hat{y}_i - y_i)xij \tag{12.6}$$

12.3.3 Step 3: Backpropagation

We need to compute the derivative of the error with respect to the weights, and these weights are the weights associated to the intermediate layers. The general unit activation in a multilayer network is

$$z_t = h\left(\sum_j w_{jt} z_j\right) \tag{12.7}$$

In forward propagation we calculate for each unit $a_t = \sum_j w_{jt} z_j$. The loss L depends on w_{jt} only through a_t. We are interested on computing

$$\frac{\delta L}{\delta w_{jt}} = \frac{\delta L}{\delta a_t}\frac{\delta a_t}{\delta w_{jt}} = \frac{\delta L}{\delta a_t} z_j \tag{12.8}$$

Now we are left of the problem of computing $\frac{\delta L}{\delta a_t} z_j$. The key idea of backpropagation is that it can be computed recursively starting from the final layer to the earliest layer.

In the case of the last layer, $\frac{\delta L}{\delta a_t} z_j$ corresponds to $\hat{y} - y$, this is called the final error. The local error associated to the earlies layer can be computed as follows:

$$\frac{\delta L}{\delta a_t} z_j = \sum_{s \in S} \frac{\delta L}{\delta a_s}\frac{\delta a_s}{\delta a_t} = h'(a_t) \sum_{s \in S} w_{ts} \frac{\delta L}{\delta a_s} z_j \tag{12.9}$$

where $a_s = \sum_{j:j \to s} w_{js} h(a_j)$.

12.4 Training a Neural Network

While training a Feedforward network, we need to do some modeling choices, for instance, we need to choose: the cost function, the form of

output, the activation functions, the architecture (e.g. number of layers), and the **optimizer** (for the training).

Given a set of training samples $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, the main idea of the training is to adjust all the weights of the network Θ such that a cost function is minimized:

$$\min_{\Theta} \sum_i L\left(y_i, f(x_i, \Theta)\right) \tag{12.10}$$

We have to choose our loss function (e.g. L2, cross-entropy), and update the weights of each layer with gradient descent. It can be used the backpropagation of the error signal to compute the gradient efficiently.

12.4.1 Gradient Descent

Recall that the gradient is the vector of partial derivates with respect to all the coordinates of the weights:

$$\nabla_w L = \left[\frac{\delta L}{\delta w_1} \frac{\delta L}{\delta w_2} \cdots \frac{\delta L}{\delta w_N}\right] \tag{12.11}$$

Each partial derivate measures how fast the loss changes in one direction. When the gradient is zero, i.e. all the partials derivates are zero, the loss is not changing in any direction.

Gradient descent finds the set of parameters that makes the loss as small as possible. The change of parameters depends on the gradients of the loss with respect to the network weights. Backpropagation is a method for computing gradient.

In order to define our optimization methods, let's see first the **Gradient Descent Rule**

Algorithm 12: Vanilla Gradient Descent

while *True* **do**
  $weights\_grad \leftarrow$ evaluate_gradient($loss\_fun$, $data$, $weights$);
  $weights \leftarrow weights + step\_size \cdot weigths\_grad$;
end

Batch Gradient Descent

In BGD, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training example and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

A pro of this algorithms is that the gradient estimates are stable, on the other hand it need to compute gradients over the entire training for one update.

Algorithm 13: BatchGradientDescent(k)

Data: Learning rate ϵ_k
Data: Initial parameter Θ
while *stopping criteria is not met* **do**
  Compute gradient estimate over N examples:
  $\hat{g} \leftarrow \frac{1}{N} \nabla_\Theta \sum_i L\left(f(\boldsymbol{x}^{(i)}; \Theta), \boldsymbol{y}^{(i)}\right)$;
  Apply Update: $\Theta \leftarrow \Theta - \epsilon \hat{g}$;
end

Stochastic Gradient Descent

To tackle this problem we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step.

Algorithm 14: StochasticGradientDescent(k)

Data: Learning rate ϵ_k
Data: Initial parameter Θ
while *stopping criteria is not met* **do**
  Sample example $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ from training set;
  Compute gradient estimate:
  $\hat{g} \leftarrow \nabla_\Theta L\left(f(\boldsymbol{x}^{(i)}; \Theta), \boldsymbol{y}^{(i)}\right)$;
  Apply Update: $\Theta \leftarrow \Theta - \epsilon \hat{g}$;
end

A problem with this algorithm is that gradient estimates can be very noisy. One possible solution can be to use larger **mini-batches**. The advantage is that the computation time per update does not depend on the number of training examples N, so it permits computation on extremely large datasets. This solution is often done with a parallel implemenetation. Using GPUs it is common for power of 2 batch sizes to offer better runtime (some kinds of hardware achieve better runtime with specific sizes of arrays).

Momentum

The progress to the minimum with SGD is very slow along a flat direction, it causes jitter along the steep one.

So we can use the SDG with **momentum**, that introduces a new variable $\boldsymbol{v}$, the velocity. The velocity is an exponentially decaying moving average of the negative gradient.

Adaptive Learning Rate Methods

So far we have assigned the same learning rate to all features. If the features vary in importance and frequency, is this a good idea? The learning rate is one of the hyperparameters most difficult to set.

Algorithm 15: StochasticGradientDescent with Momentum

Data: Learning rate ϵ_k
Data: Momentum parameter α
Data: Initial parameter Θ
Data: Learning velocity $\boldsymbol{v}$
while *stopping criteria is not met* **do**
  Sample example $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ from training set;
  Compute gradient estimate:
  $\hat{g} \leftarrow \nabla_\Theta L\left(f(\boldsymbol{x}^{(i)}; \Theta), \boldsymbol{y}^{(i)}\right)$;
  Compute the velocity update:
  $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\hat{g}$;
  Apply Update: $\Theta \leftarrow \Theta - \epsilon\hat{g}$;
end

12.5 Convolutional Neural Networks

Convolutional Neural Network (CNN or ConvNet) is a class of artificial neural network, most commonly applied to analyze visual imagery. Neural networks are extremely powerful in applications that deal with an input space which is locally structured, i.e. spatial or temporal, for example images and language, and that deal with an arbitrary input of features.

The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolutional networks are a specialized type of neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

A convolutional neural network consists of an input layer, hidden layers and an output layer. In any feed-forward neural network, any middle layers are called hidden because their inputs and outputs are masked by the activation function and final convolution. In a convolutional neural network, the hidden layers include layers that perform convolutions.

12.5.1 Convolutional layers

In a CNN, the input is a tensor with a shape: given the number of inputs n, the input height h, the input width w, and the input channels c, the shape s is defined as:

$$s = n \cdot h \cdot w \cdot c \tag{12.12}$$

After passing through a conolutional layer, the image becomes abstracted to a feature map, also called an activation map.

Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus. Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural network can

be used to learn features and classify data, this architecture is generally impractical for larger inputs such as high resolution images. It would require a very high number of neurons, even in a shallow architecture, due to the large input size of images, where each pixel is a relevant input feature. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for each neuron in the second layer. Instead, convolution reduces the number of free parameters, allowing the network to be deeper. For example, regardless of image size, using a 5 x 5 tiling region, each with the same shared weights, requires only 25 learnable parameters. Using regularized weights over fewer parameters avoids the vanishing gradients and exploding gradients problems seen during backpropagation in traditional neural networks.

In practice, CNN learns a hierarchy of features. Each layer of hierarchy extracts features from the output of the previous layer, and then trains all layers jointly.

Furthermore, convolutional neural networks are ideal for data with a grid-like topology (such as images) as spatial relations between separate features are taken into account during convolution and/or pooling.

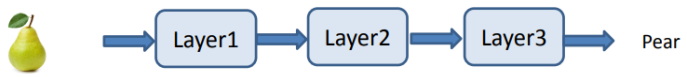


Figure 12.8

Convolutional neural networks are Feedforward neural networks with a specialized connectivity structure, where multiple layers of feature extractors are stacked: low-level layers extract local features, and high-level layers learn global patterns. Typically CNN layers transform the input matrix into an output class prediction. There are a few distinct types of operations:

- Convolution
- Non-linearity
- Pooling

Convolution

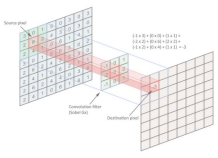


Figure 12.9

The convolution is a mathematical operation on two functions f and g that produces a third function $f \cdot g$ that expresses how the shape of one is modified by the other. It is a general purpose filter operation for images. A kernel matrix is applied to an image, and it works by determining the value of a central pixel by adding the weighted values of all its neighbors together. The output is a new modified (filtered) image, and it can be used

to smooth, sharpen, enhance, etc.

$$S(i,j) = (I \cdot K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{12.13}$$

The convolution layer is the core layer of the CNN, it consists of a set of filters, and each filter covers a spatially small portion of the input data (receptive field) and is convolved across the dimensions of the input data, producing a multi-dimensional feature map.

The intuition is that the network will learn filters that activate when they see some specific type of feature at some spatial position in the input.

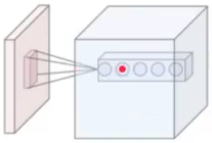


Figure 12.10

The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. A dot product is the element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Because it results in a single value, the operation is often referred to as the "scalar product".

Using a filter smaller than the input is intentional as it allows the same filter (set of weights) to be multiplied by the input array multiple times at different points on the input. Specifically, the filter is applied systematically to each overlapping part or filter-sized patch of the input data, left to right, top to bottom.

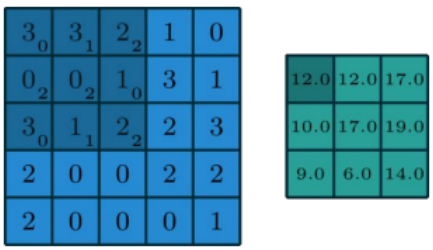


Figure 12.11: An example of an input data with a filter and the output of the total convolution.

Non-linearity

After the convolution, the non-linearity is computed. The result of the convolution as in the feedforward neural network is then passwd to an activation function (e.g. ReLO or Sigmoid)

$$\begin{aligned} y_{i,j} &= f(a_{i,j}) \\ \text{e.g. } f(a) &= [a]_+ \\ f(a) &= sigmoid(a) \end{aligned}$$

We are using non-linearity in order to have a final non-linear representation for our prediction model and, more importantly, since the non-linearity is applied element-wise, it does not affect the receptive field of the neural network.

Spatial Pooling

After the application of the non-linearity, there is a downsampling operation, called **spatial pooling**, and it is used to provide invariance to small traslation of the input.

There are several non-linear functions to implement pooling, where max pooling is the most common. It partitions the input image into a set of rectangles and, for each such sub-region, outputs the maximum.

Intuitively, the exact location of a feature is less important than its rough location relative to other features. This is the idea behind the use of pooling in convolutional neural networks. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, memory footprint and amount of computation in the network, and hence to also control overfitting. This is known as down-sampling. It is common to periodically insert a pooling layer between successive convolutional layers (each one typically followed by an activation function, such as a ReLU layer) in a CNN architecture.

$$x_{i,j} = \max_{|k|<\tau,|l|<\tau} y_{i-k,j-l} \tag{12.14}$$

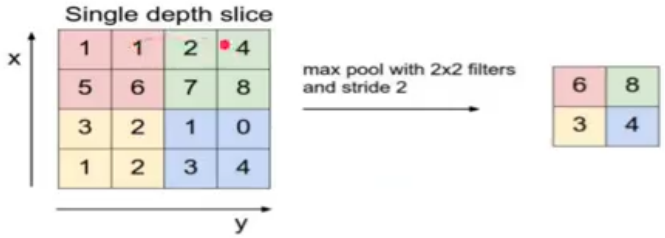


Figure 12.12: Example of max pooling.

12.6 Other Neural Networks

In particular so far we have discussed the Perceptron, the Feedforward neural network, and the Deep Feedforward neural network, where we have multiple hidden layers.

So far, we focused mainly on prediction problems with fixed-size inputs and outputs. We discussed the flexibility of CNN to address a wide range of task, but what if the input and/or output is a variable-length

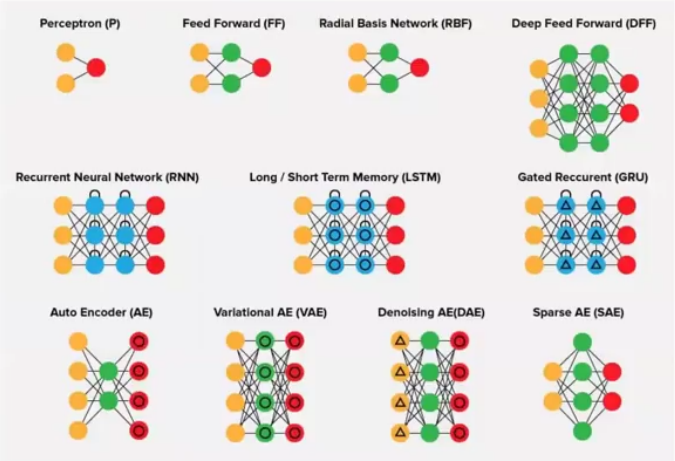


Figure 12.13: Many models for different needs.

sequence? There are many applications where we need this, for example document classification, sentiment analysis, or image captioning.

12.6.1 Recurrent Neural Networks

For this problem, different type of neural networks are required. Let's consider an example, a video frame prediction, given a video, the model should predict the category of each frame of the video — this example could be useful for segmenting a movie in different parts. Until now, we have seen a **Single2Single** type of neural network, the feedforward neural network, where we have given a single object as input and we want to predict a single object as output. But for this problem we need to implement a **Multiple2Multiple** network, the **Recurrent Network**, that has been designed in problems in which we have multiple object as input and we want to predict multiple objects as output. Note that there is another type of neural network, the **Single2Multiple**, where I have for example a single image and I want to predict multiple captions of the image.

For the problems above, Feedforward neural networks and Convolutional neural networks cannot be applied. The choice in this case is to apply another type of neural network, where, given an input x_t, the neural network passes the input to an input layer that learns an hidden representation h_t that could be eventually used by a classifier to perform some output prediction y_t.

Notice that the particularity of the Recurrent Neural Network is the **recurrence formula**, that is a function of certain parameters of the network that relates the input at time t and the latent representation of the previous state with the latent representation of the current state:

$$h_t = f_W(x_t, h_{t-1}) \tag{12.15}$$

These recurrents operations take into account the concept of time/sequence processing, and can be better understood by unrolling the formula with the scheme in Figure 12.14.

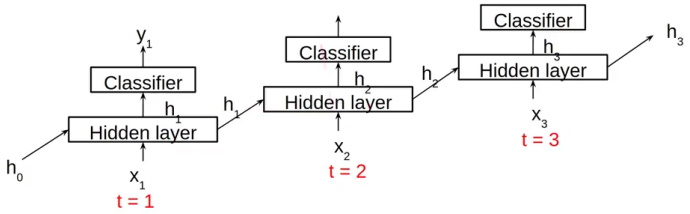


Figure 12.14: Recurrent neural network.

12.6.2 Autoencoders

Autoencoders (AE) are somwhat similar to FFNNs as AEs are more like a different use of FFNNs than a fundamentally different architecture. The basic idea behind autoencoders is to encode information (as in compress, not encrypt) automatically, hence the name. The entire network always resembles an hourglass like shape, with smaller hidden layers than the input and output layers. AEs are also always symmetrica around the middle layer(s). The smalles layers are almost always in the middle, the place where the information is most compressed. Everything up to the middle is called the encoding part, everything after the middle the deconding and the middle is called the code. One can train them using backpropagation by feeding input and setting the error to be the difference between the input and what came out. AEs can be built symmetrically when it comes to weights as well, so the encoding weights are the same as the decoding weights.

Autoencoders can be considered as an unsupervised approach for learning a lower-dimensional feature representtion from unlabeled data.

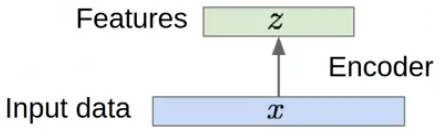


Figure 12.15: Autoencoders scheme example.

For instance, if the input is a bunch of image, in the autoencoder there is a network — the encoder network — that maps the original input data into low-dimensional latent representations.

Encoders can be realized in several ways, for instance they can be realized by linear layers + nonlinearity (e.g. sigmoid), can be realized by a deep fashion by concatenating multiple fully-connected layers, or it can be realized in a convolutional fashion by having some convolutional layers and ReLU activation function one after the other.

Figure 12.16

Independently of the encoder, the main idea is that the representation of the output z should be smaller than the input x, so the encoder should perform dimensionality reduction.

How to learn this feature representation? The idea is to train the autoencoder so that features can be used to reconstruct original data — autoencoding: encoding itself.

Once we decided the architecture of the encoder and the consequent architecture of the decoder, we need a loss function, and in the case of the autoencoders the loss function is a reconstruction function that minimizes the difference between the original data and the reconstructed version of the input data:

$$||x - \hat{x}||^2 \tag{12.16}$$

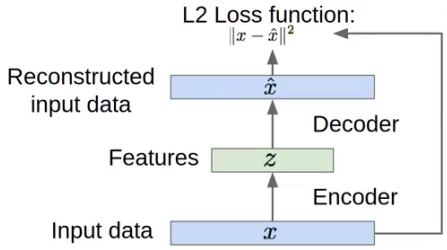


Figure 12.17: Autoencoders with loss function.

Autoencoders are aimed to learn the latent representation z, so the most common use of autoencoders is to use the common feature z in order to initialize a feature model. This means that once we have trained the autoencoder, we throw away the decoder, and we only take the latent representation z and we append the loss funcion $\hat{y}$ and the the encoder now can be used to have a better initialization of a supervised model.

Part III

Unsupervised Learning

13

Unsupervised Learning

Unsupervised learning is a type of machine learning in which the algorithm is not provided with any pre-assigned labels or scores for the training data. As a resut, unsupervised learning algorithms must first self-discover any naturally occurring patterns in that training data set.

In unsupervised learning we observe data that are sampled from an unknown distribution $p_{\text{data}} \in \Delta(\mathcal{X})$, but we lack observations about the target variables (e.g. we don't have access to class labels). It is still possible to detect some pattern to grab some information from this data, we can compensate for the lack of this features by designing an appropriate objective function.

13.1 Tasks in depth

In particular the objective function we have designed will lead us to solve three families of problems:

- Dimensionality reduction;
- Clustering;
- Density estimation.

13.1.1 Dimensionality Reduction

The **dimensionality reduction** is the transformation of data from a high-dimensional space into a low-dimensional space so that the low-dimensional space representation retains some meaningful properties of the original data, ideally close to its intrinsic dimension.

The task of the dimensionality reduction is to find a function $f \in \mathcal{Y}^{\mathcal{X}}$ mapping each high-dimensional input $x \in \mathcal{X}$ to a lower dimensional,

embedding $f(x) \in \mathcal{Y}$, where $dim(\mathcal{Y}) \ll dim(\mathcal{X})$.

The purposes of dimensionality reduction are to compress the input data by reducing the feature dimensionality, while preserving as much information as possible. The way information loss is measured yields different algorithms. This reduces time of subsequent data elaboration and/or storage. Furthermore, it enables better visualization of data and reduces the curse of dimensionality.

Figure 13.1: Dimensionality reduction.

13.1.2 Clustering

Clustering is the task of grouping a set of objects in the same group (called a cluster) are more similar to each other than to those in other groups.

The appropriate clustering algorithm and parameter settings (including parameters such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

The task of clustering is to find a function $f \in \mathbb{N}^{\mathcal{X}}$ that assigns each input $x \in \mathcal{X}$ a cluster index $f(x) \in \mathcal{N}$. All points mapped to the same index form a cluster.

Clustering is used because it allows to analyze data by grouping together data points that exhibit some regular pattern or similarity under some predefined criterion. It can be used to compress data by reducing the number of data points as opposed to reducing the feature dimensionality.

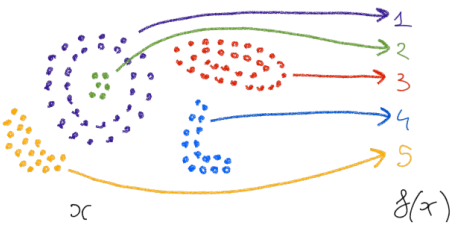


Figure 13.2: Clustering.

Applications example of clustering are: cluster users by reference (e.g. based on movie ratings), group genes families from gene sequences, find communities in social networks, etc.

13.1.3 Density Estimation

The **density estimation** is the construction of an estimate, based on observed ata, of an unobservable underlying probability density function.

A very natural use of density estimates is in the informal investigation of the properties of a given set of data. Density estimates can give valuable indication of such features as skewness and multimodality in the data. In some cases they will yield conclusions that may then be regarded as self-evidently true, while in others all they will do is to point the way to further analysis and/or data collection.

The task of density estimation is to find a probability distribution $f \in \Delta(\mathcal{X})$ that fits the data $x \in \mathcal{X}$.

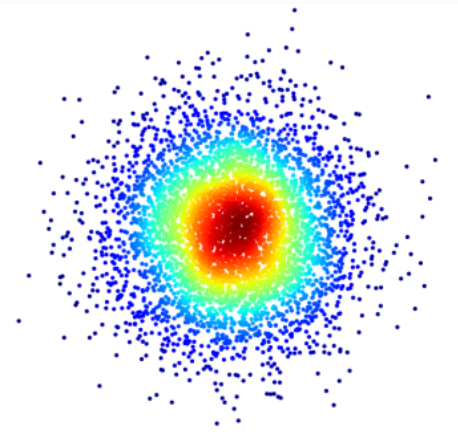


Figure 13.3: Density estimation.

Density estimation allows to get an explicit etimate of the unknown probability distribution that generated the training data. It enables the generation of new data by sampling from the estimated distribution, and it enables the detection of anomalies/novelties in terms of data points that a exhibit low probabilities according to the estimated distribution.

13.2 Principal Component Analysis

The **principal components** of a collection of points in a real coordinate space are a sequence of p unit vectors, where the i-th vector is the direction of a line that best fits the data while being orthogonal to the first $i-1$ vectors. Here, a best-fitting line is defined as one that minimizes the average squared distance from the points to the line. These directions

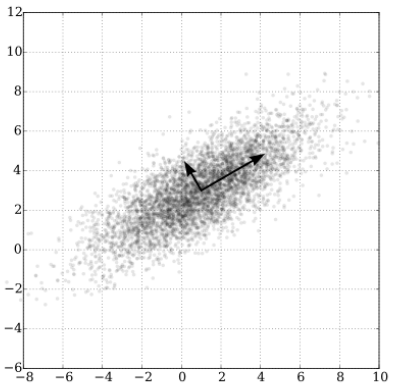


Figure 13.4: PCA of a multivariate Gaussian distribution centered at $(1, 3)$.

constitute an orthonormal basis in which different individual dimensions of the data are linearly uncorrelated.

Principal Component Analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal compoents and ignoring the rest — namely, drop dimensions of least variance.

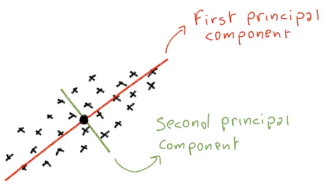


Figure 13.5

In a nutshell, we have a set of data points that are represented by black crosses symbols in Figure **??**. We want to find the direction the direction that explains the maximum variance of the data, in this case is the direction indicated by the red line — that is our principal components. Our second principal component is orthogonal to the first principal component.

When we found our components, now the directions lead us to a new coordinate system. The way PCA allows us to do dimensionality reduction is by dropping the dimennsions of least variance.

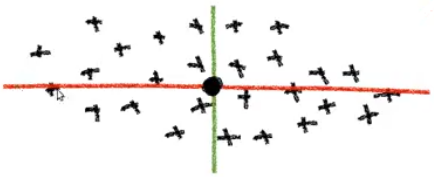


Figure 13.6

13.2.1 Variance Along Unit Direction

We are interested in computing the variance with the constraint that our parameter vector w has norm 1, this means: $\boldsymbol{w}^T\boldsymbol{w} = ||\boldsymbol{w}|| = 1$.

So what we do is to take each data point $\boldsymbol{x}_i$, and project $\boldsymbol{x}_i$ to another line that corresponds to the direction of our weight vector w. To do that we use an auxiliary point $\boldsymbol{c}$, and we compute the distance between $\boldsymbol{x}_i$ and $\boldsymbol{c}$, then $t_i = (\boldsymbol{x}_i - \boldsymbol{c})^T\boldsymbol{w}$.

Once we defined our reference point $\boldsymbol{c}$ we are interested in maximizing the variance of the data points. To do that we first have to compute expected value of t considering all the t_i of all the data points $\boldsymbol{x}_i$ in our dataset.

$$\mathbb{E}[t] = \frac{1}{n}\sum_{i=1}^{n} t_i = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x}_i - \boldsymbol{c})^T\boldsymbol{w} = \bar{\boldsymbol{x}}^T\boldsymbol{w} - \boldsymbol{c}^T\boldsymbol{w} \tag{13.1}$$

Where $\bar{\boldsymbol{x}} = \frac{1}{n}\sum_i x_i$, now we can compute the variance:

$$\mathbb{V}ar[t] = \frac{1}{n}\sum_{i=1}^{n}(t_i - \mathbb{E}[t])^2 = \frac{1}{n}\sum_{i=1}^{n}\left[(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T\boldsymbol{w}\right]^2 \tag{13.2}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(\bar{\boldsymbol{x}}_i^T\boldsymbol{w}\right)^2 = \boldsymbol{w}^T\left[\frac{1}{n}\bar{X}\bar{X}^T\right]\boldsymbol{w} = \boldsymbol{w}^T C\boldsymbol{w} \tag{13.3}$$

where $t_i = (\boldsymbol{x}_i - \boldsymbol{c})^T\boldsymbol{w}$, $\bar{X} = [\bar{\boldsymbol{x}}_1, \bar{\boldsymbol{x}}_2, ..., \bar{\boldsymbol{x}}_n]$, and $C = \left[\frac{1}{n}\bar{X}\bar{X}^T\right]$ is the covariance matrix.

13.2.2 Eigenvalue Decomposition

Let $A \in \mathbb{R}^{m\times m}$, symmetric. There exists $U = [\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_m] \in \mathbb{R}^{m\times m} and \boldsymbol{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_m)^T \in \mathbb{R}^m$ such that:

$$A = U\Lambda U^T = \sum_{j=1}^{m}\lambda_j\boldsymbol{u}_j\boldsymbol{u}_j^T \qquad \text{and} \qquad U^TU = UU^T = I \tag{13.4}$$

where U is the matrix of the eigenvectors, Λ is a square matrix with values all 0 except of the principal diagonal, that is composed by $\lambda_1, \lambda_2, ..., \lambda_m$, I is the identity matrix, $\boldsymbol{u}_j$ is an eigenvector and λ_j is the corresponding eigenvalue.

Note that we assume descending ordering $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$.

First Principal Component

$$\boldsymbol{w}_1 \in \arg\max\left\{\boldsymbol{w}^T C\boldsymbol{w} : \boldsymbol{w}^T\boldsymbol{w} = 1\right\} \tag{13.5}$$

The largest eigenvalue of C is the variance along the first principal component, and the first principal component $\boldsymbol{w}_1$ is the corresponding eigenvector.

Proof By eigenvale decomposition $C = \sum_j \lambda_j \boldsymbol{u}_j \boldsymbol{u}_j^T$ and assume eigenvalue sorted in descending order, i.e. $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m$. Then, $\boldsymbol{w}_1^T C \boldsymbol{w}_1 = \sum_j \lambda_j (\boldsymbol{w}_1^T \boldsymbol{u}_j)^2 \leq \lambda_1$ because

$$\sum_j (\boldsymbol{w}_1^T \boldsymbol{u}_j)^2 = \boldsymbol{w}_1^T \sum_j \boldsymbol{u}_j \boldsymbol{u}_j^T \boldsymbol{w}_1 = \boldsymbol{w}_1^T U U^T \boldsymbol{w}_1 = \boldsymbol{w}_1^T \boldsymbol{w}_1 = 1$$

It follows that $\lambda_1 \geq \boldsymbol{w}_1^T C \boldsymbol{w}_1 \geq \boldsymbol{u}_1^T C \boldsymbol{u}_1 = \lambda_1$ which implies that

$$\boldsymbol{w}_1^T C \boldsymbol{w}_1 = \boldsymbol{u}_1^T C \boldsymbol{u}_1$$

Therefore $\boldsymbol{u}_1$ i.e. the eigenvector corresponding to the largest eigenvalue λ_1 of C, is a first principal component and λ_1 the variance along it.

Second Principal Component

$$\boldsymbol{w}_2 \in \arg\max \left\{ \boldsymbol{w}^T C \boldsymbol{w} : \boldsymbol{w}^T \boldsymbol{w} = 1, \boldsymbol{w} \perp \boldsymbol{w}_1 \right\} \tag{13.6}$$

The largest eigenvalue of C is the variance along the second principal component, and the second principal component $\boldsymbol{w}_2$ is the corresponding eigenvector.

Proof By eigenvale decomposition $C = \sum_j \lambda_j \boldsymbol{u}_j \boldsymbol{u}_j^T$ and assume eigenvalue sorted in descending order, i.e. $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m$. Then, $\boldsymbol{w}_2^T C \boldsymbol{w}_2 = \sum_{j=1}^m \lambda_j (\boldsymbol{w}_2^T \boldsymbol{u}_j)^2 = \lambda_1 \boldsymbol{w}_2^T \boldsymbol{u}_1 + \sum_{j=2}^m \lambda_j (\boldsymbol{w}_2^T \boldsymbol{u}_j)^2 \leq \lambda_2$ because

$$\sum_j (\boldsymbol{w}_2^T \boldsymbol{u}_j)^2 = \boldsymbol{w}_2^T \sum_j \boldsymbol{u}_j \boldsymbol{u}_j^T \boldsymbol{w}_2 = \boldsymbol{w}_2^T U U^T \boldsymbol{w}_2 = \boldsymbol{w}_2^T \boldsymbol{w}_2 = 1$$

It follows that $\lambda_2 \geq \boldsymbol{w}_2^T C \boldsymbol{w}_2 \geq \boldsymbol{u}_2^T C \boldsymbol{u}_2 = \lambda_2$ which implies that

$$\boldsymbol{w}_2^T C \boldsymbol{w}_2 = \boldsymbol{u}_2^T C \boldsymbol{u}_2$$

Therefore $\boldsymbol{u}_2$ i.e. the eigenvector corresponding to the second largest eigenvalue λ_2 of C, is a second principal component and λ_2 the variance along it.

i-th Principal Component

$$\boldsymbol{w}_i \in \arg\max \left\{ \boldsymbol{w}^T C \boldsymbol{w} : \boldsymbol{w}^T \boldsymbol{w} = 1, \boldsymbol{w} \perp \boldsymbol{w}_j \text{ for } 1 \leq j < i \right\} \tag{13.7}$$

The i-th largest eigenvalue of C is the variance along the i-th principal component. The i-th principal component $\boldsymbol{w}_i$ is the corresponding eigenvector.

The proof is similar to the one we already saw.

13.2.3 PCA using Eigenvalue Decomosition

To summarize, we can provide the algorithm that allows us to compute the principal components through eigenvalue decomposition.

Given data points $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n]$, we apply some centering $\bar{X} = X - \frac{1}{n} X 1_n 1_n^T$, where 1_n is a vector of ones, so we take our data points vector and subtract the centres (mean) of the data points. Ones we have the center, we compute the covariance matrix $C = \frac{1}{n}\bar{X}\bar{X}^T$, and then we can compute the eigenvalue decomposition U, $\lambda = eig(C)$. Our principal components will be $W = U = [\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_m]$, with variances $\lambda = (\lambda_1, \lambda_2, ..., \lambda_m)$.

13.2.4 PCA Using Singular Value Decomposition (SVD)

Singular Value Decomposition

Let $A \in \mathbb{R}^{m \times n}$. There exist $U \in \mathbb{R}^{m \times k}, s \in \mathbb{R}^k$ with $s_1 \geq s_2 \geq ... \geq s_k > 0$ and $V \in \mathbb{R}^{n \times k}$ such that:

$$A = USV^T \qquad \text{and} \qquad U^T U ? V^T V = I \tag{13.8}$$

PCA with SVD

In order to use PCA, we have to compute SVD of $\bar{X} : U, \boldsymbol{s}, V = SVD(\bar{X})$. The principal components are $U = [\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_m]$, and the variances are $\left(\frac{s_1^2}{n}, \frac{s_2^2}{n}, ..., \frac{s_k^2}{n}\right)$. This because $\bar{X} = USV^T$, and $C = \frac{1}{n}\bar{X}\bar{X}^T = \frac{1}{n}USV^T VSU^T = U\frac{s^2}{n}U^T$, where $\frac{s^2}{n}$ corresponds to Λ in the case of the eigenvalue decomposition.

13.2.5 Dimensionality Reduction Using PCA

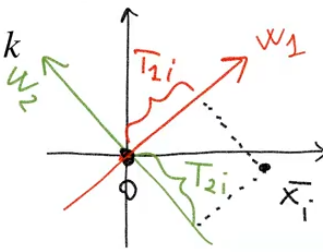


Figure 13.7

Now we discuss how it is possible to reduce dimensionality using PCA. Let's consider $\hat{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_k]$ hold the first k principal components derived from data points $\bar{X} = \left[\bar{\boldsymbol{X}}_1, \bar{\boldsymbol{X}}_2, ..., \bar{\boldsymbol{X}}_n\right]$.

If we compute

$$T = \hat{W}^T \bar{X} \in \mathbb{R}^{k \times n} \tag{13.9}$$

it will induce a change of the coordinate system with respect to the k principal components in $\hat{W}^T$, and this will reduce the dimension of the features to k if we have chosen the first k principal components.

We can use eigenvalue decomposition or SVD algorithms that avoid computing the full decomosition.

13.2.6 Alternative Interpretation

There is also an alternative interpretation of the first principal component analysis: principal components can be interpreted as the line in

the space with minimal squared distance from the data points. A similar interpretation can be given to the other principal components.

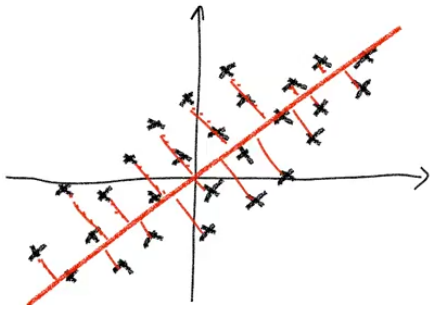


Figure 13.8: Alternative interpretation of the principal components.

Note that the **scale of the feature dimensions** matter when we are using PCA, so if the features are expressed in different units, it is recommended to normalize and scale them to have unit standard deviation.

How many principapl components should we choose? The number depends on the goal and on the application. In general we have no way to know a-priori how many components do we need, unless we are in the context of supervised classification methods, this means that first we are applying PCA and reducing the dimensions of our data points and then we can apply some classification method.

Nonetheless we can compute the cumulative proportion of explained variance, which for the first k principal components is given by:

$$\text{Eigenvalue decomposition: } \frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{m} C_{jj}} \qquad \text{SVD: } \frac{\sum_{j=1}^{k} s_j^2}{\sum_{ij} \bar{X}_{ji}^2} \tag{13.10}$$

this allows to estimate the amount of information loss

13.2.7 Kernel PCA

PCA reduces the dimensionality via a linear transformation. By using the kernel trick one can apply PCA in a higher dimensional space, yielding a non-linear transformation in the original space.

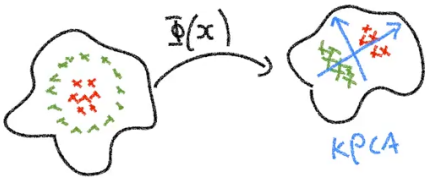


Figure 13.9: Kernel PCA.

13.3 k-Means Clustering

Given data points $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n] \in \mathbb{R}^{d \times n}$, and k a fixed number of clusters. Our goal is to fina a partition of data points into k sets $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_k$ minimizing the variation $V(\mathcal{C}_j)$ within each set $V(\mathcal{C}_j)$:

$$\min_{\mathcal{C}_1,...,\mathcal{C}_k} \sum_{j=1}^{k} V(\mathcal{C}_j) \tag{13.11}$$

The variation is typically given by the euclidean distance $V(\mathcal{C}_j) = \sum_{i \in \mathcal{C}_j} ||\boldsymbol{x}_i - \boldsymbol{\mu}_j||^2$ where $\boldsymbol{\mu}_j = \frac{1}{|\mathcal{C}_j|} \sum_{i \in \mathcal{C}_j} \boldsymbol{x}_i$ is the centroid of $\mathcal{C}_j$.

13.3.1 Optimization Algorithm

The algorithm that is used to minimize the objective function is an iterative algorithm that is very simple. We first initialize the assignment and we assign randomly each data point to one cluster, in particular we initialize the centroid of each cluster and then we will iterate the following steps while clusters change:

1. Compute cluster centroids $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_k$;
2. Assign each data point to the closest centroid forming a new cluster, i.e.

$$\mathcal{C}_j = \left\{ i \in \{1, ..., n\} : j = \arg\min_{\mathcal{C}} ||\boldsymbol{x}_i - \boldsymbol{\mu}_{\mathcal{C}}|| \right\} \tag{13.12}$$

13.3.2 Properties

The algorithm is guaranteed to converge, because it strictly improves the objective if there is at least a cluster change and the set of possible partitions is finite. It is not guaranteed to find the global minimum but a local one. the problem is NP-hard even on the plane ($d = 2$).

Furthermore, it is sensitive to the scale o features. Features normalization is required in case of features with different scales.

14

Clustering

We continue to tal about Unsupervised Learning, and in particular about **Clustering**. We have seen how the k-means algorithm as one of the simplest algorithm for finding some structure on our training data, in particular to discover groups of similar samples in our set.

When we have our row data we need to extract the features from them, now with the features of the data, we need to group them into clusters. Note that there is **no supervision**, this means that we are only given data and want to find the groupings of this data.

Clustering is the process of grouping a set of objects into classes of similar objects.

14.1 k-Means

One of the questions that we want to discuss is *"What algorithm should we choose for clustering?"*, we have already seen the k-means algorithm, but let's discuss its issues and its properties.

The k-means is the most popular clustering algorithm with a simple implementation. It starts by initializing randomly the centers, and then it iterate the clustering operation: it clusters each example to the closest center and recalculate centers as the mean of the points in a cluster.

14.1.1 Issues

k-means is a very powerful algorithm, however it has some issues, for instance is the calculation of the distance that is done with the euclidean

distance

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{14.1}$$

However for some applications the euclidean distance could be not the best choice. Let for instance consider the clustering of documents: we have one feature for each word, the value is the number of times that word occurs, and documents are points or vectors in the space.

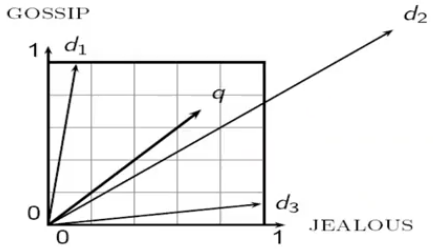


Figure 14.1: Document clustering representation.

With respect to Figure 14.1, which document is closes to q using the euclidean distance? With the euclidean distance it will be closer to d_1 and d_3, but in practice it should be closer to d_2, because the distributions of the words "GOSSIP" and "JEALOUS" are very similar.

Distance

In this case it could be more meaningful to have a different distance, in particular measure the **cosine similarity** and measure the distance as the difference of the cosine similarity.

The cosine similarity between vectors is correlated with the angular distance between two vectors, and it can be computed in this way:

$$sim(x, y) = \frac{x \cdot y}{|x||y|} = \frac{x}{|x|} \cdot \frac{y}{|y|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}} \tag{14.2}$$

The cosine similarity ranges from 0 to 1, it is good for text data and many other real-world data sets, and it is computationally friendly since we only need to consider features that have non-zero values for both examples.

The cosine distance can be computed as follows:

$$d(x, y) = 1 - sim(x, y) \tag{14.3}$$

14.1.2 Properties

We already said that the algorithm is guaranteed to converge, because it strictly improves the objective if there is at least a cluster change and

the set of possible partitions is finite. It is not guaranteed to find the global minimum (optimum) but a local one.

Regarding to the first property, each step of k-means move towards reducing the loss function (or at least not increasing it). In the first step (assignment), any other assignment would end up in a larger loss, while in step 2 (centroid computation), the mean of a set of values minimizes the squared error.

Regarding to the second property, it is important to remark that the k-means loss function is generally not convex and for most problems it has many minima, we are guaranteed to find one of them, but not necessarily the global minimum — the minimum we find depends on the initialization.

Centroids selection plays an important role with the result, results can vary drastically based on random seed selection, ome seeds can result in poor convergence rate or convergence to sub-optimal clustering. Some common heuristics are:

- Random points (not examples) in the space;
- Randomly pick examples;
- Points least similar to any existing center (furthest centers heuristic);
- Try out multiple starting points;
- Initialize with the results of another clustering method.

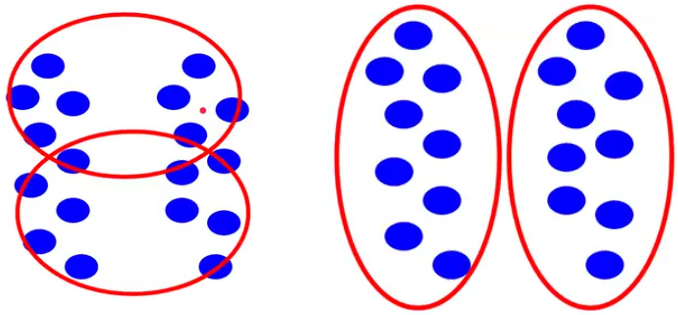
14.2 Issues for Clustering

The second question we are interested in discussing is *"What are some of the issues for clustering?"*. In this section we are going to examine some issues that are beyond the k-means algorithm but still related to the clustering algorithms.

The first issue that we have for clustering is how to represent our data points, we will see that the feature extraction algorithm plays a crucial role. A second thing that we already discussed is that the way we choose to measure the distance between examples in the training set has a strong impact on the final performance of the clustering algorithm. Then, another issue is if the problem necessitates flat clustering or hierarchical clustering algorithms. A final factor that we need to discuss is how to define the number of cluster we are going to consider in our dataset, number of cluster can be fixed a priori or data driven.

14.2.1 Types of Clutering Algorithms

Clustering algorithms can be devided in**Flat Algorithms** and **Hierarchical Algorithms**. Flat algorithms usually start with a random



(a) Spherical clusters

(b) Elliptic clusters

Figure 14.2: Spherical clusters vs. Elliptic clusters.

partial partitioning and refines it iteratively (e.g. k-means and model based clustering). Hierarchical algorithms are bottom-up agglomerative (progressively aggregate data) or top-down divisive (split the dataset).

Regarding flat algorithms, flat clustering algorithms can be implemented in two ways: **Hard clustering**, where each example belongs to exactly one cluster (e.g. k-means); and **Soft clustering**, where an example can belong to more than one cluster (probabilistic). Soft clustering makes more sense for applications like creating browsable hierarchies, suppose for example you want to put a pair of sneakers in two cluster (sports apparel and shoes), then soft clustering is required.

14.3 EM Clustering

The k-means algorithm we have seen assumes spherical clusters, this means that a sub-optimum solution can be like the one in Figure **??**, that we can see is not the optimal solution.

The improved version of k-means it's called **EM Clustering** and assumes that data came from a mixture of Gaussians (elliptical data), assigns data to cluster with a certain probability (soft clustering). In Figure **??** we can see that it provides a better clustering solution for our dataset.

The idea of EM clustering is to follow an iterative scheme as k-means, it iterates between assigning points and recalculating cluster centers. The two main differences between k-means and EM clustering are:

- We assume elliptical clusters instead of spherical clusters.
- It is a soft clustering algorithm.

14.3.1 Soft Clustering in EM Clustering

EM clustering is a soft clustering algorithm, this means that is a probabilistic algorithm. In order to understand more the meaning of

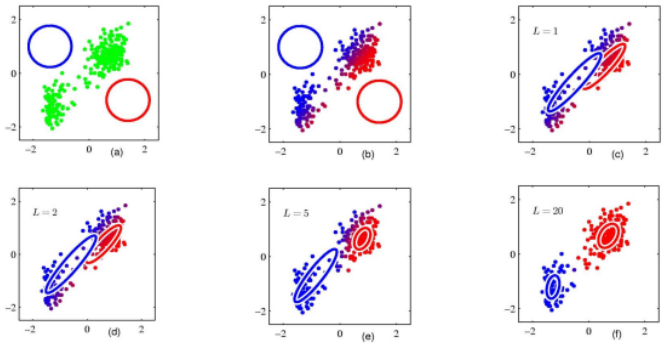


Figure 14.3: EM Clustering Algorithm execution example.

probabilistic, let's take as an example the dataset in Figure **??**: given a point ***d*** in the right-side ellipse, ***d*** has a certain probability to be $right$ (e.g. $p(right) = 0.8$), and another probability to be $left$ (e.g. $p(left) = 0.2$); this means that it is assigned to a cluster based on the highest probability to be of one of the clusters, in this case it is the left cluster.

EM Algorithm

EM algorithm operate in this way:

Algorithm 16: EM Clustering

```
Initialize clusters;
while clusters does not change do
    Calculate p(Θ_c|x) the probability of each point belonging to
      each cluster;
    Recalculate the new cluster parameters Θ_c, the maximum
      likelihood cluster centers given the current soft clustering;
end
```

14.3.2 Mixture of Gaussians

We have seen the intuition of the EM clustering, but we have not seen how do we define a Gaussian (i.e. ellipse), this requires a bit of math.

In the case of one dimensional distributions, the gaussian is parametrized by the mean and the standard deviation/variance:

$$f(x; \sigma, \Theta) = \frac{1}{\sigma\sqrt{2\pi}} exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \tag{14.4}$$

The equation can be extended in an m-dimensional space, where the gaussians are calculated as follows:

$$N[x; \mu, \Sigma] = \frac{1}{\sqrt{(2\pi)^d}\sqrt{det(\Sigma)}} exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \tag{14.5}$$

where Σ is the covariance matrix, and we have one gaussian for each cluster. Given the Algorithm **??**, the second step inside the iteration, the cluster parameters will be the mean vector and the covariance matrix that better fit the assigned clusters. In m-dimensional space we learn the means of each cluster (i.e. the center) and the covariance matrix (i.e. how spread out it is in any given direction).

14.3.3 Expectation Maximization

EM stands for Expectation Maximization:

Expectation: Given the current model, figure out the expected probabilities of the data points to each cluster $p(\Theta_c|x)$. *"What is the probability of each point belonging to each cluster?"*.

Maximization: Given the probabilistic assignment of all the points, estimate a new model Θ_c. *"Maximum likelihood estimation"*.

EM is similar to k-means, each iteration increases the likelihood of the data and is guaranteed to converge (though to a local minimum). EM is a general purpose approach for training a model when you don't have labels, but it is not just for clustering (while k-means is just for clustering).

14.4 Other Clustering Algorithms

k-means and EM clustering are by far the most popular clustering algorithms. However, they cannot handle all clustering tasks. As an example, let's consider the example in which we have non-gaussian data — i.e. in Figure 14.4 we have two circles and two clusters with k-means, but it is obviously an incorrect solution.

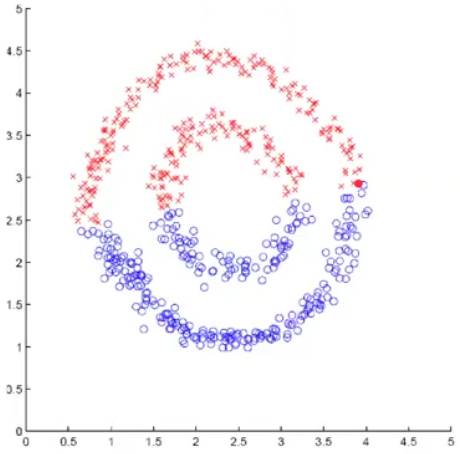


Figure 14.4: Two circles and two clusters (k-means).

14.4.1 Spectral Clustering

The idea of spectral clustering methods is represented in Figure 14.5, where we form a matrix that indicate the similarity of each data point with respect to each other.

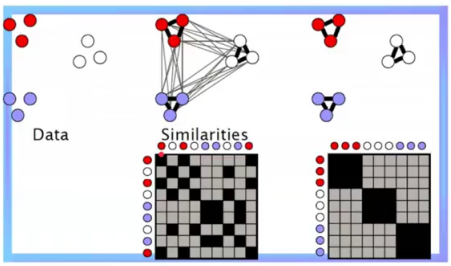


Figure 14.5

This similarity matrix can be also represented in term of graphs, where each edge indicates the similarity between two points. The black edges in the adjacent matrix, that are harder edges in the graph, are called high similar points. If we cut the lowest similar edges, we have then our high-similar clusters.

14.4.2 Hierarchical Clustering

The last family of clustering algorithm we are going to discuss is the family of **Hierarchical Clustering Algorithm**, that are perceived for flat solution.

Hierarchical clustering algorithms produce a set of nested clusters organized as a hierarchical tree. The tree is called a **dendogram**.

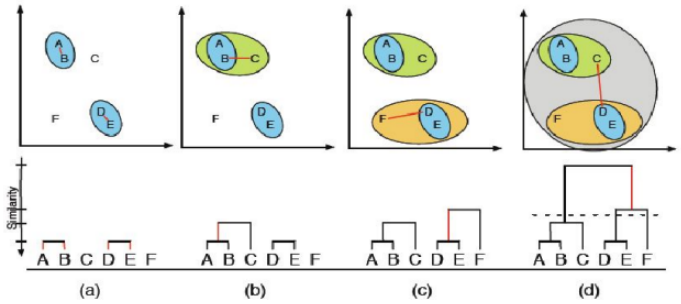


Figure 14.6: Hierarchical Clustering Dendrogram example.

15

Deep Generative Models

Another type of unsupervised learning, in particular in the field of Deep Learning, are **Deep Generative Models (DGMs)**. The first important thing that we need to point, is the difference between the terms **generative** and **discriminative**. Generative models are statistical models of the data distribution p_X or p_{XY} depending on the availability of target data. Discriminative models are statistical models of the conditional distribution $p_{X|Y}$ of the target given the input.

A discriminative model can be constructed from a generative model via the Bayes rule, but not vice versa!

$$p_{Y|X}(y, x) = \frac{p_{XY}(x, y)}{\sum_{y'} p_{XY}(x, y')} \tag{15.1}$$

15.1 Density Estimation

We are interested in the **Density Estimation**, in the case of the density estimation we are in the unsupervised learning setting, and we want to find the probability distribution $f \in \Delta(\mathcal{Z})$ that fits the data $z \in \mathcal{Z}$, where z is sampled from an unknown data distribution $p_{\text{data}} \in \Delta(\mathcal{Z})$. Since in the unsupervised learning setting we don't have annotations, we have that $\mathcal{Z} = \mathcal{X}$ (while in supervised learning $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$.

The previous density estimation is called **explicit**, on the other hand, the **implicit** density estimation aims to find a function $f \in \mathcal{Z}^{\Omega}$ that generates data $f(\omega) \in \mathcal{Z}$ from an input ω sampled from some predefined distribution $p_\omega \in \Delta(\Omega)$ in a way that the distribution of generated samples fits the (unknown) data distribution $p_{\text{data}} \in \Delta(\mathcal{Z})$.

The objective of the density estimation are: to define an hypothesis space $\mathcal{H} \subset \Delta(\mathcal{Z})$ of models that can represent probability distributions

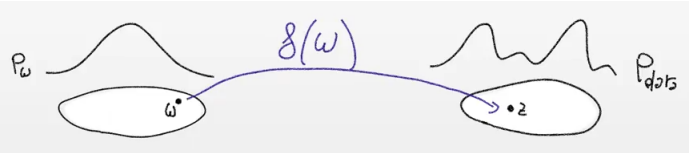


Figure 15.1: Implicit Desity Estimation.

(implicitly or explicitly); to define a divergence measure $d \in \mathbb{R}^{\Delta(\mathcal{Z})\times\Delta(\mathcal{Z})}$ between probability distributions in $\Delta(\mathcal{Z})$ (e.g. Kullback-Leibler divergence); find an hypothesis $q^* \in \mathcal{H}$ that best fits the data distributed according to p_{data}. The best fit is measured using the divergence d, i.e.:

$$q^* \in \arg\min_{q\in\mathcal{H}} d(p_{\text{data}}, q) \tag{15.2}$$

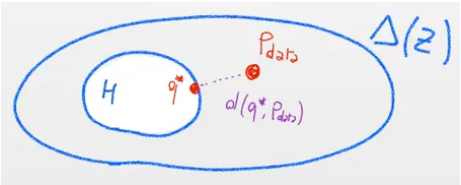


Figure 15.2

Examples of algorithms that use density estimation are:

- Variational AutoEncoder (VAE), that use explicit density, and
- Generative Adversarial Networks (GAN), that use implicit density.

These two algorithm both solve the same problems, but with different assumptions.

15.2 Variational AutoEncoder (VAE)

The idea of Variational AutoEncoder is to revisit autoencoder in a probabilistic fashion.

15.2.1 AutoEncoder

An autoencoder is a way of compressing high-dimensional data into a lower dimensional representation. An **encoder** maps input data x to a compressed representation ω. The compressed representation preserves meaningful factors of variations in the data. E.g. PCA with dimensions dropped can be seen as a linear encoder preserving as much variance in the data as possible.

In this case we assume more complex transformations from the orig-

inal space $\mathcal{X}$ to the latent space Ω that can be obtained with linear transformation and non-linearity.

An encoder is trained by leveraging a **decoder** mapping the representation ω back to the input domain yielding $\hat{x}$ (reconstruction). So, the decoder *autoencodes* its input. the objective is to minimize the divergence between input x and its reconstruction $\hat{x}$.

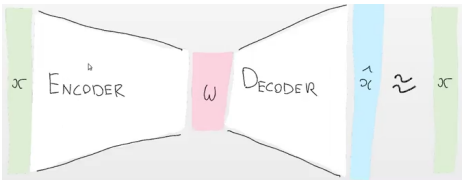


Figure 15.3: Encoding and Decoding functions representation.

After training the decoder is not needed anymore, since it was only functional to estimate the encoder. The encoder can be used to initialize or precompute features for supervised models.

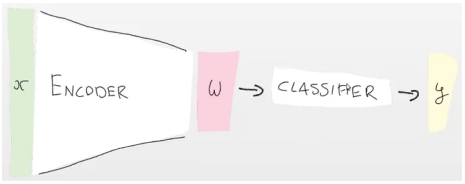


Figure 15.4: AutoEncoder usage.

15.2.2 AutoEncoder in Generative Models

AutoEncoders are generally used to train an encoder network that can be used for supervised learning purpose. But we are now interested on the usage of AutoEncoders for generate new data, infact the decoder could be used to generate new data, so AutoEncoders can be thought as a generative model, but it will not generate data according to the data distribution p_{data}!

What we would like to have is: given a latent space Ω with a prior distribution $p(\omega)$, and an input space $\mathcal{X}$ with a data distribution $p_{\text{data}}(x)$. We want a decoder that produce a mapping from the latent space to the input space where the regions of low probability of the latent input space correspond to the region of the low probability of the latent space, and the same thing with the region of high probability.

Our problem can be formalized as follows:

$$q_\theta(x) = \mathbb{E}_{\omega \sim p_\omega} \left[q_\theta(x|\theta) \right] \tag{15.3}$$

Where θ is our parameter, p_θ is the priore, and $q_\theta(x|\omega)$ is our decoder. Our objective is this optimization problem:

$$\theta^* \in \arg\min_{\theta \in \Theta} d(q_\theta, p_{\text{data}}) \tag{15.4}$$

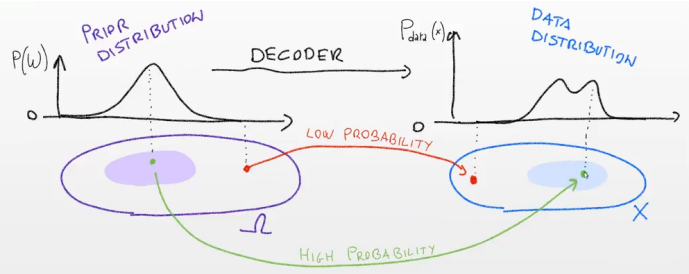


Figure 15.5: AutoEncoder usage.

One possibility to measure the discrepancy (distance) of our prior data to our p_{data} is to use the Kullbakc-Leibler (KL) divergence:

$$d_{KL}(p, q) = \mathbb{E}_{x \sim p}\left[\log \frac{p(x)}{q(x)}\right] \tag{15.5}$$

It turns out that this optimization is particularly difficult, in particular, we can make some calculations:

$$\begin{aligned} d_{KL}(p_{\text{data}}, q_\theta) &= \mathbb{E}_{x \sim p_{\text{data}}}\left[\log \frac{p_{\text{data}}(x)}{q_\theta(x)}\right] && (15.6)\\ &= -\mathbb{E}_{x \sim p_{\text{data}}}\left[\log q_\theta(x)\right] + const && (15.7)\\ &= -\mathbb{E}_{x \sim p_{\text{data}}}\left[\log \mathbb{E}_{\omega \sim p_\omega}\left[q_\theta(x|\omega)\right]\right] + const && (15.8) \end{aligned}$$

but $\mathbb{E}_{\omega \sim p_\omega}\left[q_\theta(x|\omega)\right]$ is an intractable problem, therefore some trick are required.

We only need gradients for the optimization procedure, so, skipping all the derivations, we have that:

$$\frac{\delta}{\delta\theta} d_{KL}(p_{\text{data}}, \_\theta) = -\mathbb{E}_{x \sim p_{\text{data}}} \mathbb{E}_{\omega \sim p_\omega}\left[\frac{\frac{\delta}{\delta\theta} q_\theta(x|\omega)}{\mathbb{E}_{\omega \sim p_\omega}\left[q_\theta(x|\omega)\right]}\right] \tag{15.9}$$

now, we have problems with the expectation $\mathbb{E}_{\omega \sim p_\omega}\left[q_\theta(x|\omega)\right]$, that is still intractable, because we will have some sampling of ω that will yield biased gradient estimates.

15.2.3 Variational Upper Bound

Let $q_\psi(\omega|x) \in \Delta(\Omega)$, where ψ defines other parameters, denote an encoding probability distribution:

$$\log \mathbb{E}_{\omega \sim p_\omega}\left[q_\theta(x|\omega)\right] = \underbrace{\mathbb{E}_{\omega \sim p_\psi(\cdot|x)}\left[\log q_\theta(x|\omega)\right]}_{\text{Reconstruction}} - \underbrace{d_{KL}(q_\psi(\cdot|x), p_\omega)}_{\text{Regularizer}} \tag{15.10}$$

Now we can define a variational bound:

$$d_{KL}(p_{\text{data}}, q_\theta) \leq \mathbb{E}_{x \sim p_{\text{data}}}\left[-\mathbb{E}_{\omega \sim p_\psi(\cdot|x)}[\log q_\theta(x|\omega)] + d_{KL}(q_\psi(\cdot|x), p_\omega)\right] + const \tag{15.11}$$

The reconstruction is still intractable to compute, but is easy to get unbiased estimates of gradients with respect to θ and ψ. The regularizer might have closed-form solution, e.g. using gaussian distributions.

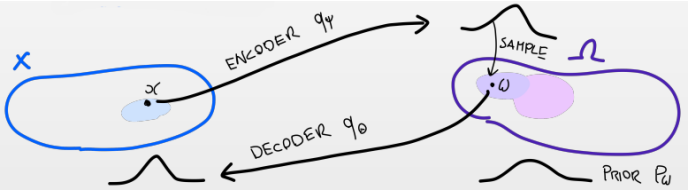


Figure 15.6

15.2.4 Conditional VAE

Even if we are interested in unsupervised learning, Variational AutoEncoders can be extended also to cases in which we have side informations in our training set and we want to generate new data conditioned on these labels. Assume we have side information $y \in \mathcal{Y}$ (e.g. digit labels) and we want to generate new data conditioned on the side information. We need to modify the encoder and decoder to take the side information in input obtaining $q_\psi(\omega|x, y)$ and $q_\theta(x|\omega, y)$.

Similarly we can also define priors conditioned on side information $p_\omega(\omega|y)$

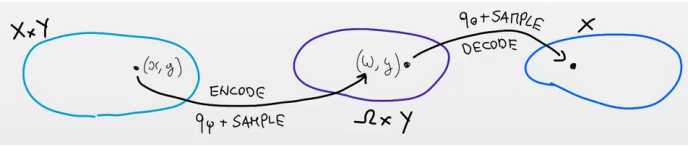


Figure 15.7

15.2.5 Issues with VAEs

Despite VAEs are a very powerful technique to generate data and they are expecially good, they have some problems. For example, since they allow to move to the latent space, the generator tends to produce **blurry data**. Another problem is the problem of **underfitting**, in particular the balance between the regularization and the reconstruction terms must be managed carefully because if the regularizer is too strong, it will tend to be predominant and will tend to annihilate the model capacity.

15.3 Generative Adversarial Networks (GAN)

Generative Adversarial Networks are the most famous generative model, but they start from a different intuition from the Variational AutoEncoder, because the idea of having an implicit density model, so not requiring to estimate $q_\theta(x)$.

GANs enable the possiblity of estimating implicit densities. We assome to have a prior density $p_\omega \in \Delta(\Omega)$ and a generator (or decoder) $g_\theta \in \mathcal{X}^\Omega$ that generates data points in $\mathcal{X}$ given a random element from Ω.

The density induced by the prior p_ω and the generator g_θ is given by:

$$q_\theta(x) = \mathbb{E}_{\omega \sim p_\omega} \delta \left[g_\theta(\omega) - x \right] \tag{15.12}$$

where δ is the Dirac delta function.

15.3.1 Objective

The original GAN objective is to find θ^* such that q_{θ^*}best fits the data distribution p_{data} under the Jensen-Shannon divergence d_{JS}, i.e.

$$\theta^* \in \arg\min_\theta d_{JS}(p_{\text{data}}, q_\theta) \tag{15.13}$$

where

$$d_{JS}(p,q) = \frac{1}{2} d_{KL}\left(p, \frac{p+q}{2}\right) + \frac{1}{2} d_{KL}\left(q, \frac{p+q}{2}\right) \tag{15.14}$$

Since our derivation is intractable to compute — in particular we cannot compute the gradient of the distace, that allows us to train our generator — we have to do some derivation that we will skip, but let's concentrate on the last formula:

$$d_{JS}(p,q) = \frac{1}{2} d_{KL}\left(p, \frac{p+q}{2}\right) + \frac{1}{2} d_{KL}\left(q, \frac{p+q}{2}\right) \tag{15.15}$$

$$= \log(2) + \frac{1}{2} \max_t \left\{ \mathbb{E}_{x \sim p} \left[\log t(x) \right] + \mathbb{E}_{x \sim q} \left[\log(1 - t(x)) \right] \right\} \tag{15.16}$$

where $t(x)$ is defined as $t(x) = \frac{p(x)}{p(x)+q(x)}$, and is like a binary classifier predictin whether x came from p or q.

GAN Objective Lower Bound

Let $t_\phi(x)$ be a classifier (or discriminator) for data point in $\mathcal{X}$. Then we get the following lower boun on our objective:

$$d_{JS}(p,q) = \log(2) + \frac{1}{2} \max_t \left\{ \mathbb{E}_{x \sim p_{\text{data}}} \left[\log t(x) \right] + \mathbb{E}_{x \sim q_\theta} \left[\log(1 - t(x)) \right] \right\} \tag{15.17}$$

$$\geq \log(2) + \frac{1}{2} \max_\phi \left\{ \mathbb{E}_{x \sim p_{\text{data}}} \left[\log t_\phi(x) \right] + \mathbb{E}_{x \sim q_\theta} \left[\log(1 - t_\phi(x)) \right] \right\} \tag{15.18}$$

Which is minimized to obtain the generator's parameter θ^*. Note that in this equation, the terms $\log(2)$ and $\frac{1}{2}$ can be neglected, because they are not changing the minimizer θ^*.

$$\theta^* \in \arg\min_\theta \max_\phi \left\{ \mathbb{E}_{x \sim p_{\text{data}}} \left[\log t_\phi(x) \right] + \mathbb{E}_{x \sim q_\theta} \left[\log(1 - t_\phi(x)) \right] \right\} \tag{15.19}$$

Now note that we have the issue that this still depends on the explicit density q_θ, however we can equivalently rewrite it as

$$\theta^* \in \arg\min_\theta \max_\phi \{\mathbb{E}_{x\sim p_{\text{data}}} [\log t_\phi(x)] + \mathbb{E}_{\omega\sim q_\omega} [\log(1 - t_\phi(g_\theta(\omega)))]\} \quad (15.20)$$

Here we want to minimize the error of the discriminator in recognizing if a sample is real or fake, and we also want to push the generator $g_\theta(\omega)$ to generate samples that are as close as possible to the real data.

15.3.2 Issues with GANs

The main problem with GAN is the **training stability**, because since we have the alternation between the generator and the discriminator, the parameters of the network may oscillate and never converge.

Another problem is the **mode collapse**, the generator might learn to perfectly generate few examples from the training set, but not cover all the variability that is in the training set.

Finally, the last problem we will see, is the **vanishing gradient**, if the discriminator is very successful, then it will have a little gradient to learn, so it leaves the generator with little gradient to learn from, as we can see in Figure 15.8.

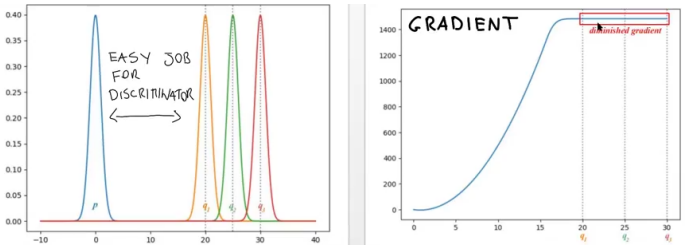


Figure 15.8

15.3.3 More GANs

More GAN-like models can be constructed by considering different divergences between probabilities and by applying similar tricks to get rid of the necessity of knowing the explicit density:

- f-GANs built on f-divergences.
- b-GANs built on Bergman divergences.
- Wasserstein GANs use the Wasserstein metric.
- Other GANs can be derived using integral probability metrics.

Furthermore, GANs and VAEs can be combined (VAE-GAN), and there exist conditional GANs that work like continual VAEs.

Part IV

Reinforcement Learning

16

Reinforcement Learning

Reinforcement Learning came to the stage when the system developed by Google Deep-Mind **AlphaGo**. It has been shown that the best player of the game Go on the earth was beaten from the machine.

We have already discussed supervised and unsupervised learning, now the last machine learning area we need to talk about is the reinforcement learning.

As we have seen, in unsupervised learning we don't have access to any annotations at all for our data, and we've seen methods for clustering, dimensionality reduction, and density estimation. When have talked about supervised learning, we have seen methods for classification and regression, such as Neural Networks, SVM, k-NN, and so on...

We are now into the introduction to the Reinforcement Learning area, that has a completely different scheme of training, where the learning follows an iterative manner.

16.1 The Idea

The idea of reinforcement learning is inspired by the way in which human and animals learns. In this kind of learning, the problems involve an **agent** interacting with an **environment**, which provides numeric rewards signals.

The goal in reinforcement learning is to learn how to take actions in order to maximize the reward.

Let's for example examine the Super Mario game, in this case the environment is the screen of the game, that can be described as the

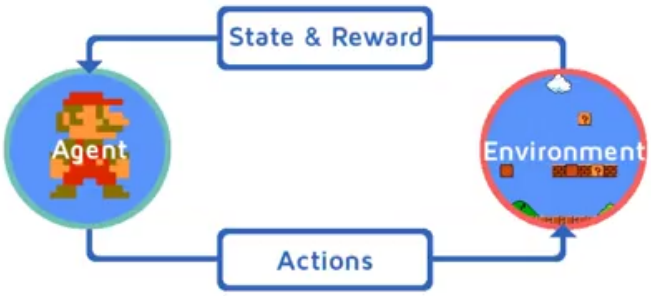


Figure 16.1

state s_t; our agent for this example is Mario, that takes an action a_t, a_t has in impact on the next state of the environment. Depending on the action Mario has made, he will take a negative or postiive reward r_t, then we move to the next state s_{t+1}, and the loop continues until the episode is finished.

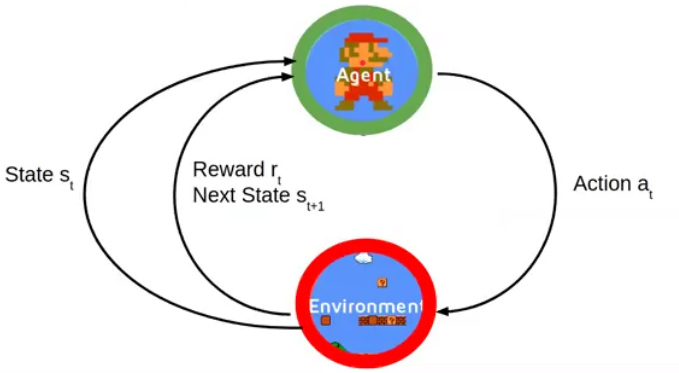


Figure 16.2: Reinforcement Learning Loop example of the Mario game.

Agent can take actions that affect the state of the environment and observe occasional rewards that depend on the state. The goal is to learn a **policy** to maximize the expected reward over time, where a *policy* is a mapping from states to actions.

Example 16.1.0 — Reinforcement Learning in Atari Games

Let's consider for example the Atari games. We can perform reinforcement learning with the following assumption:

- The **objective** is to complete the game with the highest score.
- The **state** are the raw pixel inputs of the game state.
- The **actions** are the game controls (e.g. left, righ, up and down).
- The **reward** is the score, that increases or decreases at each time step.

16.2 Markov Decision Process

The **Markov Decision Process (MDP)** is a framework used to hep to make decisions on a stochastic environment. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. Our goal is to find a policy, which is a map that gives us all optimal actions on each state in our environment. In order to solve MDPs we need Dynamic Programming (DP), more specifically the **Bellman equation**.

Recall that Dynamic Programming is a method that divides a problem into simple sub-problems easier to solve, but, differently from the divide and conquer method, it stores the results that are repeated in a table, in order to make the algorithm more efficient.

At each time step, the process is in some state s, and the decision maker may choose any action a that is available in state s. The process responds at the next time step by randomly moving into a new state s', and giving the decision make a corresponding reward $R_a(s, s')$.

The probability that the process moves into its new state s' is influenced by the chosen action. Specifically, it is given by the state transaction function $P_a(s, s')$. Thus, the next state s' depends on the current state s and the decision maker's action a. But given s and a, it is *conditionally independent* of all previous states and actions.

16.2.1 Definition

A Markov decision process is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$, where:

- $\mathcal{S}$ is a set of states called the **state space**,
- $\mathcal{A}$ is a set of actions called the **action space**, alternatively $\mathcal{A}_s$ is the set of actions available from state s,
- $\mathcal{R}_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a,

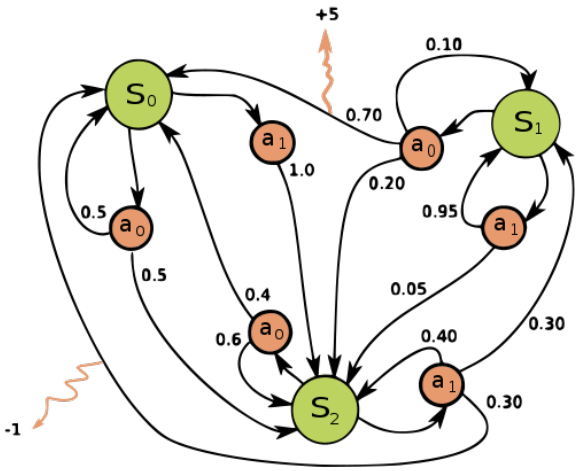


Figure 16.3: Example of a simple MDP with three states (big circles) and two actions (small circles), with two rewards (waved arrows).

- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- γ is the discount factor.

A policy function π is a (potentially probabilistic) mapping from state space $\mathcal{S}$ to an action space $\mathcal{A}$.

16.2.2 MDP Loop

At time step $t = 0$, the environment samples the initial state $s_0 \sim p(s_0)$. From that initialization, starts the loop, repeat the following steps:

- Agent selects action a_t;
- Environment samples reward $r_t \sim \mathcal{R}(\cdot|s_t, a_t)$,
- Environment samples the next state $s_{t+1} \sim P(\cdot|s_t, a_t)$,
- Agent receives reward r_t and next state s_{t+1}.

16.2.3 Objective

The goal in Marvok Decision Process is to find a good policy for the decision maker: a function pi that specifies the action $\pi(s)$ that the decision maker will choose when in state s. Once a MDP is combined with a policy in this way, this fixes the action for each state.

The objective is to choose a policy π that will maximize some cumulative function of the random rewards. A policy that maximizes the cumulative discounted reward is called an optimal policy π^*.

Cumulative Discounted Rewards

Suppose the following policy π starting in state s_0 leads to a sequence $s_0, s_1, s_2, ...$

The cumulative reward of the sequence is:

$$\sum_{t \geq 0} r(s_t) \tag{16.1}$$

State sequences can vary in length or een be infinite. Typically we define the cumulative reward as sum of rewards discounted by a factor γ:

$$r(s_0) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ... = \tag{16.2}$$

$$= \sum_{t \geq 0} \gamma^t r(s_t), \qquad 0 < \gamma \leq 1 \tag{16.3}$$

The discounting factor γ controls the importance of the future rewards versus the immediate ones. The lower the discount factor is, the less important future rewards are, and the agent will tend to focus on actions which will yield immediate rewards only. The cumulative reward is bounded, and this helps the algorithm to converge.

16.2.4 Reinforcement Learning vs Supervised Learning

In supervised learning, the loop follows these steps:

- Get input x_i samples from data distribution.
- Use model with parameters w to predict output y.
- Observe target output y_i and loss $l(w, x_i, y_i)$.
- Update w to reduce loss with SGD:

$$w \leftarrow w - \eta \nabla l(w, x_i, y_i) \tag{16.4}$$

While, as we have seen, the reinforcement learning loop follows these steps:

- From state s, take action a determined by policy $\pi(s)$.
- Environment selects next state s' based on transition model $P(s'|s, a)$.
- Observe s' and reward $r(s)$, then update the policy.

In Supervised learning the next input does not depend on the previous inputs or agent predictions, there is a supervision signal at every step, and the loss is differentiable with respect to model parameters.

In reinforcement learning agent's actions affect the environment and help to determine the next observation, the rewards may be sparse and are not differentiable with respect to model parameters.

16.3 Value Based Methods

The **value function** gives the total amount of reward the agent can expect from a particular state to all possible states from that state. With the value function you can find a policy. The value function V of a state s with respect to policy π is hte expected cumulative reward of following that policy starting in s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r(s_t)|s_0 = s, \pi\right] \tag{16.5}$$

with $a_t = \pi(s_t), s_{t+1} \sim P(\cdot|s_t, a_t)$.

The **optimal value of a state** is the value achievable by ollowing the best possible policy:

$$V^{*}(s) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r(s_t)|s_0 = s, \pi\right] \tag{16.6}$$

16.3.1 Q-Learning

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" refers to the function that the algorithm computes — the expected rewards for an action taken in a given state.

Q-Value Function

It is more convenient to define the value of a state-action pair, instead of just the state:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r(s_t)|s_0 = s, a_0 = a, \pi\right] \tag{16.7}$$

In this case, the optimal Q-value function tells how good is a state-action pair:

$$Q^{*}(s, a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r(s_t)|s_0 = s, a_0 = a, \pi\right] \tag{16.8}$$

When the optimal Q-value is found it is used to compute the optimal policy:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{16.9}$$

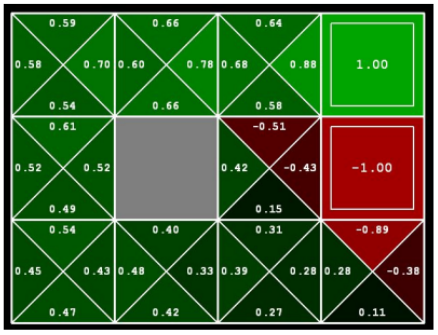


Figure 16.4: Table where we have the maximum expected future reward, for each action at each state.

Bellman Equation

The Bellman equation is a necessary condition for optimality associated with the mathematical optimization method known as dynamic programming.

Recursive relationship between optimal values of successive states and actions:

$$Q^*(s,a) = r(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a') \tag{16.10}$$

$$= \mathbb{E}_{s' \sim P(\cdot|s,a)[r(s)+\gamma \max_{a'} Q^*(s',a')|s,a]} \tag{16.11}$$

If the optimal state-action values for the next time-step $Q^*(s',a')$ are known, then the optimal strategy is to take the action that maximizes the expected value.

Algorithm

After Δt steps into the future, the agent will decide some next step. The weight for this step is calculated as $\gamma^{\Delta t}$, where γ has the effect of valuing rewards reeiving earlier higher than those received later, it may also be interpreted as the probability to succeed at every step Δt.

The algorithm has a funciton that calculates the quality of a state-action combination, that is the $Q(s,a)$. Before learning begins, Q is initialized to a possibly arbitrary fixed value. Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1}, and Q is updated.

As an example, suppose the robot in Figure 16.5 needs to reach room 5. Our components are:

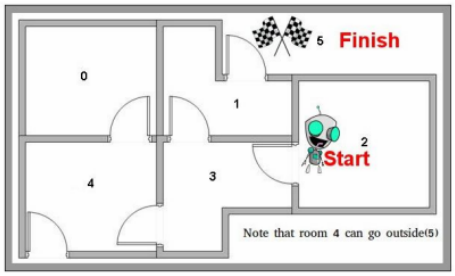


Figure 16.5: We want an agent that is able to find the goal state (5) from the initial state (i.e. 2).

- Actions $\mathcal{A} = \{0, 1, 2, 3, 4, 5\}$,
- States $\mathcal{S} = \{0, 1, 2, 3, 4, 5\}$,
- Rewards = {0,100}.

Our goal state is 5. The reward table $\mathcal{R} = \mathcal{S} \times \mathcal{A}$ is computed based on the possible actions at a certain state, where the value -1 indicate some specific action is not available. The whole point of Q-learning is that the

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

Figure 16.6

matrix $\mathcal{R}$ is available only to the environment, the agent need to learn $\mathcal{R}$ by himself through experience.

What the agent will have is a Q matrix that encodes the state, the action, and the rewards, but is initialized with zero and through experience becomes like the matrix $\mathcal{R}$. The policy can then be obtained from the Q matrix.

Algorithm 17: Q-Learning

$Q \leftarrow [1...s][1...a] = \{0...0\}\{0...0\}$;
$s_0 \leftarrow random$;
for *episode in* E **do**
  while *state* $s_i \neq s_{goal}$ **do**
    Select a random possible action a_r for s_i;
    Using a_r, consider going to this next state;
    Get maximum Q value for this next state;
    $Q^*(s, a) \leftarrow \mathcal{R}(s, a) + \gamma \max_{a'}[Q^*(s', a')]$;
  end
end

16.3.2 Deep Q-Learning

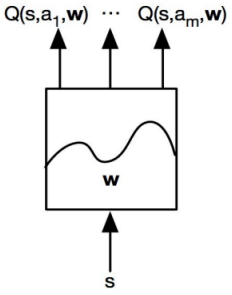


Figure 16.7

The Bellman equation is a constraint on Q-values of successive states, the problem is that state spaces for interesting problems are huge (i.e. Atari game), the solution is to approximate Q-values using a parametric function:

$$Q^*(s,a) \approx Q_w(s,a) \tag{16.12}$$

We can train a deep network that approximates Q. This is represented in Figure **??**.

The idea is that, at iteration of training, we can update a the models parameter w to push Q close to y:

$$y_i(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[r(s) + \gamma \max_{a'} Q_{w_{i-i}}(s',a') | s,a \right] \tag{16.13}$$

The loss function (that change at each iteration) is defined as:

$$L_i(w_i) = \mathbb{E}_{s,a \sim \rho} \left[(y_i(s,a) - Q_{w_i}(s,a))^2 \right] \tag{16.14}$$

where ρ is a probability distribution over states s and actions a that we refer to as the **behaviour distribution**.

In practice the gradient update will be:

$$\begin{aligned} \nabla_{w_i} L(w_i) &= \mathbb{E}_{s,a \sim \rho} \left[(y_i(s,a) + Q_{w_i}(s,a) \nabla_{w_i} Q_{w_i}(s,a) \right] && (16.15) \\ &= \mathbb{E}_{s,a \sim \rho, s'} \left[(r(s) + \gamma \max_{a'} Q_{w_{i-i}}(s',a') - Q_{w_i}(s,a)) \nabla_{w_i} Q_{w_i}(s,a) \right] && (16.16) \end{aligned}$$

SGD training: replace expectation by sampling **experiences** (s,a,s') using behaviour distribution and transition model.

The training is prone to instability, unlike in supervised learning, the targets themselves are moving, successive experiences are correlated and dependent on the policy, and the policy may change rapidly with slight changes to parameters, leading to drastic change in data distribution.

Solutions to the training instability can be to freeze the target Q network, or we can use the experience replay, that is a buffer that stores experiences from which we can sample.

16.4 Policy Gradient Methods

Instead of indirectly representing the policy using the Q-values, it can be more efficient to parametrize π and learn it directly. Especially in large or

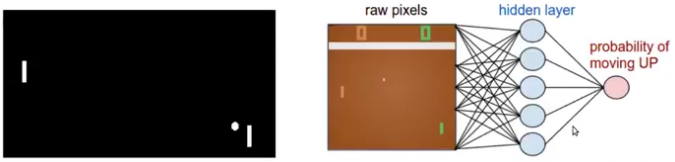


Figure 16.8: Policy Gradient for Pong game.

continuous action spaces the Q-value function can be very complicated, for example a robot grasping an object has a very high-dimensional state and it is hard to learn exact value of every (state, action) pair.

In this case it is beneficial to learn a function giving the probability distribution over actions from current state:

$$\pi_\theta(s, a) \approx P(a|s) \tag{16.17}$$

where the parameter θ is the parameter of our neural network.

Let's take as an example the pong game: the basic idea is to use a machine learning model that will learn a good policy from playing the game and receiving rewards.

16.4.1 Objective Function

Our objective function needs to find the best parameter θ (parameters of the policy) to maximize the expected reward (use gradient descent):

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right] \tag{16.18}$$

$$= \underbrace{\mathbb{E}_\tau[r(\tau)]}_{\text{Expectation of return over trajectories } \tau=(s_0,a_0,r_0,s_1,a_1,r_1,\ldots)} \tag{16.19}$$

$$= \int_\tau r(\tau) p(\tau;\theta) d\tau \tag{16.20}$$

where $p(\tau;\theta)$ is the probability of trajectory τ under policy with parameters θ, and is defined as follows:

$$p(\tau;\theta) = \prod_{t\geq 0} \pi_\theta(s_t, a_t) P(s_{t+1}|s_t, a_t) \tag{16.21}$$

Optimization

We need a convenient way to compute the graident of $[r(\tau)]$, in order to do that it turns out that the gradient can be expressed in the following way:

$$\nabla_\theta J(\theta) = \mathbb{E}[r(\tau) \nabla_\theta \log p(\tau;\theta)] \tag{16.22}$$

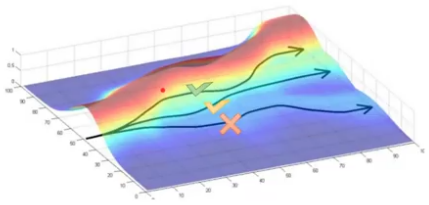


Figure 16.9

Now, we can continue our computations with some rules of the logarithms for the probability of trajectory τ:

$$p(\tau;\theta) = \prod_{t\geq 0} \pi_\theta(s_t, a_t) P(s_{t+1}|s_t, a_t) \tag{16.23}$$

$$\log p(\tau;\theta) = \sum_{t\geq 0} [\log \pi_\theta(s_t, a_t) + \log P(s_{t+1}|s_t, a_t)] \tag{16.24}$$

$$\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(s_t, a_t) \tag{16.25}$$

In this way we do not need to know any information about the environment dynamics p. We can now compute the gradient of our objective function:

$$\nabla_\theta(\theta) = \mathbb{E}_\tau \left[\left(\sum_{t\geq 0} \gamma^t r_t \right) \left(\sum_{t\geq 0} \nabla_\theta \log \pi_\theta(s_t, a_t) \right) \right] \tag{16.26}$$

We can compute this expecation by sampling N trajectories $\tau_1, ..., \tau_N$:

$$\nabla_\theta(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T_i} \gamma^t r_{i,t} \right) \left(\sum_{t=0}^{T_i} \nabla_\theta \log \pi_\theta(s_{i,t}, a_{i,t}) \right) \tag{16.27}$$

16.4.2 Reinforce Algorithm

This leads to the **reinforce algorithm**, the intuition behind that is that if we are going up the hill it means that we are receiving an higher reward, we will change the model parameters and thus the policy to increase the likelihood of trajectories that move higher. A representation of that can be found in Figure 16.9

Algorithm 18: Reinforce

Sample N trajectories τ_i using current policy π_θ;
Estimate the policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} r(\tau_i) \left(\sum_{t=0}^{T_i} \nabla_\theta \log \pi_\theta(s_{i,t}, a_{i,t}) \right) \tag{16.28}$$

Update the parameters by gradient ascent:

$$\theta \leftarrow \theta + \eta \nabla_\theta J(\theta) \tag{16.29}$$