Introduction to Computational Intelligence

Editors

Leandro L. Minku, George Cabral,
Marcella Martins, Markus Wagner

An IEEE Computational Intelligence Society Open Book

Preface

Due to its power to solve large scale problems and analyse vast amounts of data that would be difficult or time consuming for humans to deal with manually, the field of Computational Intelligence has grown tremendously in importance over the past years. We nowadays see widespread use of computational intelligence in the most varied applications. A few examples include approaches to detect credit card fraud, recognise faces, transcribe voice to text, identify spam, route and schedule deliveries, design aerodynamic high speed trains, etc.

It is thus not surprising that we see a growing number of people who are keen to learn about this field. However, there is a lack of open resources that combine several different types of computational intelligence approaches in one place, so that people can easily get an introduction to this field. Those eager to learn about computational intelligence may also struggle to get help from others when trying to understand existing approaches, whereas those willing to start teaching this topic may struggle to find free resources to guide them.

This *open* book has been created as a community effort to overcome these issues. The notion of *openness* of this book includes, but goes beyond open access. In addition to being available through an open license so that resources on computational intelligence are accessible to all, this book is hosted in github at:

`https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1`.

Such initiative will enable the book to be continuously improved over time through pull requests to fix typos, add clarifications, add new exercises, add examples of open software code, add video lectures on the content, etc. Therefore, this book is *open* for the community to propose enhancements over time. The book is also associated to github discussion boards, so that people can ask questions and the community can help with answering those questions, creating an *open* community that all can join in.

If you would like to propose an enhancement through a pull request to this book, we ask you to first contact the current chair of the IEEE Computational Intelligence Society Education Portal Subcommittee (`https://cis.ieee.org/`). The chair will advise you on how to proceed. Minor changes to existing chapters will be handled by the subcommittee directly, whereas the subcommittee will liaise with the original authors to obtain their consent for incorporating larger pull requests.

We hope that you will find the book a useful resource to learn about computational intelligence.

Leandro L. Minku, on behalf of the editors of the
IEEE CIS Computational Intelligence Open Book – First Edition

About the Editors

Dr. Leandro L. Minku is an Associate Professor at the School of Computer Science, University of Birmingham (UK). Prior to that, he was a Lecturer in Computer Science at the University of Leicester (UK). He received the PhD degree in Computer Science from the University of Birmingham (UK) in 2010. Dr. Minku's main research interests are machine learning in non-stationary environments / data stream mining, online class imbalance learning, ensembles of learning machines and computational intelligence for software engineering. His work has been published in internationally renowned journals such as IEEE Transactions on Neural Networks and Learning Systems, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Software Engineering and ACM Transactions on Software Engineering and Methodology. Among other roles, Dr. Minku is Associate Editor-in-Chief for Neurocomputing, Senior Editor for IEEE Transactions on Neural Networks and Learning Systems, and Associate Editor for Empirical Software Engineering Journal and Journal of Systems and Software. He was also the General Chair for the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2019 and 2020), and Co-chair for the Artifacts Evaluation Track of the International Conference on Software Engineering (ICSE 2020).

Dr. Markus Wagner is an Associate Professor at Faculty of Information Technology, Monash University (Australia). He has done his PhD studies at the Max Planck Institute for Informatics in Saarbruecken, Germany and at the University of Adelaide, Australia. For the outcomes of his studies, he has received the university's Doctoral Research Medal — the first for his school — and three best paper awards. His research topics range from mathematical runtime analysis of heuristic optimisation algorithms and theory-guided algorithm design to applications of heuristic methods to renewable energy production, professional team cycling and software engineering. So far, he has been a program committee member 80+ times, and he has written 150+ articles with 200+ different co-authors. He has chaired several education-related committees within the IEEE CIS, where he also served as founding chair of two task forces. Markus is an ACM Lifetime Member, is on SIGEVO's Executive Board and serves as the first ever Sustainability Officer. He has contributed to GECCOs as Workshop Chair and Competition Chair, and most recently as General Chair.

Dr. George Cabral received his PhD degree from the Federal University of Pernambuco (Brazil) in 2014. His postdoc was conducted at the University of Birmingham (UK) in 2019. Currently, he is an adjunct professor at the Department of Computing at the Federal Rural University of Pernambuco (Brazil). He serves as reviewer for reputed journals such as Neurocomputing and Applied Software Computing. In addition, he has systematically served as committee member in prestigious conferences such as ICSE, ASE and ICONIP. His recent plublications include papers in reputed venues such as IEEE Transactions on Software Engineering and at the International Conference on Software Engineering (ICSE). He has taught courses involving Computational Intelligence for more than a decade and since 2020 he is applying this knowledge in practice for auditing public finances at the state of Pernambuco - Brazil. His research interests include Novelty Detection, One-class Classification, Data Mining, Class Imbalance Learning, Software Defect Prediction, Concept Drift, Online Learning, etc.

Marcella Scoczynski is an Assistant Professor at Federal University of Technology - Parana UTFPR, Brazil. She has done her PhD on Computer Engineering at Federal University of Technology - Parana UTFPR, Brazil. Her thesis has awarded at the Theses Competition during Brazilian Conference on Intelligent Systems (BRACIS 2018) and at the Theses Contest during 5th IEEE Latin American Conference on Computational Intelligence (LA-CCI 2018). Her main research interests are numerical and combinatorial optimization, evolutionary computation and metaheuristics (with a particular interest in estimation of distribution algorithms), and landscape analysis. She is a researcher in projects conducted by Frontier Development Lab (FDL), NASA and SETI Institute.

List of Contributors

The list below contains information about contributors who have made large revisions to chapters of this book.

Contributor	Chapter	Commit
Leandro L. Minku	Chapter III	https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/commit/186ef1d4c1c2b47047981f7cbb8dc8d05dd80651
		https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/commit/b5ca72bf53302ea3a307afb16fd34519204746b5

Contents

Part III Learning Systems

Part III-(A) Supervised Learning

List of Figures

List of Tables

Mathematical Notation

In general, the following mathematical notations will be used in this book:

- Scalar: lower case, e.g., a, b.
- Column vector: lower case, bold, e.g., $\mathbf{x}$.
- Vector element: lower case with subscript, e.g., x_1, x_2.
- If enumerating vectors (e.g., having multiple vectors), superscript will be used to differentiate this from indices, e.g., $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$.
- Matrix: upper case, bold, e.g., $\mathbf{X}$.
- Matrix element: upper case with subscripts, e.g., $X_{1,2}$.
- When considering that a matrix is a vector of vectors, row i and column j can be represented by $\mathbf{x}_j^{(i)}$.
- Sets: calligraphy font in upper case, e.g., $\mathcal{T}$.
- Generic data structure with unspecified format (e.g., it could be a vector, a matrix, or any other structure): lower case, bold, e.g., $\mathbf{x}$.

Part I
History and Definitions of Computational Intelligence

Chapter 1
Introduction

Leandro L. Minku

Even though the term *Computational Intelligence* (CI) has been used for many years, no single agreed definition exists so far. Traditionally, CI has been considered to be the "theory, design, application and development of biologically and linguistically motivated computational paradigms" [1]. Possibly, this definition has been proposed because many researchers adopting the term CI were associated to the IEEE Computational Intelligence Society (CIS), which has roots in the IEEE Neural Networks Society (NNS) and its predecessor the IEEE Neural Networks Council (NNC).

As explained in [2], the IEEE Neural Networks Council (NNC), established in November 1989, was the publisher of the IEEE Transactions on Neural Networks, the IEEE Transactions on Evolutionary Computation, and the IEEE Transactions on Fuzzy Systems journals. Their field of interest was specified as "the theory, design, application, and development of biologically and linguistically motivated computational paradigms" [3], which precisely matches the CI definition given above. The IEEE NNC then transitioned to become the IEEE NNS in November 2001, with a view of continuing to focus on the same field of interest [3]. In November 2003, the IEEE NNS then changed its name to IEEE CIS.

As evidenced by the three journals that were originally published by the IEEE NNC, the three main pillars of CI have traditionally been neural networks, evolutionary computation, and fuzzy systems [1]. These are biologically- and linguistically-inspired topics, being well aligned with the CI definition above. However, many different biologically-inspired algorithms have been proposed since then. Moreover, key events sponsored or technically co-sponsored by the IEEE CIS nowadays also include other CI topics that are not necessarily biologically- or linguistically-inspired. This includes flagship conferences such as the International Joint Conference on Neural Networks (IJCNN) and the IEEE Congress on Evolutionary Computation

University of Birmingham, UK

(IEEE CEC). Therefore, the definition above does not include all topics that may currently be referred to as CI topics.

As of October 2022, the Wikipedia entry of Computational Intelligence [4] explains that the "expression computational intelligence (CI) usually refers to the ability of a computer to learn a specific task from data or experimental observation." This definition is fairly general, being well aligned with topics covered in some of the key conferences in the field and not requiring biological or linguistic inspiration. In particular, this definition is inclusive but not limited to the traditional CI pillars of neural networks and evolutionary computation. However, it arguably does not match very well the traditional pillar of fuzzy systems. This is because fuzzy systems do not necessarily learn from data or experimental observation. Instead, they may be based on linguistically-inspired knowledge bases provided by humans.

According to Bezdek [5], it is believed that the term CI originated from the name of the Canadian's Artificial Intelligence (AI) society founded in 1974: the Canadian Society for Computational Studies of Intelligence. Nick Cercone and Gordon McCalla, who were members of the society, decided to create an AI journal. After much debate and some influence from the term "Computer Vision", they decided that the term CI was more adequate to describe their field than the term AI [5]. Therefore, they created the journal named International Journal of Computational Intelligence (IJCI) in 1983. It is unclear from Bezdek [5]'s account whether this means that the term CI was originally intended to be a sub-field of AI. However, this is a possible way to view the field of CI.

The term AI itself has many different possible definitions [6]. One of the well received definitions is the one explained by Russell and Norvig [6]: AI is concerned with the study of agents that act rationally, i.e., computer programs that act so as to achieve the best (expected) outcome. This is a very general definition that can be seen as incorporating all of the CI approaches. It includes the three traditional CI pillars as well as other non-biologically-inspired and non-linguistically-inspired approaches commonly seen in current CI venues. It includes CI approaches that learn to perform a task based on data or experimental observation, and those that perform a task based on linguistically-inspired approaches such as fuzzy systems. It also includes approaches that are not usually considered as CI approaches, such as agents that act rationally based on a knowledge base consisting of crisp logical statements (i.e., statements using boolean logics such as propositional logic, first order logic, etc). Therefore, *one may consider CI to be AI algorithms that are not based on crisp logical statements*. This is the view that this book will take. The focus will be on the algorithms, rather than on the systems where they may be embedded.

The next parts of this book will give several examples of CI algorithms that fit within this definition. By the end of this book, we hope readers to become more familiar with the field, gaining a better understanding of what this definition means and entails. However, as pointed out by Bezdek [5], many

different definitions of CI and arguments exist [7, 8, 9, 10, 11, 12, 13, 1]. Therefore, even though this book is adopting this definition, it is important to emphasize that this is not a universal definition.

References

1. What is computational intelligence?, `https://cis.ieee.org/about/what-is-ci`, Accessed in October 2022.
2. Evolution of the ieee computational intelligence society (cis), `https://cis.ieee.org/committees/history-committee/history/evolution`, Accessed in October 2022.
3. P. P. Bonissone. Editorial: Welcome to the ieee neural networks society. *IEEE Transactions on Evolutionary Computation*, 6:531–532, 2002.
4. Computational ingelligence, `https://en.wikipedia.org/wiki/Computational_intelligence`, Accessed in October 2022.
5. J. C. Bezdek. (computational) intelligence: What's in a name? *IEEE Systems, Man, and Cybernetics Magazine*, pages 4–14, 2016.
6. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition edition, 2010.
7. R. Marks. Intelligence: Computational versus artificial. *IEEE Transactions on Neural Networks*, 4:737–739, 1993.
8. J. C. Bezdek. On the relationship between neural networks, pattern recognition and intelligence. *International Journal of Approximate Reason*, 6:85–107, 1992.
9. J. C. Bezdek. What is computational intelligence? In J. M. Zurada, R. J. Marks, and C. J. Robinson, editors, *Computational Intelligence: Imitating Life*, pages 1–12. IEEE Press, 1994.
10. J. C. Bezdek. Computational intelligence: A snapshot. In M. Palaniswami, Y. Attikiouzel, R. J. Marks, D. Fogel, and T. Fukuda, editors, *Computational Intelligence: A Dynamic System Perspective*, pages 9–15. IEEE Press, 1995.
11. R. Eberhart, R. W. Dobbins, and P. K. Simpson. *Computational Intelligence PC Tools*. Academic, 1996.
12. D. Fogel. Review of the book computational intelligence: Imitating life. *IEEE Transactions on Neural Networks*, 6:1562–1565, 1995.
13. J. C. Bezdek. Computational intelligence defined—by everyone! In O. Kaynak, L. A. Zadeh, B. Turksen, and I. J. Rudas, editors, *Computational Intelligence: Soft Computing and Fuzzy-Neuro Integration with Applications*, volume NATO ASI series F, v. 162, pages 10–37. Springer-Verlag, 1998.

Part II
Search-Based Optimization

Chapter 2
Introduction

Leandro L. Minku

Many real world problems involve finding solutions that minimize or maximize one or more functions. For example:

- Routing problems, e.g., to find a path from a city of origin to a city of destination that minimizes the distance travelled, while ensuring that non-existent direct paths between any two cities are not used.
- Bin packing problems, e.g., to find an assignment of items to bins that minimizes the number of bins used, while ensuring that the maximum volume of the bins is not exceeded.
- Scheduling problems, e.g., to find an allocation of staff to tasks in a software project, so as to minimize the cost and duration of this project, while ensuring that staff are only allocated to tasks for which they have the necessary skills.
- Machinery configuration problem, e.g., to find the numeric values of certain parameters that will maximise the efficiency of a machine.

Such problems are called *optimization problems*. The variables that one wishes to optimize (e.g., the path in a routing problem, the assignment in a bin packing problem, and the allocation in a scheduling problem) are referred to as *design variables*. They represent candidate solutions to the problem. Problems where the design variables are discrete are referred to as combinatorial optimization problems, whereas problems where the design variables are continuous are referred to as continuous optimization problems. In practice, a problem may have a mix of discrete and continuous variables.

The functions to be optimized (e.g., the travelled distance in a routing problem, the number of bins in a bin packing problem, the cost and duration in a scheduling problem and the efficiency in a machine configuration problem) are referred to as *objective functions*. When the function is to be

School of Computer Science, University of Birmingham, UK

minimized, the objective functions are also often called *cost functions*, and when it is to be maximized, they are often called *quality functions*.

Optimization problems also frequently impose constraints on the solutions (e.g., ensuring that non-existent direct paths are not used in a routing problem, ensuring that the maximum volume of the bins is not exceeded in a bin packing problem, and ensuring that staff have the necessary skills in a scheduling problem). When such constraints exist, a solution is only valid if it satisfies these constraints. Candidate solutions to optimization problems are referred to as *feasible* if they satisfy the constraints of the problem, and *infeasible* when they fail to satisfy one or more constraints.

The solution that maximizes / minimizes the objective functions is referred to as the optimum. Some optimization problems have a single optimum whereas others may have multiple optima. Many real world problems have *local optima*. These are solutions that are better than other solutions in their neighbourhood, but are not the optimal solutions for the problem. In such case, the optimal solutions are typically referred to explicitly as *global optima* rather than just optima, not to cause confusion with the local optima.

Some real world problems may require a global optimum to be found. However, in many real world problems, it is considered acceptable to find a local optimum whose objective values are good enough / close enough to those of the global optima. This is a particularly important point to consider when deciding which algorithm to adopt for solving a given problem. Many optimization algorithms are not guaranteed to find a global optimum within a reasonable amount of time, but computational intelligence algorithms can usually find good enough solutions fast enough. Together with their multi-purpose applicability, this makes them very attractive for adoption in practice.

Section 2.1 will explain the elements required to formally define optimization problems. Section 2.2 then explains what are search-based optimization algorithms, which are computational intelligence algorithms to solve these problems. The next chapters of the book will provide several examples of search-based optimization algorithms and optimization problems that can be solved by them.

It is worth noting that, when there are multiple objectives, it may happen that there is no solution that can maximize / minimize all of the objectives at the same time. This is because objectives are typically conflicting with each other. For example, an allocation of staff to tasks in a scheduling problem that minimises cost may result in a high duration for the project. In this case, one may be interested in finding an optimal *set* of solutions, where each solution represents a different trade-off among the objectives. Solutions within this optimal set are better than solutions not in this set in terms of the objectives being optimized. However, no solution in this set can be considered as better than any other solution in this set when taking all objectives into account at the same time. For example, an allocation with the minimum possible cost but a large duration cannot be considered better or worse than a solution

with the minimum possible duration but a large cost when considering both objectives at the same time. This introductory book will focus on problems with a single objective to be optimized, but some algorithms for problems with multiple objectives will be briefly discussed.

2.1 Formulating Optimization Problems

Without loss of generality, an optimization problem is a problem with the following canonical form:

$$\begin{array}{lll} \text{minimize} & f_k(\mathbf{x}), & k = \{1, 2, \cdots, p\} \\ \text{subject to} & g_i(\mathbf{x}) \leq 0, & i = \{1, 2, \cdots, m\} \\ & h_j(\mathbf{x}) = 0, & j = \{1, 2, \cdots, n\} \end{array}$$

Here, we wish to find a solution $\mathbf{x} \in \mathcal{X}$ that minimizes the functions $f_k(\mathbf{x})$ while satisfying the constraints $g_i(\mathbf{x}) \leq 0$ and $h_i(\mathbf{x}) = 0$, where p is the number of objective functions to be optimised, m is the number of constraints of the type g_i, n is the number of constraints of the type h_j, and $\mathcal{X}$ is the domain of $\mathbf{x}$.

Such kind of problem contains three main components:

- *Design variables* $\mathbf{x}$. These are the variables that represent candidate solutions to the optimization problem. The domain $\mathcal{X}$ of $\mathbf{x}$ depends on the optimization problem. In particular, the variables could be numeric, categorical or ordinal variables, or any other kind of variable that may be relevant to the problem. In particular, $\mathbf{x}$ could be composed of variables of different types. It is also worth noting that even though $\mathbf{x}$ could be a vector of variables, it could also be a matrix or some other data structure. For example, if one is trying to allocate staff members to tasks in a problem, it would be reasonable to consider that $\mathbf{x}$ is a matrix where the rows represent staff members and the columns represent tasks, where each position $x_{i,j}$ contains the value 1 if staff member i is allocated to task j and 0 otherwise.
  The design variables and their domain define the *search space* of the optimization problem. This is the space of all possible candidate solutions to the problem.
- *Objective function(s)* $f_k(\mathbf{x})$, where $k = \{1, 2, \cdots, p\}$. These are the functions that we wish to optimise (maximize or minimize). They represent potentially conflicting goals. We refer to a problem where $p = 1$ as a single-objective optimization problem, and to a problem where $p > 1$ as a multi-objective optimization problem. When dealing with a single-objective optimization problem, the k value is frequently omitted from the name of the objective function, i.e., we typically write $f(\mathbf{x})$ instead of

$f_1(\mathbf{x})$. You may also hear the term many-objective optimization problems when there are $p > 3$ objectives.
You will also note that the canonical form above lists a minimisation problem. We could replace "minimize" by "maximize" in the canonical form above if we are dealing with a maximisation problem. It is also possible to use a mix of objectives to be minimized and maximized. However, as it is possible to convert maximisation problems into minimisation problems, the canonical form of optimization problems typically lists only minimisation without loss of generality.
The objective values associated to each possible value of the design variables are frequently referred to as the objective landscape. This is because, when plotting them, the plot can resemble a landscape with mountains, valleys and plateaux. For example, Figure 2.1 shows the objective landscape for an illustrative continuous optimization problem with a 1-dimensional real valued design variable. It is worth noting, though, that this figure is for illustrative purposes only. In real world problems, the objective landscapes can be very complex and involve multi-dimensional design variables. The exact landscape for such problems is not known beforehand and would not be possible to entirely plot, reason why an optimization algorithm is necessary to find the optimum/optima.

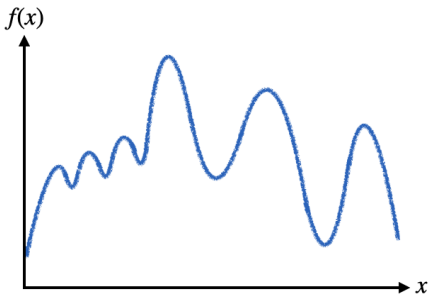


Fig. 2.1: Example of objective landscape for an illustrative continuous optimization problem with a 1-dimensional design variable.

- *Constraint(s)* $g_i(\mathbf{x}) \leq 0$ and $h_j(\mathbf{x}) = 0$, where $i = \{1, 2, \cdots, m\}$ and $j = \{1, 2, \cdots, n\}$. These are the constraints that a candidate solution $\mathbf{x}$ needs satisfy in order to be a feasible solutions to the problem. When $m = 0$ and $n = 0$, the problem is called an unconstrained optimization problem. When $m > 0$ or $n > 0$, the problem is called a constrained optimization problem.
The g_i type of constraints are called *inequality constraints*, whereas the h_j type of constraints are called the *equality constraints*. Problems may also involve constraints with inequalities of the type $\geq$ instead of $\leq$. However,

it is possible to convert inequalities of the type $\geq$ to inequalities of the type $\leq$, reason why the canonical form above lists only $\leq$ without loss of generality.
Strict inequalities ($>$ or $<$) are typically not used in optimization problems, as they can lead to ill-posed problems. For instance, consider a problem where we wish to minimize $f(x) = x^2$, subject to $x > 0$, where x is a real value. Had the constraint been $x \geq 0$, the optimal solution would have been $x = 0$. However, as the constraint is $x > 0$, there is no minimizing value. In particular, we can always get x values that are closer and closer to zero, without ever reaching a minimum. Alternatively, if x was an integer value, this problem would not occur. However, in this case, it would be possible to convert the strict inequality $x > 0$ into the inequality $x \geq 1$, meaning that the canonical form of the optimization problem does not need to have strict inequality constraints.
Some people also use terms such as *hard constraints* and *soft constraints*. When such terms are used, hard constraints refer to constraints that must be satisfied for the solution to be feasible, whereas soft constraints are constraints that we wish to satisfy, but that do not really lead to infeasible solutions when violated.

In order to formulate an optimization problem, all the components above must be specified. The more mathematical the problem formulation is, the less ambiguous it is likely to be. However, more mathematical formulations typically become more abstract, meaning that it may become more difficult to understand its underlying meaning in the context of the problem of interest. Therefore, it is advisable to provide a problem formulation that is the most formal (mathematical) possible, while also including an explanation of it using natural language.

2.2 Search-Based Optimization Algorithms

Optimization algorithms are algorithms that attempt to find optimal solutions to optimization problems. *Search-based optimization algorithms* are computational intelligence algorithms that conduct a search process in an attempt to find an optimal solution. They can be seen as high level frameworks that try to combine basic heuristics to more efficiently and effectively explore a search space, thus being frequently referred to as *meta-heuristics*. *Heuristics* are strategies that use readily accessible though loosely applicable information to control problem-solving processes [2]. Informally, heuristics can be seen as rules of thumb that may help to solve a problem more efficiently, though potentially loosing in optimality. The ability of search-based optimization algorithm to find good solutions in a reasonable amount of time for a variety of different problems has made them popular optimization algorithms.

Search-based optimization algorithms (or meta-heuristics) typically create one (or more) full initial candidate solutions to the problem, and then try to iteratively improve such candidate solution(s) based on some strategies (heuristics) to search for an optimal solution. For example, a search-based optimization algorithm to solve a routing problem might start with an initial solution which consists of a vector of cities [A, B, C, G]. The algorithm would consider this to be a full *candidate* solution to the problem, despite it being infeasible – it does not reach the destination city F. The algorithm would then modify this candidate solution over time, possibly by replacing, adding or removing cities, in an attempt to find better candidates solutions to the problem.

When multiple candidate solutions are created and maintained at each iteration, the algorithm is typically called a *population-based* algorithm, in contrast to algorithms that create and keep a *single individual* candidate solution in each iteration. Different search-based optimization algorithms may spend different amounts of resources on exploring or exploiting the search space in an attempt to find an optimum. *Exploration* consists in exploring different regions of the search space to find which of them are more promising (more likely to contain better solutions), whereas *exploitation* consists in exploiting a specific region in order to find the optimal solution. This part of the book will explain a number of different popular search-based optimization algorithms, giving an introduction to this field.

References

1. C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.
2. J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

Chapter 3
Local Search

Sara Ceschia, Luca Di Gaspero, Andrea Schaerf

Local Search is an algorithmic paradigm for combinatorial search and optimization, which has been shown to be very effective for many problems in the scientific literature. A large number of techniques based on local search have been proposed to successfully address a variety of practical problems.

Local search techniques belong to the larger class of the so-called *selective* methods that are based on the exploration of the search space composed by complete solutions. This is opposed to *constructive* methods that start from an empty solution and build the complete one in a step-by-step manner by iteratively adding a new piece to the partial solution constructed that far.

In addition, local search techniques are non-exact, as they do not guarantee to find the optimal solution. In the cases in which the optimal solution is found, there is no way to prove that it is indeed optimal, unless it has been established with exact techniques that it is not possible to reach a value lower than that (i.e., it is a proven *lower bound*).

Local search is based on the simple idea of *navigating* the search space by iteratively stepping from one solution to one of its *neighbors*. The neighborhood of a solution is the set of states that can be obtained by applying a "local change" to it, which consists in modifying the value assigned to a small number of variables representing the solution. The definition of the neighborhood relation is dependent on the specific problem under consideration, although there are some common patterns that can be applied to search spaces with similar structure. The mechanism upon which the neighbor is selected at each step is one of the main design choices that varies among different local search techniques. In any case, this selection mechanism relies upon the definition of the *cost function* f, which assesses the quality of each neighbor, and it is also problem dependent.

Even though it is difficult to assess precisely when local search should be used instead of other optimization methods, a general observation is that its

DPIA, University of Udine
{sara.ceschia,luca.digaspero,andrea.schaerf}@uniud.it

behavior depends on the landscape of the cost function. More precisely, if the landscape is relatively *smooth* with respect to local changes, it is more likely that local search techniques will be successful.

In this chapter, we discuss the key elements of the local search paradigm, which are independent of both the specific problem and the specific local search technique chosen to solve it.

We also introduce the basic local search techniques and discuss some improvements and variations of such basic methods. We refer to the following chapters and the book by Hoos and Stützle [1] for more complex local search techniques.

3.1 Local Search Elements

We first introduce one by one the key elements of local search. In detail, we will introduce the notions of *search space*, *neighborhood relation*, *cost function*, *initial solution selection*, *move selection and acceptance criterion*, and *stop criterion*.

3.1.1 Search Space

Given a search or optimization problem P and an instance I of P, we associate to it a search space S, with the following properties:

- each element $s \in S$ represents a solution of I, not necessarily feasible (i.e., it might violate some constraint of the problem);
- for search problems, at least one feasible solution of I is represented in S;
- for optimization problems, at least one optimal solution of I is represented in S.

If the previous requirements are met, we have a *valid representation* of the problem. We refer to an element $s \in S$ as a *state*. A state correspond to a solutions of the problem, but not all solutions are necessarily represented by some state in the search space.

Consider for example the classical n-queens problem, which consists in placing n queens in an $n \times n$ chessboard so that they do not attack each other neither vertically, nor horizontally, nor diagonally.

The direct representation of the problem would be with an $n \times n$ matrix of boolean-values, each representing the presence/absence of a queen in the corresponding square. However, the intuitive search spaces for local search already partition the chessboard in columns, making use of n integer-valued variables $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ such that the assignment $x_i = j$ corresponds to

place a queen in position (j, i) in the chessboard. Since this representation places only one queen in each column, it implicitly provides against vertical attacks.

There are two typical options for the definition of the search space for the n-queens problem:

- **Assignment**: any variable $x_1, \ldots, x_n$ can assume any value in the domain $\chi = \{1, \ldots, n\}$.
- **Permutation**: variables can assume only distinct values in the domain $\chi = \{1, \ldots, n\}$, resulting in a permutation of the numbers $1, \ldots, n$.

For example, $\mathbf{s_1} = [4, 6, 1, 6, 8, 4, 2, 1]$ is a state for the instance with $n = 8$ for **Assignment**, and $\mathbf{s_2} = [4, 1, 7, 6, 8, 3, 2, 5]$ is a state for both search spaces. These states are depicted in Figures 3.1a and 3.1b.

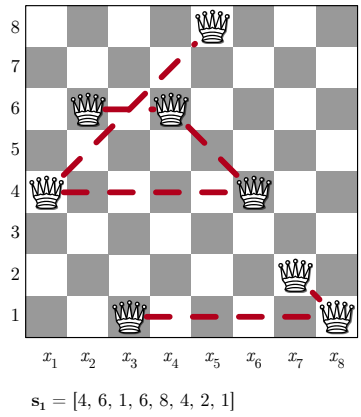


Fig. 3.1a A state for the n-queens problem in the **Assignment** search space.

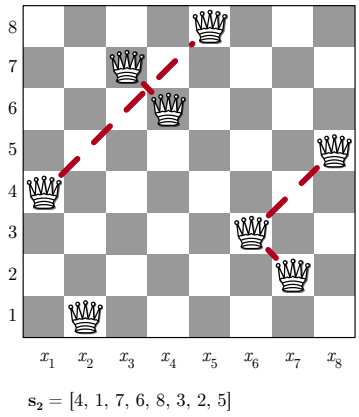


Fig. 3.1b A state for the n-queens problem in the **Permutation** search space (and also in the **Assignment** one).

In the **Assignment** space, horizontal and diagonal attacks have to be taken care by the cost function f. In the **Permutation** space, instead, since the search space is restricted so that two variables cannot have the same value, also horizontal attacks are prevented by construction. In this case, the only possible constraint violations come from the diagonal attacks. It is easy to see that both choices are valid representations in the sense defined above.

3.1.2 Neighborhood Relation

Given a problem P, an instance I and a search space S for it, we assign to each element $s \in S$ a set $\mathcal{N}(s) \subseteq S$ of neighboring states of s. The set $\mathcal{N}(s)$ is

called the *neighborhood* of s and each member $s' \in \mathcal{N}(s)$ is called a *neighbor* of s.

The set $\mathcal{N}(s)$ does not need to be listed explicitly, but usually it is implicitly defined by referring to a set of possible *moves*, which define transitions between states. A move m is defined by a small set of attributes that describe local modifications of some parts of s. We call $s \oplus m$ the state obtained by the application of move m to state s. The "locality" of moves is the key ingredient of local search, and it has also given the name of the whole search paradigm. Nevertheless, from the definition above there is no implication that there is "closeness" in some sense among neighbors, and in fact complex neighborhood definitions can be used as well.

The only condition that needs to be imposed on the neighborhood relation $\mathcal{N}$ is that the search space $\mathcal{S}$ is connected under $\mathcal{N}$. That is, every state s of $\mathcal{S}$ can be reached from any other state s' by a finite-length sequence of moves coming from $\mathcal{N}$.

Following the n-queens example, we consider two possible neighborhood relations:

- Change (C), for **Assignment** search space:
  The move $\mathsf{C}\langle q, v\rangle$ assigns value v to variable x_q.
  Precondition: $x_q \neq v$.

- Swap (S), for **Permutation** search space:
  The move $\mathsf{S}\langle q_1, q_2\rangle$ swaps the values assigned to variables x_{q_1} and x_{q_2}.
  Precondition: $q_1 \neq q_2$.

See Figures 3.2a and 3.2b for an example of each move, where the black arrows represent the transitions.

Notice that the Change neighborhood could not be applied to the **Permutation** search space, as the moves would lead outside the search space by duplicating some values. The Swap neighborhood, instead, can also be applied to the **Assignment** search space, but the space would not be connected under this neighborhood, as the number of repetitions of each value does not change upon the application of Swap moves, so that certain states (with different number of repetitions) are never reached.

Nowadays there is a large body of scientific literature that shows the effectiveness of different neighborhoods on the various problem domains (e.g., routing and scheduling [2]). Nonetheless, the definition of a suitable neighborhood for the specific novel problem remains a creative activity that the designers have to undertake using their own intuition and experience. The proof that the search space is connected under a specific neighborhood is also a designer's task, which however is relatively easy to do for most problems. For example, for the n-queens problem, for the two proposed combinations of search space and neighborhood, it is rather trivial to show that the search space is connected.

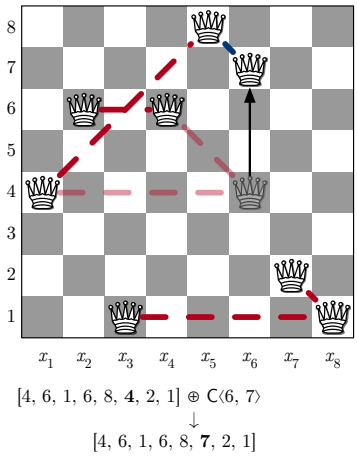


Fig. 3.2a An example of Change move.

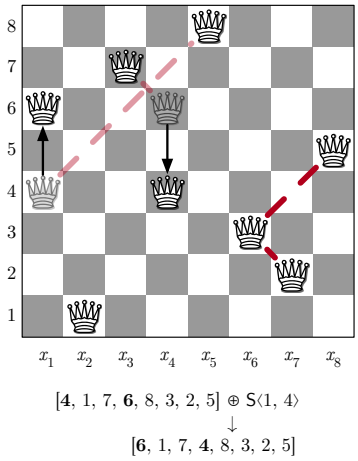


Fig. 3.2b An example of Swap move.

One of the key features of the design of the neighborhood relation is the possibility to compute efficiently the so-called *delta* function, which is the cost difference between two neighbor states. For example, for a Change move for the n-queens problem, only the attacks created (resp. removed) by the presence in the new (resp. old) position of the specific queen involved in the move needs to be evaluated. This is easily done in linear time (with respect to n). On the contrary, the full evaluation of the new state obtained by applying the move would have a quadratic computational cost.

3.1.3 Cost Function

The selection of the move to be performed at each step is based on the *cost function* f that associates to each element $s \in S$ a value $f(s)$ that assesses the quality of the solution. For the sake of simplicity, we assume that the value of f is integer-valued and non-negative, or in other words, that the co-domain of f is the set of natural numbers $\mathbb{N}$.

When applied to search problems, the function f normally counts the number of constraint violations, which is also known as *distance to feasibility*. For example, in the n-queens problem, the customary cost function counts the number of attacking pairs of queens. For the state $s_1 = [4, 6, 1, 6, 8, 4, 2, 1]$ represented in Figure 3.1a, we have $f(s_1) = 6$, as there are 6 pairs of queens at-

tacking each other $\{(x_1, x_5), (x_1, x_6), (x_2, x_4), (x_3, x_8), (x_4, x_6), (x_7, x_8)\}$ connected in the figures with red dotted lines.[1]

In the case of an optimization problem (which, without loss of generality, is assumed to be a minimization one), f typically merges together the distance to feasibility and the objective function f of the problem. Therefore, the cost function is typically defined as a weighted sum, with the highest weight assigned to the distance to feasibility, so as to give preference to feasibility over optimality. An alternative option is to do not assign weights, but rather consider the cost function as a pair, and perform the comparison of values in a hierarchical way, with priority assigned to feasibility.

For some optimization problems the search space can be defined in such a way that it represents only feasible solutions, and in those cases the cost function normally coincides with the objective function of the problem.

In some cases, the objective function is complemented by some auxiliary components which account for "invisible improvements" that incorporate some problem knowledge that is not explicitly considered in the objective function. These auxiliary components might be useful to guide the search toward promising regions of the search space. As an example of this concept, consider the bin-packing problem for which the objective is to minimize the number of bins and the selected neighborhood just moves one object from one bin to another. In this case, it could be useful to include in the cost function an auxiliary component that favors states in which bins are filled in an unbalanced way, rather then a situation in which objects are equally distributed in the bids. Indeed, such an auxiliary component could create a search trajectory composed of improving moves leading towards the removal of one bin.

Finally, it is also possible that the cost function is a surrogate of the real objective function in the case that the latter one is computationally expensive [3].

3.1.4 Initial Solution Selection

An essential component of any local search procedure is the selection of the initial solution. Typical choices are a random generation or the use of a greedy constructive heuristic. Greedy constructive heuristics add a new element of the solution at the time, taking what seems the best option available at the moment, without worrying whether it will bring the overall optimal solution. It is also possible to use any other search method to obtain the initial solution.

[1] Note that we count also the so-called *X-ray* attacks, i.e. the ones between non consecutive queens in a row (or diagonal) of more than two of them, which by the chess rules would not attack each other.

For example, for the **Permutation** space for the n-queens, the random state can be obtained by creating the identity permutation $[1, 2, \ldots, n]$ and then applying some shuffling procedure.

In order to decide whether to use just a random initial state or spend some effort to search for a greedy one, we should take into account different aspects. In fact, on the one hand, a better initial state would in general require less local search iterations in order to reach high quality solutions. This saves computational time, which could be used to perform more steps or more independent runs. On the other hand, though, it is also possible that the greedy heuristics biases the search toward some specific regions of the search space, so that it might become difficult to move away toward different areas.

In some cases, the greedy procedure might be necessary in order to obtain an initial feasible solution that a random procedure would not reach. This behavior would allows us to avoid to be forced to include the distance to feasibility in the cost function by finding a feasible initial solution, thus simplifying the cost function of the local search procedure.

3.1.5 Move Selection and Acceptable Criterion

At each search step, a single move is selected. The way this selection is performed characterizes the specific local search strategy, and we will discuss the different options in the following sections.

However, the selection of a specific move does not inevitably imply that the move is *accepted* and performed, so that the current state is changed. On the contrary, the move is normally subject to an acceptance condition, which is also dependent on the specific technique under consideration.

As a general rule, moves that improve (i.e., decrease) the cost function are accepted, but also worsening moves (i.e., increasing ones) could be accepted in specific situations, so as to let the search move away from a *local minimum*. A local minimum is a state s such that $f(s) \leq f(s')$ for all $s' \in \mathcal{N}(s)$. When this condition holds also with a strict inequality $<$, we call the state a *strict* local minimum. Escaping from local minima is the key issue of all local search techniques.

3.1.6 Stop Criterion

To end the presentation of the common key elements of local search we discuss the stop criterion, which determines when the search is over and the best solution found that far is returned.

Many criteria have been proposed in the literature, starting from the basic ones, typically based on the expiration of a given amount of time, to more complex strategies (see for example [4, Sect. 3.2]). The basic time-expiration strategies can either refer to the wall-clock time or to more abstract temporal measures, such as the number of iterations. The latter one has the advantage of not being machine dependent.

Other options might regard specific qualities of the solution reached or the so-called *stagnation* detection. That is, when no improvement is obtained for a given number of iterations, the search is stopped. This way, search trials that are exploring promising paths are let run longer than those that are stuck in regions without good solutions or where the search is trapped. It is also possible to combine different criteria, by stopping the search when the first of a set of conditions is met. Similarly to the initial solution selection, the stop criterion can be part of the specific technique, therefore we will further discuss it in the following sections.

Combining all the key elements presented so far, the pseudocode of a generic local search procedure is shown in Algorithm 1.

Algorithm 1 LocalSearch

Parameters: SearchSpace $\mathcal{S}$, Neighborhood $\mathcal{N}$, CostFunction f
Output: s_{best}

```
1: s ← InitialState(S)
2: s_best ← s
3: while not StopCriterion do
4:     m ← SelectMove(s, N)
5:     Δf ← f(s ⊕ m) − f(s)
6:     if AcceptMove(m, Δf) then
7:         s ← s ⊕ m
8:         if f(s) < f(s_best) then
9:             s_best ← s
10:        end if
11:    end if
12: end while
13: return s_best
```

The specific definitions of the subprocedures $InitialState$, $StopCriterion$, $SelectMove$, and $AcceptMove$ characterize the specific local search technique.

3.2 Basic Local Search Techniques

The simplest local search techniques are based on some form of the so-called *Iterative Descent* (also known as *Iterative Improvement* or *Hill Climbing*). The idea behind these techniques is to select at each step a move that im-

proves the value of the objective function or reduces the distance to feasibility, never performing worsening moves.

3.2.1 Steepest Descent

The most well-known form of Iterative Descent is the so-called *Steepest Descent* (SD) technique. At each iteration, SD selects, from the whole neighborhood $\mathcal{N}(s)$ of the current state s, the element $s' = s \oplus m$ which has the best value of the cost function f. The SD procedure accepts the move m only if it an improving move, i.e., it decreases the value of the cost function. Consequently, it naturally stops as soon as it reaches a local minimum. Therefore, assuming (as customary for combinatorial optimization) that the search space is finite and thus at least one local minimum exists, we don't need to define any other specific stop criterion.

The exhaustive exploration of the neighborhood is obtained by enumerating the moves in a fixed order. For example, the Swap neighborhood for the 5-queens problem, can be enumerated in the following lexicographic order: $\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 1,5\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 3,4\rangle, \langle 3,5\rangle, \langle 4,5\rangle\}$.

Notice that if there are two moves that lead to the same state, such as $\langle i,j\rangle$ and $\langle j,i\rangle$ in this case, only one of them should be included in order to prevent wasting time by visiting the same neighbor twice.

The enumeration of the neighborhood raises the issue of tie-breaking in case of multiple solutions whose value of the cost function is equally good. The typical strategy is to break ties in a uniform random way. That is, if there are k neighbors with the same minimal cost, then each of them might be selected with probability $1/k$.

3.2.2 First-Improving Descent

The exhaustive neighborhood exploration prescribed by the SD technique might be rather expensive from the computational point of view. To overcome this problem, the *First-Improving Descent* (FID) technique accepts a move and moves to a new neighbor as soon as an improving move is found. FID also stops when there are no improving moves so that a local minimum is detected.

In order to avoid the bias toward some specific attributes (e.g., the variables with low indexes in the above example), the exploration of the neighborhood should not proceed always in the same fixed order. The simplest way to avoid such a bias is to start from a random move and proceed onward in a circular way, by going from the last move to the first one. The procedure stops when the initial random one is encountered

again. For example, the exploration could proceed in the following order $\{\langle 2,4\rangle, \langle 2,5\rangle, \langle 3,4\rangle, \langle 3,5\rangle, \langle 4,5\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 1,5\rangle, \langle 2,3\rangle\}$ and stop as soon as an improving move has been found.

3.2.3 Random Descent

Another option to mitigate the computation burden of exhaustive neighborhood exploration consists in sampling the neighborhood by drawing random moves. This leads to the *Random Descent* (RD) technique that draws a (uniform) random move at each iteration and performs it only if it is improving.

Care should be taken to ensure that the draw is uniform. Drawing the attributes one at the time selecting among available values might bias the search toward moves that have a smaller number of possible values for the latest attributes. For example, for the Swap neighborhood, drawing a random value i between 1 and $n-1$ and then a random value between $i+1$ and n, would move the queens with higher indexes move often.

The presence of a local minimum cannot be detected by RD, consequently one of the stop criteria mentioned in Section 3.1.6 should be used in order to decide when to finish the search.

Notice that the exhaustive exploration of the neighborhood and the random selection can be seen as two extreme options in the trade-off between effectiveness and efficiency of the move selection strategy. The intermediate strategies explore exhaustively only a specific share of the neighborhood. To this regard, a classical choice is to select randomly a variable, but exploring all possible alternative values for that variable and selecting the best one.

3.2.4 Non-Ascent Techniques

All the techniques discussed above can be modified by changing the acceptance rule so that it accepts also states with equal cost value (the so-called *sideways* moves). This variant allows the search to move away from a local minimum, but is still trapped by a strict local minimum.

In this case, we call these methods Non-Ascent techniques, rather than Descent ones (Steepest, First-Improving, and Random). Non-Ascent techniques have the ability to navigate through *plateaux*, which are areas of the search space with equal cost, which are often present in the landscape of many problems.

The navigation through *plateaux* is often useful in reaching states from which the cost can be decreased again. On the other hand, in some cases such navigation might take a long time without producing any improvement.

3.3 Discussion

In this concluding section of the chapter we discuss some general issues of local search procedures.

3.3.1 Diversification vs. Intensification

The main issue of local search techniques is the way they deal with the local minima of the cost function. Even if the search procedure employs some specific mechanism for escaping them (see, e.g., the chapter on Simulated Annealing), a local minimum still behaves as a sort of *attractor*. Intuitively, when a trajectory moves away from a local minimum and steps through a solution "near" to it, even though it is not allowed to go back to the minimum itself, it still tends to move "forward" it (i.e., is attracted) instead of moving in an "opposite" direction.

For the above reason, the search procedure needs to use some form of *diversification* strategy that allows the search trajectories not only to escape a local minimum but to move "far" from it thus avoiding this sort of *chaotic trapping*.

On the other hand, for practical problems the landscape of the cost function usually has the property that the value is correlated in neighbors (and near) states. Therefore, once a good solution is found, it is reasonable to search in the proximity of it for a better one. For this reason, when a local minimum is reached the search should be in some way *intensified* around it.

In conclusion, the search algorithm should be able to balance two sometimes conflicting objectives; it should diversify and intensify by moving outside the attraction area of already visited local minima, but not too far from it.

3.3.2 Smart Neighborhood Exploration

Another issue in local search is the computational cost of the exploration of the neighborhood. Various ideas have been proposed in the literature in order to reduce the cost of the exploration, without penalizing the effectiveness of the search. We mention here just one idea, known as the *elite candidate list* (see, e.g., [5, Section 3.2]).

The mechanism is based on the idea of reusing the evaluations made in the previous explorations of the neighborhood. The intuition is that if a move m was good in a given solution s, very likely it will be still good in the neighbor $s' = s \oplus m'$ obtained by performing another move m'. Based on this intuition,

during each neighborhood evaluation, aside selecting the best move, we also collect a set of other promising moves (called elite moves).

In the subsequent iterations, the elite moves previously selected, but not executed yet, are re-evaluated and possibly executed without redoing the full exploration. The full exploration, and the corresponding collection of elite moves, is performed when all previous elite moves have been executed or discarded.

3.3.3 Strategic Oscillation

A strategy to overcome the risk of being trapped in a local minimum comes from the manipulation of the cost function.

The cost function of a typical local search procedure is composed by a weighted sum of a number of components, coming from the distance to feasibility and the objective function of the problem. Such weights reflect the relative importance of the various components of the objective function and the constraints.

Even though the weights reflect the "real" importance of the corresponding components, to improve effectiveness, their values might be modified during the search procedure so as to vary the landscape of the cost function. The temporary modification of the landscape of the cost function, known as *strategic oscillation* can help to escape from local minima.

The idea can be pushed further, by assigning an independent dynamic weight not to the cost components but to each single constraint. Such finer grain weighting could be useful for dealing with problems with strong asymmetries, i.e., problems in which the number of constraints involved can be different by a large factor from variable to variable.

3.3.4 Complex Neighborhoods

The last topic that we briefly discuss in this introductory chapter on local search regards the neighborhood relation. We know from the literature that for most problems there are various potential neighborhood relations available. Obviously, a state that is a local minimum for one neighborhood might not be a minimum for another neighborhood, so that the alternate use of different ones could help with the key issue of escaping local minima.

Even in cases of one single neighborhood, it is possible to consider chains of moves of length two or more that can be used as alternative relations, which would clearly have different local minima.

In general, there are different ways to employ more than one neighborhood relation during the search. They range from considering the set-union of all

neighborhoods (see, e.g., [6]), to the interleaving of distinct phases in which the basic neighborhoods are used, to the use of chains of basic moves as the basic neighborhood.

In all cases, the resulting neighborhood relation would be rather large and the exhaustive exploration problematic from the efficiency point of view. The discussion on how to explore a large neighborhood is complex and it is outside the scope of this chapter (see [7]).

3.4 Exercises

Exercise 1 Consider the classical *graph coloring* problem, in which a given indirected graph has to be colored with a fixed number of colors avoiding that two adjacent nodes have the same color. Define a valid search space, a suitable neighborhood relation, and the cost function for this problem.

Exercise 2 Consider the classical *permutation flow-shop* problem, in which the order of a set of *jobs* and the start and end times of their processing on a predetermined sequence of machines has to be decided. The objective is to minimize the *makespan*, that is the latest completion time of a job on the last machine. Define a valid search space, one or more suitable neighborhood relations, and the cost function for this problem.

Exercise 3 Consider the problem of Exercise 2, with in addition the presence of *strict* due dates for the jobs, that is each job must be completely processed within its due date. How do you modify the cost function? Conversely, if the due dates are soft and the problem requires to minimize the total *tardiness*, i.e., the summation of the lateness of each job, how is the cost function defined in this case?

References

1. Holger H. Hoos and Thomas Stützle. *Stochastic Local Search – Foundations and Applications*. Morgan Kaufmann Publishers, San Francisco, CA (USA), 2005.
2. Toshihide Ibaraki, Shinji Imahori, Mikio Kubo, Tomoyasu Masuda, Takeaki Uno, and Mutsunori Yagiura. Effective local search algorithms for routing and scheduling problems with general time-window constraints. *Transportation science*, 39(2):206–232, 2005.
3. Slawomir Koziel, David Echeverría Ciaurri, and Leifur Leifsson. Surrogate-based methods. In Slawomir Koziel and Xin-She Yang, editors, *Computational Optimization, Methods and Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
4. Alberto Franzin and Thomas Stützle. Revisiting simulated annealing: A component-based analysis. *Computers and Operations Research*, 104:191–206, 2019.
5. Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

6. Ruggero Bellio, Sara Ceschia, Luca Di Gaspero, and Andrea Schaerf. Two-stage multi-neighborhood simulated annealing for uncapacitated examination timetabling. *Computers & Operations Research*, 132:105300, 2021.
7. R. Ahuja, Ö Ergun, J. Orlin, and A. Punnen. A survey of very-large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

Chapter 4
Simulated Annealing

Alberto Franzin, Thomas Stützle

The origins of simulated annealing lie in the work of Metropolis *et al.* in statistical physics [1], who proposed an algorithm to simulate the displacement of particles in a solid body. A random displacement is effectively performed ("accepted") if it lowers the total energy of the system, while it is performed with probability $\exp(-\Delta E/k_B T)$ if the energy variation ΔE is positive. In the Metropolis formula, k_B is the Boltzmann constant and T is the temperature at which the displacement is evaluated. For a temperature value T, the higher ΔE, the lower the chance the displacement effectively happens. Likewise, if the given ΔE is held constant but positive, the higher the temperature, the higher the probability of the displacement.

In one of the first instances of algorithmic development inspired by natural phenomena, Kirkpatrick, Gelatt and Vecchi [2], and independently Černy [3], thought to view a combinatorial optimization problem as a solid, whose states are the feasible solutions of the problem. The objective function value of a particular solution is related to the level of energy of the system in a particular configuration. The system is first heated and then cooled little by little until it "freezes". Out of the metaphor, the algorithm starts from an initial solution and iteratively evaluates candidate ones. The temperature used to evaluate an exchange starts at a high value, and is progressively lowered following a geometric trend. For temperature values low enough, the probability of accepting worsening moves becomes effectively zero, and the algorithm behaves like an iterative improvement.

The function originally proposed to determine the acceptance or the rejection of a move is the so-called *Metropolis criterion* [1], where a solution of equal or better quality is accepted always. If, however, the new solution $\mathbf{x}'$ is worse than the current solution $\mathbf{x}$, it is accepted with a probability $\exp(-(f(\mathbf{x}') - f(\mathbf{x}))/T)$ for minimization or with a probability $\exp(f(\mathbf{x}') - f(\mathbf{x})/T)$ for maximization problems, where $f(\cdot)$ is the objective

Université Libre de Bruxelles

function. Here, we always will assume minimization problems; it is straightforward to adapt the algorithm for a maximization problem. The probability of accepting worsening moves depends on the relative worsening in terms of solution quality from the incumbent one and on the scalar parameter T, called *temperature*. For a proposed worsening move, high values of the temperature parameter provide a higher chance of accepting it than lower values. High temperature values therefore promote a stronger exploration behaviour, while low values are associated to a strong exploitation. Simulated annealing can thus be considered as a local search with a diversification mechanism governed by the temperature parameter.

The choice of the value for the temperature parameter is crucial in obtaining good results. The original formulation, the conventions and the folklore deriving from the annealing metaphor, but also the theoretical works on simulated annealing, make use of a sequence of values. In general, the temperature value is set at a "high" initial value, and progressively decreased to a final "low" value, to make the algorithm transition from a highly exploration behaviour to a converging, exploitative one. Optionally, the temperature value is raised again or restored to its initial value, to promote a cycle of diversification and intensification phases.

Immediately after its introduction, simulated annealing attracted the attention of several researchers, thanks to its simple formulation and the quality of the results obtained in several works. It became therefore one of the most popular algorithms for combinatorial optimization problems, and it has been used in thousands of works. Along with its use, researchers became interested in modeling its behaviour from a theoretical perspective. A series of works, mostly in the 80s' and early 90s', focused on conditions for the cooling scheme to obtain the optimal solution for a given problem and instance, mostly under unrealistic conditions such as an infinite number of evaluations [4, 5, 6, 7]. Yet, by simply enumerating the search space one could guarantee to find the optimal solution even in finite time but, in the end, very large time which is usually infeasible. Thus, a more important question would refer to the convergence speed of simulated annealing, that is, how quickly good solutions are found; yet, results addressing this question are rare. Also, practical annealing schedules decrease the temperature much faster than what is implied by the theoretical results and therefore the convergence proofs do not apply in this case.

In this chapter we summarize research on simulated annealing and its behaviour. We show how implementing a simulated annealing algorithm can be seen as a configuration task, and makes use of efficient artificial intelligence tools to automatically generate high-performing algorithms for different scenarios. We then explore quickly the results and the algorithms we obtain and then conclude the chapter.

4.1 A simulated annealing algorithm

4.1.1 A basic simulated annealing algorithm

Implementing a basic simulated annealing algorithm is quite fast. First, one needs a starting solution, which can be a uniformly random solution in the simplest case, or one computed using a simple heuristic like a constructive one. Second, one needs a neighbourhood, which is an essential ingredient. Sometimes the neighbourhood can be more complicated as for some problems one needs to have various neighbourhoods and decide how to schedule them. Then one has to choose a simulated annealing acceptance criterion. As the simplest one, use the Metropolis criterion, which we have detailed in the introduction of this chapter [1, 8, 9]. As a reminder, an improving or equal move is always accepted and a worsening one is accepted with a probability $\exp\left(-(f(\mathbf{x}') - f(\mathbf{x}))/T\right)$. The rest is embedding the annealing criterion in an *annealing schedule*, which is also called *cooling schedule*. It is defined by an initial temperature T_0, a scheme saying how the new temperature is obtained from the previous one, the number of search steps to be performed at each temperature, and a termination condition. A common choice is in simulated annealing to use a geometric cooling, which is that $T_{i+1} = \alpha T_i$, where α is a parameter with $0 < \alpha < 1$. This operation, called annealing step, is usually done after a number of search steps (that is, candidate solution evaluations) performed at the same temperature, often for a number of evaluations that is a multiple of the neighbourhood size. Finally the termination criterion is often based on the acceptance ratio, that is, the ratio of proposed steps versus the accepted steps.

4.1.2 A component-based overview of simulated annealing

To use a full-fledged simulated annealing one has to set still the control parameters, such as the initial temperature value, the temperature update factor, or the number of moves to be evaluated at the same temperature. We have now a somewhat reasonable algorithm and in the meantime, with the thousands of papers, much more involved simulated annealing algorithms have been proposed. So in [10], we identified a total of seven components, whose function and connection define what a simulated annealing is, and how it works. Two other components are necessary to elaborate information for the specific problem considered, and are not unique to simulated annealing. Problem-specific components can be considered as inputs to simulated annealing. The outline is given in Algorithm 2, and the components are subsequently described.

Algorithm 2 Component-based formulation of SA. The components we have identified for our analysis are written in SMALLCAPS.

Parameters: a problem instance $\mathcal{I}$, a NEIGHBOURHOOD $\mathcal{N}$ for the solutions, an INITIAL SOLUTION $\mathbf{x}_0$, control parameters
Output: the best solution $\mathbf{x}^*$ found during the search.

```
1: best solution x* := incumbent solution x̂ := x_0
2: i := 0
3: Initialize temperature T_0 according to INITIAL TEMPERATURE
4: while STOPPING CRITERION is not met do
5:    choose a solution x_{i+1} in the NEIGHBOURHOOD of x̂ according to NEIGHBOURHOOD
      EXPLORATION criterion
6:    if x_{i+1} meets ACCEPTANCE CRITERION then
7:       x̂ := x_{i+1}
8:       if x̂ improves over x* then
9:          x* := x̂
10:      end if
11:   end if
12:   if TEMPERATURE LENGTH is met then
13:      update temperature according to COOLING SCHEME
14:      reset temperature according to TEMPERATURE RESTART scheme
15:   end if
16:   i := i + 1
17: end while
18: return x*
```

INITIAL TEMPERATURE. The initial value of the temperature parameter, and, usually, the maximum value the temperature will take. It also defines the maximum amount of diversification, that is, how likely the algorithm is to accept worsening moves, and it is therefore best computed starting from instance characteristics. For example, the initial temperature can be set proportionally to the initial solution value, or computed according to some search space statistics such as the maximum gap or the average gap between consecutive moves computed during a preliminary random walk.

INITIAL SOLUTION (problem-specific). Simulated annealing starts from an initial solution that has to be provided by a problem-specific component, for example, using a constructive heuristic, another local search, or even a random solution. Anyway, if one starts from a problem-specific, good candidate solution one should use a low temperature start, since otherwise the initial solutions gets destroyed.

NEIGHBOURHOOD (problem-specific). The neighbourhood function generates the solutions that are at distance of one move from the incumbent one. This is also the component that deals with constraints when necessary, repairing infeasible solutions or ensuring that the candidate solutions generated are feasible. In the simplified environment this is one neighbourhood, but it may be that for other problems various neighbourhoods are important.

NEIGHBOURHOOD EXPLORATION. This component is devoted to selecting one candidate solution to evaluate for acceptance, in the neighbourhood of

the incumbent one. The traditional simulated annealing uses the random selection of a neighbour solution, but alternative schemes are possible, such as a sequential evaluation following an enumeration of the solution in the neighbourhood, or even mini-local searches [11, 12].

Acceptance Criterion. The function that determines whether the solution selected by the Neighbourhood Exploration should be accepted and replace the incumbent. The traditional and most popular criterion is the Metropolis condition described at the beginning of this chapter, but several other criteria have been proposed. Notably, the Threshold Acceptance is a deterministic variant of simulated annealing that always accepts candidate moves that worsen the solution quality of an amount bounded by the temperature value [13, 14]. The Late Acceptance Hill Climbing can also be considered as an acceptance criterion that can be fit in the simulated annealing structure [15]. This latter scheme maintains a history of accepted moves, and always accepts a candidate solution that is improving over either the incumbent one, or the incumbent solution of κ iterations before.

Cooling Scheme. The function that controls the decrease of the temperature parameter. The original simulated annealing papers use a geometric criterion, that decreases the temperature value by a multiplicative factor $\alpha \in (0, 1)$, i.e. $T_{i+1} = \alpha T_i$, with α constant throughout the whole search [2, 3]. In accordance with the annealing metaphor and the idea of transitioning from exploration to exploitation, cooling schemes generally yield monotonically decreasing sequences of temperature values. Some authors have, however, deviated from the annealing behaviour and proposed schemes that always keep the same temperature value, or have the temperature value "fluctuate", allowing for phases of both controlled increase and decrease of the temperature parameter [16]. This fluctuation is intended to exhibit a behaviour similar to a fixed temperature value, while at the same time promoting alternating phases of limited exploration and exploitation, thus bridging the two opposite SA behaviours.

Temperature Length. This component controls the amount of candidate solutions that will be evaluated using the same temperature value; in other words, it determines when the Cooling Scheme should be used. The simplest options are constant values, either absolute or proportional to some problem characteristic such as instance or neighbourhood size [17, 18]. However, adaptive schemes that make the decision on when to update the temperature based on the outcome of the search (for example, a certain amount of accepted moves [19]) can provide a more fine-grained control of the exploration/exploitation trade-off.

Temperature Restart. This component determines if and when the temperature value should be reset to its original value, or set to another higher value (reheat). Its purpose is to promote a new phase of exploration once the temperature has decreased enough to void its diversification potential. As for the Temperature Length there are many possibilities, both constant and adaptive [20, 21].

Stopping Criterion. The last algorithmic component determines when the search terminates. This can be set according to fixed conditions such as a given runtime or amount of moves evaluated, or adaptive conditions such as a too low acceptance rate [8, 9].

4.1.3 Composing efficient simulated annealing algorithms

To implement a new or existing simulated annealing, the developer needs to face several choices. First of all, it is necessary to define the choices that are related to the specific problem considered, that is, an initial solution from where to start the search, and the neighbourhood function used to generate new candidate solutions. Then the search components, that determine how to select a new candidate solution, whether to accept it, and how to update the control parameters have to be implemented.

Since the structure of SA is fixed, designing a simulated annealing algorithm means to choose which components and which numerical parameters to use. A good starting point to determine which choices to make is the existing body of literature. Even when limiting ourselves to the algorithm-specific components, we can find plenty of options for each one of them.

In [22] we collected a total of 66 different options for all the components, reported in Table 4.1, which are enough to assemble millions of valid simulated annealing algorithms. Many of those components have their own set of numerical parameters to choose [23, 24].

The traditional, manual implementation of an algorithm is effectively an exploration of the space of possible valid algorithms. Finding the one that performs the best over the entire application space (i.e. the whole test set) is clearly a daunting task. Developers therefore often revert on results from the literature and on established practices and conventions. The first reasonable algorithm, either assembled or found in the literature, is then fine-tuned on a limited set of instances.

Aside from being tedious, this process has several drawbacks. First, the combination of components and parameters raises a complex interaction that is often unexpected to the user. The developer is then forced to evaluate a restricted set of options on a limited set of instances, thus prioritizing the algorithmic solutions already explored in past applications and in the literature. The bias that follows from this process likely results in sub-optimal algorithms, whose behaviour and results are unexplained.

In this chapter we demonstrate how the use of powerful algorithm engineering tools is beneficial in automatically designing simulated annealing algorithms that can outperform manually designed ones. We follow the Programming by Optimization philosophy that envisages increasing levels of automation of the development process [25]. The implementation of an algorithms moves away from a purely manual task, to one that relies on auto-

Table 4.1: List of options we implemented for the algorithmic components of simulated annealing. Several of these components include additional numerical parameters. For a more formal description of each component and their implementation we refer to [10].

Initial Temperature	Fixed value; proportional to initial solution value; proportional to maximum gap in random walk; proportional to average gap in random walk; Connolly initial temperature scheme; Misevicius initial temperature scheme; simplified Misevicius initial temperature scheme.
Neighbourhood Exploration	Random; sequential; Ishibuchi-Misaki-Tanaka 1; Ishibuchi-Misaki-Tanaka 2.
Acceptance Criterion	Metropolis; bounded Metropolis; precomputed Metropolis; Generalized simulated annealing; geometric acceptance; Threshold acceptance; Great Deluge; Record-to-Record; Late Acceptance Hill Climbing; Hill Climbing.
Cooling Scheme	Geometric 1; geometric 2; Lundy-Mees 1; Lundy-Mees 2; Q8-7; quadratic; arithmetic; no cooling; temperature band.
Temperature Length	Fixed; # moves proportional to problem size; proportional to neighbourhood size; # of accepted moves; bounded (# accepted moves, max # moves); arithmetic increase; geometric increase; logarithmic increase; exponential increase.
Temperature Restart	Never; Restart or reheat based on: # moves; minimum temperature value; % of initial temperature value; global acceptance rate; acceptance rate in the last moves; no recently accepted moves.
Stopping Criterion	runtime; # moves; minimum temperature; # cooling steps; # temperature restarts; # moves with no accepted solution; global acceptance rate; most recent acceptance rate; no new recent best solution.

matic tools to improve existing algorithms, towards an automatic design of new algorithms.

This task can be framed as a machine learning one [26]. In a nutshell, our goal is to find the parameter configuration that obtains the best results on a *test set* of instances, that ideally represent a realistic practical application. In absence of an explicitely structured knowledge on the relationship between configurations and instance characteristics in terms of results, we employ a *training set* of instances, different from the test set but belonging to the same distribution. On this training set we select the configuration with the best results overall. For more detailed instructions we refer to the existing literature, e.g. [25, 27, 28, 29] in general, and [10] for SA specifically.

4.2 Experiments

4.2.1 Materials and methods

We consider the Quadratic Assignment Problem (QAP), one of the classic NP-hard problems where we want to find the assignment of n facilities to n locations with the minimal cost [30]. A QAP instance is defined by two matrices, one with the distances $D_{k,l}$ between two locations k and l, and one with the flows $F_{i,j}$ between two facilities i and j. We represent a solution as a permutation π, whose elements $\pi(i)$ contain the location of facility i. The objective function is

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} F_{ij} D_{\pi(i)\pi(j)}. \tag{4.1}$$

We use two groups of different QAP instances. In the first one the distance matrices define a structure in the landscape, and we refer to it as the class of *structured* instances. In the second one, instead, the matrices are generated uniformly at random, and we call this the *random* instances class. Each class contains 100 instances of sizes 60, 80 and 100, equally partitioned into a training set and a test set. The training set is used to automatically configure and design the algorithms using irace, and the test set is used to evaluate the configurations obtained, whose results are reported in the rest of this section. Every simulated annealing execution runs for ten seconds. We use the EMILI framework, where we implemented 66 components and the relative numerical parameters, for a total of 100 configurable parameters [29]. We use a budget of 2000 experiments to configure the numerical parameters of two existing simulated annealing algorithms, and we automatically design simulated annealing algorithms using 10K, 20K, 40K and 80K experiments, where one experiment is a SA run. The experiments are done on a machine with two Intel Xeon E5-2680v3 CPUs at 2.5GHz with 16MB cache, with 2.4GB of RAM available for each experiment.

The algorithm configurator irace implements the iterated racing algorithm in an easy-to-use `R` package [24, 31, 32]. The basic idea is to spend computational budget only on configurations that are worth evaluating, rather than waste it on poor ones. To configure an algorithm it requires only the definition of the parameters, a training set of instances, and a script that executes the algorithm with a list of command-line parameters (provided by irace as input to the script) and returns a measure of the quality of the configuration observed on the instance. Such metric defines the configuration task as an optimization problem, and commonly used ones include the solution quality found by heuristic algorithms, or the runtime needed to find the optimal solution. irace starts from a set of random configurations, and begins to evaluate all of them on the training instances, sequentially. The configurations are ranked for each instance, based on the results reported. After a few in-

stances, irace begins to discard the configurations that perform statistically worse than the best ones, and proceeds to evaluate only the surviving ones. This process of configuration evaluation and statistical test, called *race*, is iterated until either the budget for evaluations is exhausted, or a minimum number of surviving configuration remains. At this point, new configurations are sampled, with parameter values around the surviving configurations, and a new race takes place. Race after race, the new configurations get sampled increasingly closer to the last best ones. When the total evaluation budget is terminated, irace returns to the user the best configurations found.

EMILI is a `C++` algorithmic framework for the development of modular stochastic local search algorithms [29]. Algorithms are instantiated at runtime by selecting the desired combination of command-line parameters, that determine which components and numerical parameters are selected, and how they are combined. The modular structure of EMILI makes it possible to design algorithms bottom-up, so, in principle, of any shape. However, its flexibility can of course be exploited to design algorithms of a given structure, as we do in this chapter by requiring the components to be placed in their proper place in the template of simulated annealing in Algorithm 2.

4.2.2 Simulated annealing algorithms for the QAP

We start by reproducing two of the many simulated annealing algorithms proposed for QAP. These algorithms are called `BR1` and `Q8-7` [33, 11]. The only common algorithmic component between the two algorithms is the Metropolis acceptance criterion. `BR1` starts from an initial temperature computed as the maximum gap observed between consecutive solutions in a random walk [33]. It uses the random exploration of the neighbourhood to select a move to evaluate. The temperature is updated with a geometric cooling scheme with coefficient 0.5 every $2 \times n$ proposed moves, where n is the problem size, and restarted to its original value after $|N(\mathbf{x})|$ moves, that is, the size of the neighbourhood. `BR1` therefore performs only a few but substantial temperature updates. In its original formulation, `BR1` terminates its search after 10 temperature restarts.

The `Q8-7` algorithm from [11] uses a custom mechanism to define the initial temperature, relating it to the final one. This algorithm uses a sequential exploration of the neighbourhood to select candidate moves, imposing an order on the solutions in the neighbourhood and following it. One particular characteristic of this algorithm is the `Q8-7` cooling scheme, an adaptation of the Lundy-Mees cooling scheme [5]. After the cooling phase, once m consecutive moves have been rejected, this scheme restores the temperature to the value at which the current best solution was found, forces the acceptance of the current proposed move, and stops the temperature update, keeping the same value until the end of the search. The m parameter is set proportionally

to the total number of moves evaluated by the algorithm. During the initial cooling phase of the `Q8-7`, the temperature is lowered after every move evaluated, and there is no temperature restart. The algorithm as reimplemented from the original paper terminates after $50 \times |N(\mathbf{x})|$ moves.

Both algorithms start from a randomly generated permutation, and use the 2-exchange neighbourhood

$$\mathcal{N}(\pi) = \{\pi' \mid \pi'(j) = \pi(h) \wedge \pi'(h) = \pi(j) \wedge \forall l \notin \{j, h\} \ : \pi'(l) = \pi(l)\} \quad (4.2)$$

that swaps two elements in positions i and j in a solution π. In total, `BR1` has five tunable numerical parameters, while `Q8-7` has six; as we will not tune the termination condition, this number reduces to respectively four and five. The pseudocode for `BR1` and `Q8-7` is reported, respectively, in Algorithms 3 and 4.

Algorithm 3 Component-based formulation of `BR1` with its original parameters. The components are highlighted in *italic*. The tuned numerical parameters we obtained are: $\theta' = (k = 8.979, \beta = 51, \alpha = 0.9163, \gamma = 83)$ for the structured instances and $\theta' = (k = 6.3173, \beta = 94, \alpha = 0.9206, \gamma = 79)$ for the random instances, and the termination is replaced with a time-based one (10 seconds).

Parameters: a problem instance $\mathcal{I}$, the 2-*exchange neighbourhood*, *a random permutation* $\mathbf{x}_0$, control parameters $\theta = (k = 1, t = 10, \beta = 2, \alpha = 0.5, \gamma = 1)$
Output: the best solution $\mathbf{x}^*$ found during the search.

best solution $\mathbf{x}^*$:= incumbent solution $\hat{\mathbf{x}} := \mathbf{x}_0$
$i := 0$
Initialize temperature T_0 as *the max gap between consecutive solutions in a random walk of length* $l = 10^4$*, with a scaling factor* k
while *less than* t *temperature restarts have happened* **do**
    choose a solution $\mathbf{x}_{i+1}$ in the 2-*exchange neighbourhood* of $\hat{\mathbf{x}}$ *at random*
    if $\mathbf{x}_{i+1}$ meets *Metropolis criterion* **then**
        $\hat{\mathbf{x}} := \mathbf{x}_{i+1}$
        if $\hat{\mathbf{x}}$ improves over $\mathbf{x}^*$ **then**
            $\mathbf{x}^* := \hat{\mathbf{x}}$
        end if
    end if
    if β *times the instance size number of moves have been evaluated* **then**
        update temperature according to *geometric cooling with factor* α
        reset temperature *to initial value* γ *times the neighbourhood size*
    end if
    $i := i + 1$
end while
return $\mathbf{x}^*$

Algorithm 4 Component-based formulation of `Q8-7` with its original parameters. The components are highlighted in *italic*. The tuned numerical parameters we obtained are: $\theta' = (k = 6.7411, \alpha = 0.9114, \beta = 0.2183, m = 600676, j = 97)$ for the structured instances and $\theta' = (k = 5.5683, \alpha = 0.8597, \beta = 0.04, m = 234615, j = 76)$ for the random instances, and the termination is replaced with a time-based one (10 seconds).

Parameters: a problem instance $\mathcal{I}$, the 2-*exchange neighbourhood*, *a random permutation* $\mathbf{x}_0$, control parameters $\theta = (k = 1, t = 10, \alpha = 1, \beta = 1, m = |N(s)|, j = 1)$
Output: the best solution $\mathbf{x}^*$ found during the search.

1: best solution $\mathbf{x}^*$:= incumbent solution $\hat{\mathbf{x}} := \mathbf{x}_0$
2: $i := 0$
3: Initialize temperature T_0 according to *custom formula [11], with scaling factor k*
4: **while** *less than* $t \times |N(\mathbf{x})|$ *moves* have been evaluated **do**
5: choose a solution $\mathbf{x}_{i+1}$ in the 2-*exchange neighbourhood* of $\hat{\mathbf{x}}$ using a *sequential neighbourhood exploration*
6: **if** $\mathbf{x}_{i+1}$ meets *Metropolis criterion* **then**
7: $\hat{\mathbf{x}} := \mathbf{x}_{i+1}$
8: **if** $\hat{\mathbf{x}}$ improves over $\mathbf{x}^*$ **then**
9: $\mathbf{x}^* := \hat{\mathbf{x}}$
10: **end if**
11: **end if**
12: **if** *j moves have been evaluated* **then**
13: update temperature according to *Q8-7 cooling scheme with parameters α, β, m*
14: *never* reset temperature
15: **end if**
16: $i := i + 1$
17: **end while**
18: return $\mathbf{x}^*$

4.2.2.1 Reinstantiating and improving existing algorithms

We start with three types of experiments. In the first one we run the algorithms using their original parameter settings, including the termination (thus resulting in different running times); in the second one we take the original parameters but force each algorithm to run for 10 seconds; and a third one where we tune the parameters with irace, forcing an algorithmic runtime of 10 seconds. The results are reported in Figure 4.1.

In the first experiment the results obtained by `BR1` and `Q8-7` are similar, with a relative percent deviation (RPD) around 4%. This is because the algorithms are fairly old and most likely they have been manually fine-tuned on small instances with little computational power available. If we let them run for a longer time, which can be done by simply replacing the termination condition, this clearly favours `BR1` that improves its results. The same modification has however a surprising effect on `Q8-7`, which significantly worsens its results. This can be explained by considering how the Q8-7 cooling scheme is defined in the original paper. When this component is re-implemented exactly as described, one of the parameters is computed relatively to the expected

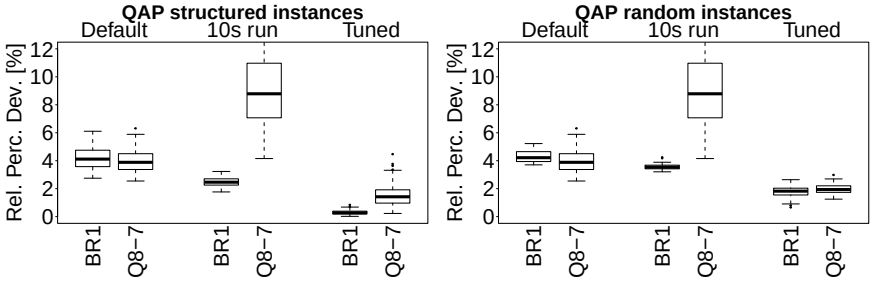


Fig. 4.1: Percentage deviation from the best known solutions obtained by the two simulated annealing algorithms on our structured and random QAP test set in their default settings, when let run for ten seconds, and when tuned using irace on a separate training set.

number of total move evaluations of the search. This value defines therefore a delicate balance, that collapses if this value increases dramatically, as it is the case in our experiments. The resulting value makes therefore the temperature update too slow, and the algorithm is unable to remain on good converging paths as it has a very high probability of accepting worsening moves for a too large part of the search. We then tune the numerical parameters of the algorithms using irace, paying attention to modifying the Q8-7 cooling scheme such that its parameters can be independently configured. In this case, on both instance classes both algorithms significantly improve their performance. On the structured instances, however, `BR1` with an RPD of around 0.2% clearly outperforms `Q8-7`, which finds solutions on average worse than 1% over the best known ones. On the random instances, instead, both algorithms obtain solutions on average just below 1% of RPD, with `BR1` marginally better than `Q8-7`.

4.2.2.2 Generating new algorithms

Finally, we exploit the whole set of options identified in the literature for the different simulated annealing components, using irace to automatically select the best combination and thus design new algorithms from scratch. We report in Figure 4.2 the results obtained by the algorithms that result from automated design tasks with 10K, 20K, 40K and 80K experiments. The algorithms obtained with the highest budget are reported in Algorithms 5 and 6 for structured and random instances, respectively.

On the structured instances the results are slightly better than those obtained by `BR1`, with the exception of the configuration found with 20K experiments. This is a somewhat easy scenario, and it takes relatively little effort to find good solutions. The random instances scenario is instead more challeng-

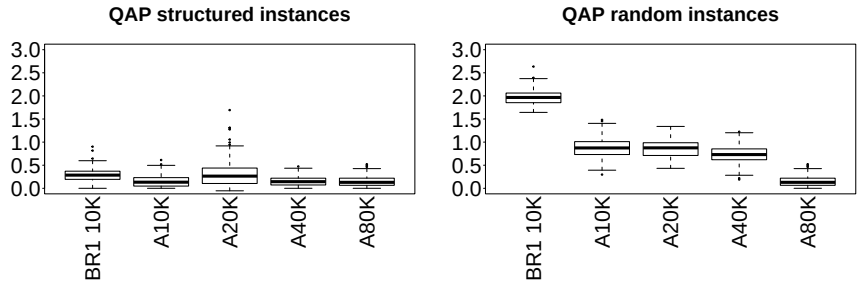


Fig. 4.2: Percentage deviation from the best known solutions obtained when automatically designing simulated annealing algorithms from scratch using 10K, 20K, 40K and 80K experiments on our structured and random QAP test set, compared with `BR1` tuned with 10K experiments.

Algorithm 5 Component-based formulation of the SA automatically designed for the structured instances with a tuning budget of 80 thousands experiments. The components are highlighted in *italic*.

Parameters: a problem instance $\mathcal{I}$, the 2-*exchange neighbourhood*, *a random permutation* $\mathbf{x}_0$, control parameters $\theta = (p, 0.2249, k = 0.6969, \beta = 4645.392, \alpha = 0.6921, \gamma = 32)$
Output: the best solution $\mathbf{x}^*$ found during the search.

1: best solution $\mathbf{x}^*$:= incumbent solution $\hat{\mathbf{x}} := \mathbf{x}_0$
2: $i := 0$
3: Initialize temperature T_0 as *the value that gives an expected initial acceptance probability* p *of worsening moves in a random walk of length* $l = 10^4$*, with a scaling factor* k
4: **while** *less than* 10 *seconds of runtime* **do**
5: choose a solution $\mathbf{x}_{i+1}$ in the 2-*exchange neighbourhood* of $\hat{\mathbf{x}}$ *at random*
6: **if** $\mathbf{x}_{i+1}$ meets *Metropolis criterion* **then**
7: $\hat{\mathbf{x}} := \mathbf{x}_{i+1}$
8: **if** $\hat{\mathbf{x}}$ improves over $\mathbf{x}^*$ **then**
9: $\mathbf{x}^* := \hat{\mathbf{x}}$
10: **end if**
11: **end if**
12: **if** *the temperature value drops below* β **then**
13: update temperature according to *geometric cooling with factor* α
14: reset temperature *to initial value* γ *times the neighbourhood size*
15: **end if**
16: $i := i + 1$
17: **end while**
18: return $\mathbf{x}^*$

ing, and it takes more experiments to find a suitably good configuration. In fact, while the tuning with 10 thousand experiments improves a lot over `BR1`, it takes 40 thousand experiments to marginally improve the results. Using 80 thousand experiments, however, it is possible to find configurations that find solution qualities comparable to the structured instances case.

Algorithm 6 Component-based formulation of the SA automatically designed for the random instances with a tuning budget of 80 thousands experiments. The components are highlighted in *italic*.

Parameters: a problem instance $\mathcal{I}$, the 2-*exchange neighbourhood*, *a random permutation* $\mathbf{x}_0$, control parameters $\theta = (k = 0.5438, \delta = 632, \alpha = 0.0.5927, \beta = 0.65, \gamma = 1208, r = 71822, s = 0.0828)$
Output: the best solution $\mathbf{x}^*$ found during the search.

best solution $\mathbf{x}^*$:= incumbent solution $\hat{\mathbf{x}} := \mathbf{x}_0$
$i := 0$
Initialize temperature T_0 as *the average gap between consecutive solutions in a random walk of length* $l = 10^4$*, with a scaling factor* k
while *less than* 10 *seconds of runtime* **do**
    choose a solution $\mathbf{x}_{i+1}$ in the 2-*exchange neighbourhood* of $\hat{\mathbf{x}}$ *as the best one among* δ *randomly chosen ones*
    if $\mathbf{x}_{i+1}$ meets *Metropolis criterion* **then**
        $\hat{\mathbf{x}} := \mathbf{x}_{i+1}$
        if $\hat{\mathbf{x}}$ improves over $\mathbf{x}^*$ **then**
            $\mathbf{x}^* := \hat{\mathbf{x}}$
        end if
    end if
    if *exponentially increasing temperature length with parameters* r, s is met **then**
        update temperature according to *geometric cooling variant with factors* α, β
        reset temperature *to the one of the best solution found if no move accepted in the last* γ *ones*
    end if
    $i := i + 1$
end while
return $\mathbf{x}^*$

For different budgets, the results obtained on the structured instances are very similar, and so are the algorithms. They all feature original simulated annealing components such as the Metropolis acceptance, a geometric cooling scheme with cooling rates between 0.61 and 0.83 and a random neighbourhood exploration. There are instead different strategies for the temperature length and restart. The initial temperature is defined in different ways, all of them relatively to some statistics computed during a preliminary random walk in the solution space. The algorithms obtained for the random instances are more different among each other. The only common component is the Metropolis acceptance criterion. A closer inspection of the search trajectory reveals instead that the algorithms effectively maintain the same or almost the same temperature value for large portions of the search. In other words, the tuning process ends up shaping a relatively complex algorithm that works like a very simple one. Our set of options includes the possibility of maintaining the same temperature throughout the whole search, but it is more difficult to initially sample one with good settings, that has the chance of surviving during the tuning.

The difference in the algorithms obtained can be explained with a closer inspection to the landscapes traversed by the algorithms. On the random

instances the neighbourhoods have a distribution of solution values relatively similar to solutions in areas of different quality around the solution space, making this a scenario that does not require variations of the algorithm parameters. On the structured instances, instead, neighbourhoods centered around average quality solutions have different distributions of values than neighbourhoods around good solutions, and in this case the flexibility of simulated annealing makes it more likely to adapt the algorithm to the different areas of the landscape encountered [34].

It has to be noted that we considered only one tuning task for every budget, and repeating each task with a different random seed may give algorithms that are different, to a certain extent. At the same time, we have run the configuration tasks for the algorithm that has the full training set of 100 parameters for four different budgets. We commonly observe that a higher budget usually is good especially if the configuration tasks have many parameters. Anyway, one should observe that on the structured instances with 20K one has a relatively poor algorithm, something that can happen due to the stochasticity in the configuration process.

4.3 Summary and Discussion

In this chapter, we have seen how to implement a simulated annealing algorithm in terms of the design choices to make. We have shown how to combine the knowledge on algorithmic components and parameter settings with automatic configuration tools to develop efficient simulated annealing algorithms. This methodology is nonetheless general, and works for any stochastic local search method. There are some inherent advantages for this such as making these components and parameters available for future use, allowing experimental analyses to identify the circumstances under which every component will be most successful, and exploiting directly the recent advances in the automatic configuration of algorithms. This applies even more strongly when we want to design an algorithm that finds good solutions as soon as possible, or, in other words, that exhibits a good *anytime behaviour*.

In the experiments we observed how good simulated annealing algorithms look like for different QAP instance classes. In general, in [10] we have many more simulated annealing algorithms for the QAP identified and, independent of which simulated annealing we have, we found the automated configured simulated annealing with the variety of our settings always superior to these specifically designed algorithm for the QAP. An importance analysis conducted across different problems and instance classes indicated in fact that the acceptance criterion is the most important component in a simulated annealing, followed by the neighbourhood exploration criterion. These two components are precisely the ones that operate locally in the neighbourhood. In general, we have seen that different scenarios require different algorithms,

but even for the same scenario we may have different algorithms that perform equally well. Quantifying the appropriate diversification and understanding what algorithm could obtain the desired behaviour are anyway tasks better performed with the help of automatic tools. They can, in fact, select the best options for each algorithmic component and parameter, thus making the best out of the available body of knowledge that can be found in the literature.

A different approach is extending our approach to other stochastic local search methods and to generate extended frameworks. Ideally, these extensions would be generated within a same framework so that possibly rich hybrids among these methods may be generated. This would enable us to compare automatically designed simulated annealings against other automatically designed stochastic local searches, to study the role, impact and composition of simulated annealing algorithms when combined with or used as part of other stochastic local searches. Ultimately, we can try to understand when and how to move beyond the simulated annealing structure, to automatically design bottom up new methods without constraining them to a predetermined form.

4.4 Exercise

In addition to `BR1` and `Q8-7`, in the literature there are several papers proposing SA algorithms for the QAP or for problems that can be modeled as such. We propose an open-ended exercise to become acquainted not only with Simulated Annealing, but also with a component-based perspective on stochastic local search algorithms, and its automatic optimization. You can take inspiration from the Supplementary Material of this chapter[1], but you can also start with a new clean implementation.

1. Choose a SA for the QAP to implement. You can start from the works we cited in this Chapter, or you can look for other SA algorithms.
2. Identify the components of the algorithm you chose comparing it with against the template we provide in Section 4.1.2, along with their numerical parameters.
3. Understand how they interact: think about each of them as a separate function, and analyze which data can be considered *input* and *output*. Use the template given in Algorithm 2 as a reference to determine the flow of information among the components.
4. Implement the algorithm, making sure you can specify the numerical values as command line parameters. SA is a stochastic algorithm, so remember to make it possible to specify a random seed too.

[1] `https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/tree/main/Part%202%20-%20Search-Based%20Optimization/Simulated%20Annealing/SupplementaryMaterial`

5. Run some tests on the instances we provide in the Supplementary Material[2]. Use different random seeds and record the results you observe.
6. Play with the numerical parameters, trying to find a parameter combination that performs consistently better than the original parameter values.
7. Use irace and the templates provided in the Supplementary Material[3] to automatically tune the numerical parameters [24]. Re-run the tests, and observe the difference of results.
8. If you feel brave, implement alternative components, such as new functions to update the temperature value, or to determine whether to accept a candidate solution. You can take these ideas from existing papers, or come up with new components on your own. An implementation that reflects the component-based perspective of SA will make it way easier to observe the impact of your new components. You can even introduce a new command line parameter to add the choice at runtime.

Acknowledgments

Alberto Franzin acknowledges support from the Service Public de Wallonie Recherche under grant n°2010235 - ARIAC by DIGITALWALLONIA4.AI. Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS, of which he is a Research Director.

References

1. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.
2. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
3. V. Černý. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.
4. S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.
5. M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34(1):111–124, 1986.

[2] https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/tree/main/Part%202%20-%20Search-Based%20Optimization/Simulated%20Annealing/SupplementaryMaterial

[3] https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/tree/main/Part%202%20-%20Search-Based%20Optimization/Simulated%20Annealing/SupplementaryMaterial

6. B. Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329, 1988.
7. F. Romeo and A. Sangiovanni-Vincentelli. A theoretical framework for simulated annealing. *Algorithmica*, 6(1-6):302–345, 1991.
8. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation: Part I, graph partitioning. *Operations Research*, 37(6):865–892, 1989.
9. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation: Part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.
10. A. Franzin and T. Stützle. Revisiting simulated annealing: A component-based analysis. *Computers & Operations Research*, 104:191–206, 2019.
11. D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46(1):93–100, 1990.
12. H. Ishibuchi, S. Misaki, and H. Tanaka. Modified simulated annealing algorithms for the flow shop sequencing problem. *European Journal of Operational Research*, 81(2):388–398, 1995.
13. G. Dueck and T. Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90(1):161–175, 1990.
14. P. Moscato and J. F. Fontanari. Stochastic versus deterministic update in simulated annealing. *Physics Letters A*, 146(4):204–208, 1990.
15. E. K. Burke and Y. Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017.
16. T. C. Hu, A. B. Kahng, and C.-W. A. Tsao. Old bachelor acceptance: A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7(4):417–425, 1995.
17. M. S. Hussin and T. Stützle. Tabu search vs. simulated annealing for solving large quadratic assignment instances. *Computers & Operations Research*, 43:286–291, 2014.
18. S. Jajodia, I. Minis, G. Harhalakis, and J.-M. Proth. CLASS: computerized layout solutions using simulated annealing. *International Journal of Production Research*, 30(1):95–108, 1992.
19. D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. *Management Science*, 37(1):98–113, 1991.
20. S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1984.
21. D. Abramson, M. K. Amoorthy, and H. Dang. Simulated annealing cooling schedules for the school timetabling problem. *Asia-Pacific Journal of Operational Research*, 16(1):1–22, 1999.
22. A. Franzin, L. Pérez Cáceres, and T. Stützle. Effect of transformations of numerical parameters in automatic algorithm configuration. *Optimization Letters*, 12(8):1741–1753, 2018.
23. F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In C. A. Coello Coello, editor, *Learning and Intelligent Optimization, 5th International Conference, LION 5*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, Heidelberg, 2011.
24. M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari. The `irace` package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.
25. H. H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, February 2012.
26. M. Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence*. Springer, Berlin/Heidelberg, 2009.

27. F. Mascia, M. López-Ibáñez, J. Dubois-Lacoste, and T. Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, 51:190–199, 2014.
28. T. Stützle and M. López-Ibáñez. Automated design of metaheuristic algorithms. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 272 of *International Series in Operations Research & Management Science*, pages 541–579. Springer, 2019.
29. F. Pagnozzi and T. Stützle. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *European Journal of Operational Research*, 276:409–421, 2019.
30. T. C. Koopmans and M. J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.
31. M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.
32. O. Maron and A. W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Research*, 11(1–5):193–225, 1997.
33. R. E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169–174, 1984.
34. A. Franzin and T. Stützle. A landscape-based analysis of fixed temperature and simulated annealing. Technical Report TR/IRIDIA/2021-005, IRIDIA, Université Libre de Bruxelles, Belgium, 2021.

Chapter 5
Particle Swarm Optimization

Diego Oliva, Alfonso Ramos-Michel, Mario A. Navarro, Eduardo H. Haro, Angel Casas

Nature is the source of inspiration for different processes in science and engineering. Since nature is an example of the world, humans have tried to imitate different behaviors over the years. This occurs not only in a personal way but also happens in the creations and constructions. The collective behavior that is present in an animal that lives in groups is a clear example of how nature solves the problem of finding sources of food, shelters, or places to migrate. In the case of food sources or hunting, a group is more capable of finding food than a single animal. This means that with more members of the group have more probability of catching prey.

In computer sciences, intelligent algorithms' design could be seen as an attempt to imitate nature. In this sense, techniques such as Particle Swarm Optimization (PSO) is an example of inspiration from nature as a base to solve complex problems [1]. The PSO is part of a group of methods called meta-heuristic algorithms that are widely important in artificial intelligence and applied mathematics. Meta-heuristic algorithms can use a biological, physical, or social phenomenon as a source of inspiration and provide the basis for modeling operators to perform optimization.

The PSO was initially proposed by Kennedy and Eberhart in 1995. It was developed to simulate the unpredictable choreography of birds flocking, but later it was used to solve many problems defined discrete-valued spaces where the domain of the variables is infinite [1]. Nonetheless, particle swarm optimization has attracted the scientific attention of miscellaneous engineering and physics areas; this is because there are several fields of study where it is necessary to find better solutions according to specific established criteria. Some other classical optimization approaches cannot be used in these types of problems because they could get caught in local optima [2]. Even though particle swarm optimization has some similarities with genetic algorithms and other optimization approaches, the main difference is that instead

Universidad de Guadalajara, CUCEI

of using mutation/crossover operators, it uses the global communication and real-number randomness as the swarm particles [3].

The PSO works with a population of candidate solutions that are called particles. Each particle of the population collaborates with its search strategy to improve the quality of the solutions. In the early research related to PSO, we can see that the operators move the population as birds flock or school of fishes. PSO is then a simple optimization algorithm that allows to explore a search space and find the optimal solution. Due to its simplicity, PSO is a popular method, and it has been applied in different domains, from benchmark optimization problems to medical applications [4]. PSO has been studied by researchers over the years not only to use it for solving complex problems but also to understand how this algorithm works and to validate its performance from theoretical and empirical points of view [5]. PSO is considered a powerful tool for optimization, and it is a classical approach that helps to inspire other algorithms such as swarm intelligence [6]. However, PSO, like many other algorithms, has some drawbacks; for example, it could be trapped in sub-optimal solutions in complex search spaces (multi-modal problems). In this sense, in the related literature, they are more than 100 modifications of the PSO, and their modification is still growing every year [7].

The popularity of PSO is reflected in the number of cites and publications indexed in different databases like Scopus. Figure 5.1 shows a plot extracted from Scopus related to PSO from 2010 to 2021. From this graph, it is possible to see that in 2010 they were around 4000 documents published, and in 2020 the number of publications was between 8000 and 9000.

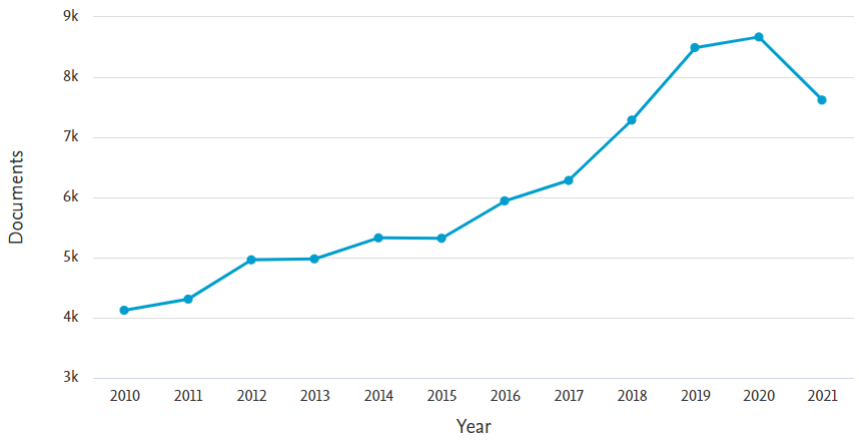


Fig. 5.1: Number of articles per year of PSO between 2010 and 2021.

To graphically show where the PSO impacts in the last ten years Fig. 5.2 presents a pie chart. The main field of application is engineering, followed by computer sciences, and third place in mathematics.

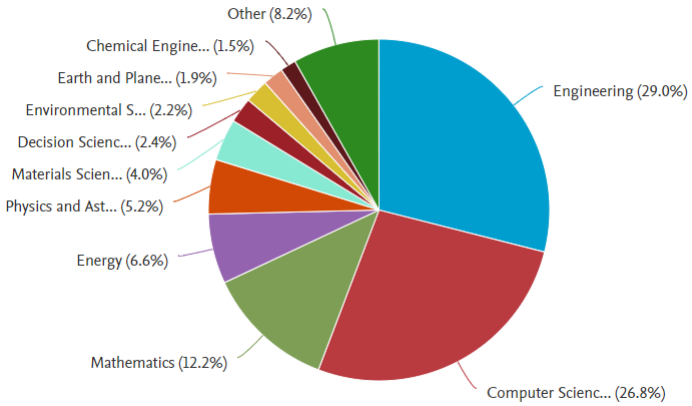


Fig. 5.2: Documents by subject area related to PSO between 2010 and 2021.

In this chapter, an introduction to the PSO algorithm is given. The goal is to provide an overview of the PSO, explain its basic concepts and operators, and present examples and exercises. Besides, the variants of the PSO are also discussed based on their importance in the scientific community. In the same context, they also studied the most important application of this vital algorithm.

The rest of the chapter is organized as follows: Section 5.1 explains the operators of the PSO. Section 5.2 discusses the main modification of the PSO. In Section 5.3 are analyzed the most important PSO applications. Finally, Section 5.4 discuses some conclusions and proposes some exercises.

5.1 The Fundamentals of the PSO Algorithm

This section introduces the basic concepts of the standard PSO and how it works. The pseudocode is described easily, and an example permits an analysis of the behavior of the algorithm.

5.1.1 The PSO Structure

Essentially, each particle in the swarm (which is considered a candidate solution) is randomly distributed and represented by a vector in a multidimensional search space; all this set of particles is considered the initial population. Once established the initial population, PSO searches for optima by updating each particle according to notions such as position, velocity, inertia, etc. These notions can be defined as a vector usually called *the velocity vector*, which helps to determine the next movement of the particle. Nonetheless, this movement is not entirely random; each particle is attracted towards both its own personal best position and the best position of the swarm. Then, the population is evaluated in the objective function to determine the population quality. This also allows finding the best element that will be defined as a criterion to beat in the subsequent evaluation. The new generated positions of the particles and the speed with which they are moving are calculated considering the value of the best global element and the actual value of every particle compared with a random number [8].

When the population is initialized and evaluated, the particle with the lowest or highest objective value is obtained depending on whether it is a minimization or maximization problem, respectively. This found particle is called *the global particle* gB. On the other hand, in each iteration of the algorithm, the current particle is compared with the newly generated particle; in other words, particles have to be evaluated in the objective function every time they are moved. If the generated particle is better than the current one, then it is replaced by the new particle, receiving the usual name of *the actual best particle* lB. The general pseudocode of the particle swarm optimization algorithm is shown in Algorithm 7. In contrast, the phases of initialization and movement of particles that compose the PSO are explained in the following two subsections.

In order to better understand the pseudocode shown in Algorithm 7, it is enough to appreciate that it is divided into two main parts, the unnumbered part and the numbered part. The unnumbered part refers to those terms that the algorithm requires in order to perform its operation, terms such as the population number, dimensions, etc. Once the previous points have been established, the numbered part is given, which begins with the initialization of the population and its respective evaluation. The terms “repeat” and “until” refer to the beginning and end of the while loop, which is the main body of the algorithm.

Algorithm 7 Particle swarm optimization pseudocode

Parameters → Dimensions, Bounds, Maximum iteration number, Number of particles.
Output → gB.

```
Initialize the particles of population.
Evaluate the objective function.
repeat
    for All particles in all dimensions do
        Generate a new velocity.
        Calculate a new position.
        Evaluate again the objective function.
    end for
    Update the best particle of population.
until the maximum number of iterations is reached.
```

5.1.1.1 Initialization of the PSO

All meta-heuristic algorithms have an initialization phase that has the primary purpose of creating the initial population, where each particle represents a candidate solution for the optimization problem. These particles are randomly generated in a search space, which is delimited by the established bounds. Generally, the initialization phase also defines the parameters of the problem to optimize [8]. The initialization of the PSO algorithm is defined by Eq. 5.1, which describes the optimization of its individual particles.

$$\mathbf{x}_{i,j}^{t} = lb_j + rand(ub_j - lb_j) \tag{5.1}$$

where $\mathbf{x}_{i,j}^{t}$ is the $i-th$ population particle, $i \in \{i = 1, 2, 3, ..., N\}$ represents the index of a given particle, N is the maximum number of particles, j represents the j_{th} dimension of the design variable, where $0 < j \leq d$ and d is the dimensionality of the design variable. The iteration number is represented by t. While lb_j and ub_j define the lowest and upper limit of the design variable, respectively. It is worth saying that $rand$ is a uniformly distributed random number between 0 and 1. Once established the respective position of each particle, it is necessary to define the velocity each one of them will move around the search space to find the global optimal. We will explain that phase next.

5.1.1.2 Velocity and Movement of Particles

To find the velocity, it is necessary to get the best global and local values of each particle, commonly called $\mathbf{gB}^t$ and $\mathbf{lB}_i^t$ respectively. Eq. 5.2 defines the velocity calculation for each particle i:

$$\mathbf{v}_i^{t+1} = w \times \mathbf{v}_i^t + c_1 \times \mathbf{r1}_i^t \times (\mathbf{lB}_i^t - \mathbf{x}_i^t) + c_2 \times \mathbf{r2}_i^t \times (\mathbf{gB}^t - \mathbf{x}_i^t) \tag{5.2}$$

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{v}_i^{t+1} \tag{5.3}$$

where $\mathbf{v}_i^{t+1}$ is the particle i's velocity of the iteration $t+1$, $\mathbf{v}_i^t$ is the particle i's velocity in the previous iteration, the vector that contains each particle i's position is $\mathbf{x}_i^t$ and $\mathbf{r1}_i^t$ and $\mathbf{r2}_i^t$ represent d-dimensional vectors containing uniformly distributed random numbers between 0 and 1. It is worth to say that c_1 and c_2 are called learning coefficients, and w represents an inertia weight that affects the convergence and exploration-exploitation trade-off in PSO process. Since inception of inertia Weight in PSO, a large number of variations of Inertia Weight strategy have been proposed, nonetheless, it is generally established as 1. Meanwhile, Eq. 5.3 is used to displace the particles to new positions in the next iteration. Where $\mathbf{x}_i^{t+1}$ is the vector where the new obtained position of particle i at iteration $t+1$ is stored, $\mathbf{x}_i^t$ corresponds to the previous position of particle i calculated at iteration t, and finally $\mathbf{v}_i^{t+1}$ is the velocity vector obtained by Eq. 5.2.

It is essential to mention that the majority of modifications to the PSO algorithm have as purpose to find new ways to accelerate the particles as best as possible [9]. To end this section, Fig. 5.3 shows a graphic description of particle's velocity and their movement in the particle swarm optimization algorithm for a better understanding by the reader.

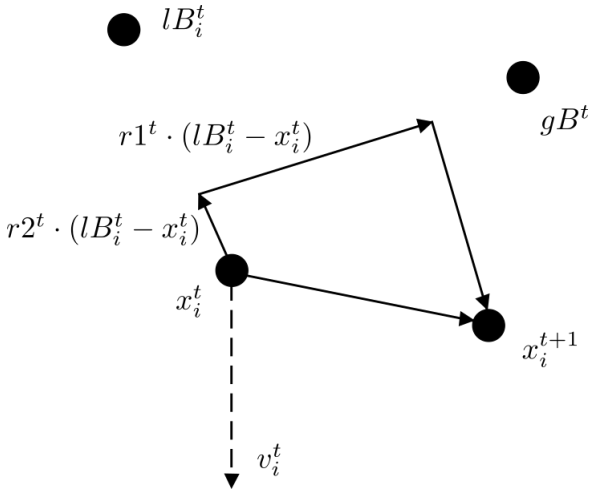


Fig. 5.3: Velocity and movement of particles in PSO.

5.1.2 A PSO Example

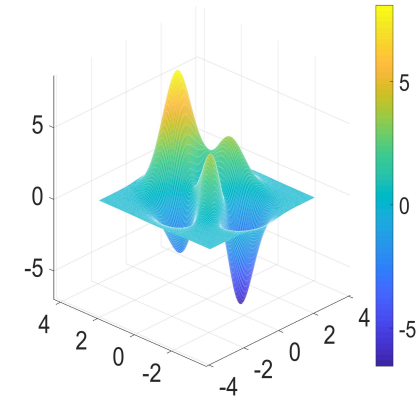
To show how the PSO works here is presented a graphical example. Figure 5.4a shows the peaks function plotted in 3D with a lateral view. This function is commonly used in the literature to explain the behavior of optimization algorithms. Figure 5.4b shows the random initialization of the population of a particle (swarm) in PSO [1]. The population has 30 particles uniformly distributed in the search space defined by the peaks function with an upper bound as $ub = 3$ and a lower bound as $lb = -3$ (from the top view). The examples show the distribution of particles at different stages of the optimization process establishing 1000 iterations as a stop criteria. In Figs. 5.4c, 5.4d, 5.4e and 5.4f it is shown the interaction of the swarm in 50, 300, 600 and 999 iterations in the test function. Finally, in Fig. 5.4f it is possible to see that the swarm has found the zone of the global minimum according to the color bar for the z-axis of Fig. 5.4a.

5.2 The PSO Variants

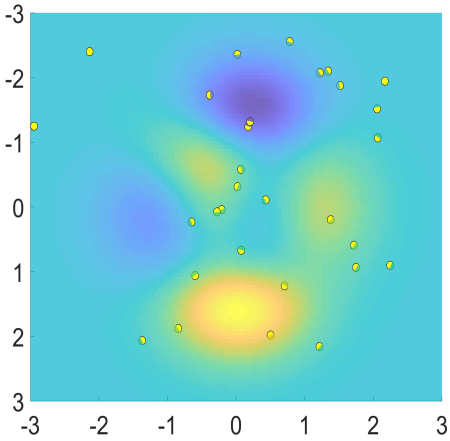
This algorithm has been so successful since its emergence that it has been the base for designing new methods. Same that preserves the essence and structure of the PSO but adds improvements in operators, parameters, initialization, or combines concepts that make the algorithm more specialized for specific problems. The most important variants are listed below, divided into two fields, single-objective and multi-objective optimization:

5.2.1 Top PSO Variants for Single-Objective Problems

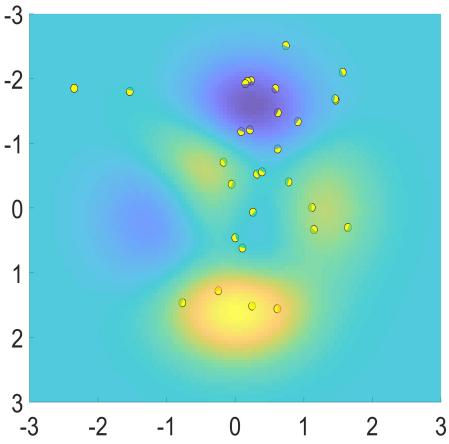
There are many publications related to PSO variants for single-objective optimization. Most of them were compiled and sorted by year of publication, adding the citations obtained from publication date up to the current year 2021. Based on this information, the top 30 most cited PSO variants were obtained (see Table 5.2.1).



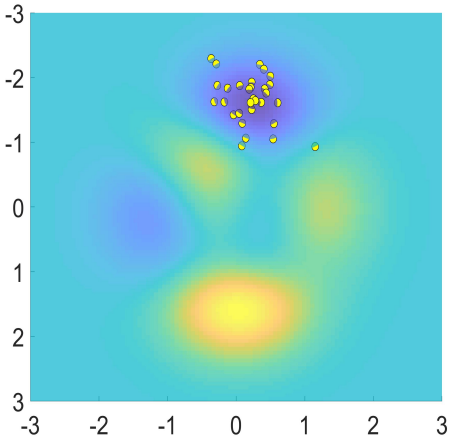
(a) Peaks function.



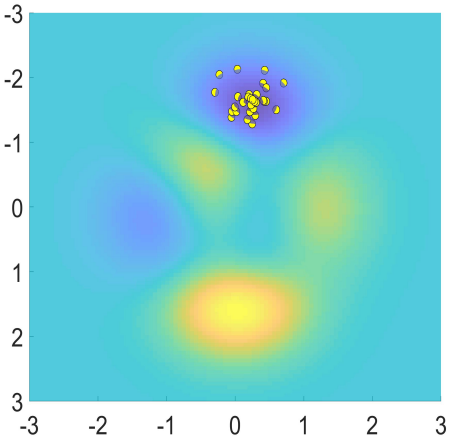
(b) Distribution of the swarm at the initialization.



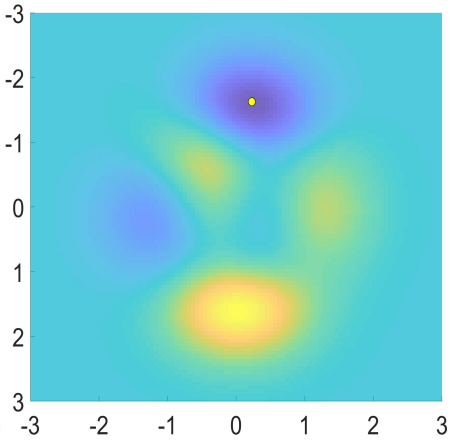
(c) Distribution of the swarm at 50 iterations.



(d) Distribution of the swarm at 300 iterations.



(e) Distribution of the swarm at 600 iterations.



(f) Final distribution of the swarm at 999 iterations.

Fig. 5.4: An example of the PSO along the iterative process.

Table 5.1: Top 30 PSO variants

PSO variants	publication year	current citations
PSO(PSO)	1995	68303
Binary PSO (BPSO)	1997	6051
Comprehensive Learning PSO (CLPSO)	2006	3426
Cooperative PSO (CPSO_S)	2004	2303
Fully-informed PSO (FIPS)	2004	2030
Quantum Delta- Potential-Well-Based PSO (QDPSO)	2004	1990
Fuzzy PSO (FPSO)	2001	1876
Quantum-inspired version of the PSO algorithm (QPSO)	2004	1080
Dynamic neighborhood PSO (DNPSO)	2003	976
Bare bones particle swarms	2003	806
Cooperatively Coevolving Particle Swarms (CCPSO)	2008	679
Fitness-to-Distance Ratio PSO (FDRPSO)	2003	598
Attractive Repulsive Particle Swarm Optimization (ARPSO)	2002	597
PSO with passive congregation (PSOPC)	2004	470
Dissipative PSO (DPSO)	2002	464
PSO with spatial particle extension (SEPSO)	2002	457
Species in a Particle Swarm Optimizer (SPSO)	2004	366
Optimized PSO (OPSO)	2006	333
Standard PSO(SPSO)	2011	308
Orthogonal PSO (OPSO)	2008	275
Hybrid gradient descent PSO (HGPSO)	2004	264
Extended Particle Swarms (XPSO)	2005	264
Dual Similar PSO Algorithm (DSPSOA)	2006	264
Parallel PSO (PPSO)	2005	255
Evolutionary Iteration PSO (EIPSO)	2007	255
Adaptive PSO (APSO)	2002	218
Iteration PSO (IPSO)	2007	190
Enhanced leader PSO(ELPSO)	2015	190
Angle Modulated PSO (AMPSO)	2005	169
Co-evolutionary PSO	2002	162

The following is a brief description of the most widely used PSO variants. Considering the ranking in Table 5.2.1 and some state of the art reviews, [10], [11] [12] [13] indicating which are the most popular variants. The objective is to present some of the most representative varieties of the PSO algorithm considering significant modifications in terms of several concepts such as social topology between particles, new definitions of equations for velocity and position, and methodologies that make the algorithm adaptive in some of its global parameters.

5.2.1.1 Standard PSO 2011

In the initialization process of standard PSO (SPSO), the size of the swarm is defined by the user, but it is empirically suggested $N = 40$ particles. The algorithm starts initializing the particles following a random distribution [14], while the other elements are initialized following a uniform distribution. If any element goes out of bounds, these act as a barrier returning the element within the bounds $[lb_j, up_j]$ resetting the particle velocity to 0, where j corresponds to the dimension number. Until the 2007 versions, the velocity is updated dimension by dimension, as can be seen in Eq. 5.2, and the performance of

the algorithm depends mainly on the coordinate system obtained (rotation sensitivity) [15], [16].

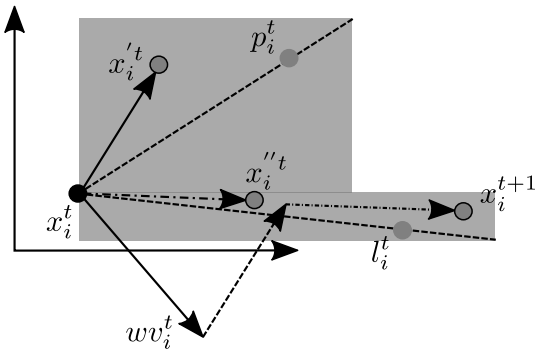


Fig. 5.5: Geometrical representation of standard PSO until 2007

Figure 5.5 shows that each particle at each time step gets its distribution of all nearby positions and which is also a combination of two rectangles D with sides parallel to the axes, with uniform probability distribution within each rectangle. In contrast, the variant proposed in 2011 [15] suggests the idea of an adaptive random topology in terms of communication between search agents and exploits the idea of rotational invariance (see Fig. 5.6).

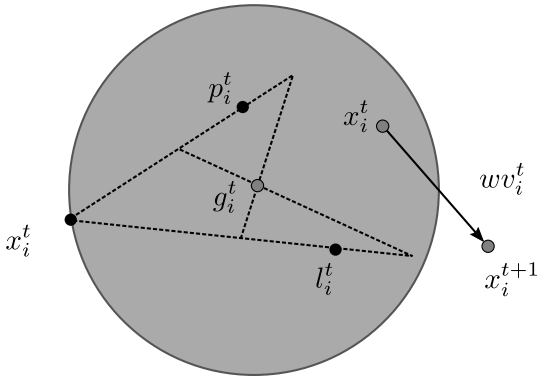


Fig. 5.6: Geometrical representation of standard PSO

For each particle i and at each time stage t, a center of gravity $\mathbf{g}_i^t$ is defined around three points:

- The current position ($\mathbf{x}_i^t$)

- Point a little "beyond" the best previous personal position ($\mathbf{p}_i^t$) defined as:

$$\mathbf{p}_i^t = \mathbf{x}_i^t + \ c_1\mathbf{r}1_i^t \otimes (\ \mathbf{lB}_i^t \ - \mathbf{x}_i^t) \tag{5.4}$$

- Point a little "beyond" the best previous position in the neighborhood ($\mathbf{l}_i^t$) defined as:

$$\mathbf{l}_i^t = \mathbf{x}_i^t + \ c_2\mathbf{r}2_i^t \otimes (\ \mathbf{gB}^t \ - \mathbf{x}_i^t) \tag{5.5}$$

- Center of gravity($\mathbf{g}_i^t$) finally defined as:

$$\mathbf{g}_i^t = \frac{\left(\mathbf{x}_i^t + \mathbf{p}_i^t + \mathbf{l}_i^t\right)}{3} \tag{5.6}$$

In the case of Fig. 5.6 the *distribution of all possible next positions* (DNPP) obtained is a D-dimensional sphere defined as $\mathcal{H}_i(\mathbf{g}_i^t, ||\mathbf{g}_i^t, \mathbf{x}_i^t||)$, which is rotational invariant about its center, i.e., it does not change when the function is rotated.
In SPSO-2011, velocity is updated as follows:

$$\mathbf{v}_i^{t+1} = w\mathbf{v}_i^t + \mathcal{H}_i(\mathbf{g}_i^t, ||\mathbf{g}_i^t - \mathbf{x}_i^t||) - \ \mathbf{x}_i^t \tag{5.7}$$

While particle position is updated following Eq. (5.3).

5.2.1.2 Fully-informed PSO

Human social behaviors inspired the authors to conclude that individuals are not influenced by a single individual but by those around them. Based on observations, they proposed a modification of the velocity equation of the canonical version of PSO [1] with the purpose that each particle is influenced by the performance of all its neighbors and not by the performance of a single individual. The equations for the update of velocity and position are defined as:

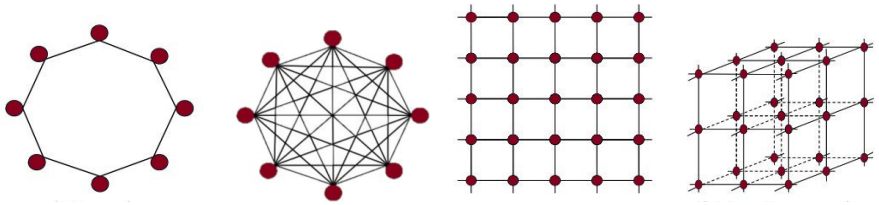
$$\mathbf{v}_i^{t+1} = w\mathbf{v}_i^t + \sum_{m=1}^{|C_i|} \gamma_m^t \frac{(\mathcal{Y}_m^t - x_i^t)}{|C_i|} \tag{5.8}$$

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \ \mathbf{v}_i^{t+1} \tag{5.9}$$

Inspired by the fact that an individual is influenced by the behavior of the individuals around him, the authors define the Eq. 5.8. Where C_i is the set of particles in the neighborhood of i, and $\gamma_m^t \in [0, c_1 + \ c_2]^j$, where c_1 and c_2 are the learning coefficients, j is the dimensionality of the problem and the position $\mathcal{Y}_m^t$ represents the best position that particle m has visited until iteration t (a position that has obtained the best evaluation of the objective

function). The main contribution of this algorithm published in 2004 that makes it different from other variants is the type of topology used in its particles [17]. Since they all have the exact source of information, there is no difference in the amount of information shared. Some of the topologies that can affect or improve the performance of PSO are the following:

- *Ring*: this topology has a network configuration where the particle connections create a circular trajectory. Each particle in the network is fully connected to two others, thus forming a single continuous route, resulting in a ring structure as illustrated in Fig. 5.7a. The algorithm that results from this topology is called Lbest PSO; in addition, such a topology can be generalized to a network structure in which neighborhoods of larger size are used.
- *Star*: this social topology is shown in Fig. 5.7b. Here, it is observed that all the particles in the swarm are interconnected. A PSO using the star topology is commonly named the Gbest PSO . The original implementation of the PSO used the star topology [1].
- *Von Neumann*: this topology is a square grid where each particle has four neighbors. The 2-D variant is represented in Fig. 5.7c, and the 3-D variant is represented in Fig. 5.7d. This class of topologies allows the communication between the particles to be unusual and different from other social communication topologies.



(a) Ring. (b) Star. (c) Square. (d) Cube.

Fig. 5.7: Social topologies

5.2.1.3 Comprehensive Learning PSO

The main novelty of this algorithm is that the authors propose a new function for updating the particle velocity defined as:

$$\mathbf{v}_i^{t+1} = w * \mathbf{v}_i^t + c * \mathbf{r1}_i^t * \left(\mathbf{lB}_{f_i}^t - \mathbf{x}_i^t\right) \tag{5.10}$$

where $fi = [f_i(1), f_i(2), \cdots, f_i(D)]$ defines which particles of lB the particle i should follow, and D represents the dimensionality of the problem. The value of $lB_{f_i(j),j}$ can be any particle's lB including its own lB. The decision

depends on probability P_c, referred to as the learning probability, which can take different values for different particles. For each dimension of particle i, the authors generate a random number. If this random number is higher than P_{c_i}, the corresponding dimension will learn from its own lB; otherwise, it will learn from another particle's lB.The selection method used by the tournament and the main steps for selection are as follows:

- First, two particles are chosen randomly from the population, except the particle with its updated velocity.
- The objective value of the two particles(lB) is compared, and the best of these is selected. In CLPSO the objective value is defined as the higher, the better, which means that to solve minimization problems, use the negative values of the function as objective values.
- Finally, using the winning particle as an example for the next generations to learn from that dimension, if all individuals are their own lB, then a random dimension is chosen for the particles to learn from the best lB particles working on it.

5.2.1.4 Fuzzy Adaptative PSO

This version of PSO is based on fuzzy inference systems. For more information about this computational area, the reader is advised to refer to the following references [19, 20]. Readers may understand this version of PSO more fully after learning about fuzzy inference systems, but a brief description of this approach is provided below.

Fuzzy inference systems are basically conditional statements expressed in the form IF A THEN B, where A and B are labels of fuzzy sets defined by appropriate membership functions [18]. A fuzzy inference system is depicted in Fig. 5.8 and is composed of four functional blocks as described below:

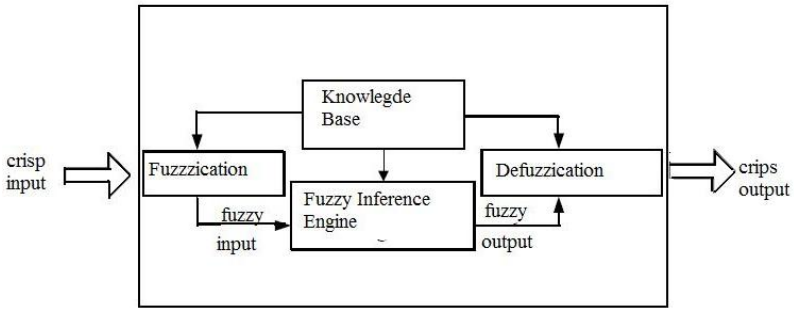


Fig. 5.8: Fuzzy inference system

- A *knowledge base* containing a number of fuzzy if-then rules and the definitions of the memebership functions of the fuzzy sets used in the fuzzy rules.
- A *fuzzy inference engine* which performs the inference operations based on the fuzzy rules.
- A *fuzzification interface* that transforms the crisp inputs into degrees of match with linguistic values.
- A defuzzification interface which transforms the fuzzy results of the inference into a crisp output.

The authors of the paper [21] published in 2001 proposed to adapt PSO by using a fuzzy system to adjust the inertia weight dynamically. As mentioned previously, the inertia weight is an important parameter in PSO, since it affect the convergence and exploration-exploitation trade-off in PSO process. As the inertia weight is a global variable, it can be applied to the whole population. The authors thus designed a fuzzy system to adapt the inertia weight dynamically as a global variable. The fuzzy system's input variables are the performance of the best candidate solution found so far by the PSO in normalised format and the current inertia weight, whereas the output variable is the change in inertia weight. All variables are associated to three fuzzy sets: low, medium, and high with associated membership functions as $left\_triangle$, $triangle$ and $right\_triangle$, respectively. The definitions of these three membership functions are:

$$f_{left\_triangle} = \begin{cases} 1 & \text{if } x < x_1 \\ \frac{x_2 - x}{x_2 - x_1} & \text{if } x_1 \leq x \leq x_2 \\ 0 & \text{if } x > x_2 \end{cases} \tag{5.11}$$

$$f_{triangle} = \begin{cases} 0 & \text{if } x < x_1 \\ 2\frac{x - x_1}{x_2 - x_1} & \text{if } x_1 \leq x \leq \frac{x_2 + x_1}{2} \\ 2\frac{x_2 - x}{x_2 - x_1} & \text{if } \frac{x_2 + x_1}{2} < x \leq x_2 \\ 0 & \text{if } x > x_2 \end{cases} \tag{5.12}$$

$$f_{right\_triangle} = \begin{cases} 0 & \text{if } x < x_1 \\ \frac{x - x_1}{x_2 - x_1} & \text{if } x_1 \leq x \leq x_2 \\ 1 & \text{if } x > x_2 \end{cases} \tag{5.13}$$

where x_1 and x_2 are critical parameters that determine the shape and location of the functions. Other definitions of membership functions are possible, but the authors have found that the ones they proposed are useful for various problems and are easy to implement on microcontrollers and microprocessors.

5.2.2 Major PSO Variants for Multi-Objective Optimization

PSO is a powerful tool for solving single-objective problems; for that reason, researchers expanded its capabilities to the multi-objective domain. The idea is to generate PSO-based approaches that can optimize more than one objective function. Assume a multi-objective optimization problem in the canonical form given in the introduction of Part II of this book:

$$\begin{array}{ll} \text{minimize} & f_k(\mathbf{x}), \qquad k = \{1, 2, \cdots, p\} \\ \text{subject to} & g_i(\mathbf{x}) \leq 0, \quad i = \{1, 2, \cdots, m\} \\ & h_j(\mathbf{x}) = 0, \quad j = \{1, 2, \cdots, n\} \ , \end{array}$$

where $\mathbf{x} \in \mathcal{X}$ is the design variable being optimised, p is the number of objective functions to be optimised, m is the number of constraints of the type g_i and n is the number of constraints of the type h_j. Multi-objective optimization (MO) aims to find a set of solutions called a Pareto set [22], where each solution is non-dominated by any other solution in the search space. A solution $\mathbf{x}^{(1)}$ is said to dominate another solution $\mathbf{x}^{(2)}$ if and only if:

$$\begin{array}{l} \forall i, f_i(\mathbf{x}^{(1)}) \leq f_i(\mathbf{x}^{(2)}) \text{ and} \\ \exists i, f_i(\mathbf{x}^{(1)}) < f_i(\mathbf{x}^{(2)}) \end{array} \tag{5.14}$$

As such, the Pareto set can be seen as composed of solutions that represent different trade-offs among the objectives. This section briefly discusses the most relevant versions of the PSO for MO problems. These variants were selected based on their citations in scientific repositories.

5.2.2.1 MOPSO

This algorithm proposed in 2002 introduces concepts of multiobjective optimization to modify the original PSO algorithm and implement it to solve multiobjective optimization problems. The authors named this approach Multi-Objective Particle Swarm Optimization (MOPSO) [23] and validated it by comparing it with algorithms such as Pareto Archived Evolution Strategy (PAES) [24] and the Non-dominated Sorting Genetic Algorithm II (NSGA II) [25]. The general structure of the MOPSO algorithms is listed below:

1. Initialize population with random position and velocity vectors.
2. Evaluate the objective values of each particle.
3. Archive=non-dominated solutions found so far.
4. Determine the personal best of each particle.
5. Apply Eqs. 5.2 and 5.3.
6. Repeat steps 2 to 6 until the stop criterion is reached.

In MOPSO the velocity and position update equations remain similar to Eq. (5.2) and Eq. (5.3) of PSO, but the personal best and global best are computed differently. The personal best is determined based on the non-dominance concept. If the new position of a particle dominates the previous personal best of this particle, the new position replaces the previous personal best. If the previous personal best dominates the new position, then the previous personal best is kept. If the new position and the previous personal best are non-dominated by each other, one of them is chosen uniformly at random to become (or remain) the personal best of this particle. The global best is randomly selected among the non-dominated solutions from the archive by giving higher chance to select solutions located in less crowded regions of the search space, i.e., in regions where there are few archive solutions.

This multi-objective algorithm has been widely used in the literature for applications in different fields; three of the most notable fields, according to the reports in [26] are Electrical Engineering, Industrial Engineering, Flow-shop, and Job-shop Scheduling.

5.2.2.2 SMPSO

In 2009, Nebro et al. [27] studied six representative variants of MOPSO and found that they had problems solving multi-modal problems, i.e., problems with two or more optimal solutions. The problem was the "swarm explosion" [28]: erratic movements toward the upper and lower boundaries of particle positions that occur when particles acquire very high velocity. Based on a MOPSO algorithm, they presented in [27] a new multi-objective PSO algorithm using the velocity constraint strategy [28]. Their proposal could obtain proper particle positions when high velocity was present, avoiding the swarm explosion effect. The new algorithm, called Speed-constrained Multi-objective PSO (SMPSO), uses the following equations to adjust the velocity of the particles:

$$\chi = \frac{2}{2 - \varphi - \sqrt{\varphi^2 - 4\varphi}} \tag{5.15}$$

where

$$\varphi = \begin{cases} c_1 + c_2 \text{ if} & \mathrm{c_1 + c_2 > 4} \\ 1 \quad \text{if} & \mathrm{c_1 + c_2 \leq 4} \end{cases} \tag{5.16}$$

where c_1 and c_2 are the parameters to control the effect of the personal and global best particles. Once the velocity of the particles is calculated in the original form [27], it is multiplied by the constriction coefficient χ, which helps to enable diverse solutions so that exploitation is not too much intensified. Then, the velocity is constrained by Eq. 5.17:

$$v_{i,j}^t = \begin{cases} \Delta b_j & \text{if } v_{i,j}^t > \Delta b_j \\ -\Delta b_j & \text{if } v_{i,j}^t \leq -\Delta b_j \\ v_{i,j}^t & \text{otherwise} \end{cases} \tag{5.17}$$

where

$$\Delta b_j = \frac{(ub_j - lb_j)}{2} \tag{5.18}$$

where ub and lb are the upper and lower boundaries of search space defined by $\mathcal{X}$, and Δb_j is the center point between them. By implementing this strategy, SMPSO overcomes the swarm explosion effect of MOPSO algorithms. They reported that the approximations to the Pareto front found by their algorithm were better than the MOPSO algorithms. The SMPSO was the fastest algorithm converged to the Pareto front in their experiments.

5.2.2.3 FC-MOPSO

This algorithm presented in 2017 [29] originally targeted at the problem of multi-objective design of engineering problems with few function evaluations. It generalizes the concept of personal best $\mathbf{lB}_i^t$ to the set P_i, which contains all non-dominated solutions found so far by the particle i. A particle called $\mathbf{psel}_i$ is selected from P_i for the purpose updating the velocity (and position) of the particle i in the algorithm. Similarly, the concept of local best $\mathbf{lB}_i$, which is used by some PSO variants to capture the best particle found in the neighbourhood of particle i instead of in the population of particles as a whole, is also generalized to randomly select non-dominated solutions.

The authors also suggest that FC-MOPSO starts with random velocities no greater than $v_m = 0.1(ub - lb)$, where ub and lb are the upper and the lower boundaries of the search space defined by $\mathcal{X}$, respectively. In other words, they prevent the method from starting with large velocities while starting with some speeds to start the search at a more advanced stage.

5.3 PSO Applications

The applications presented here were chosen based on the relevance of current works consulted in the *GoogleScholar* repositories.
In recent years, robotics has undergone significant development. The specialization of many tasks requires the optimization of execution times and precision in the robot's movements. In this sense, the modeling parameters of the robot were explored in [30], where the researchers used a combination of the Least Square (LS) method and PSO to estimate a robotic arm's inertia parameters. The estimation performed automatically by the algorithm matches closely with the manufacturer-provided values. Trajectory planning for mobile robots through PSO was implemented by Abdalla et al. [31], Zeng

et al. [32], and Wang et al. [33]. Zeng et al. used a PSO based on a non-homogeneous Markov chain and Differential Evolution algorithm to generate a path into a space with obstacles for intelligent robots. Abdalla et al. proposed a combination of the artificial potential field (APF) with fuzzy logic, where PSO was used to optimize the membership functions of the fuzzy controller to acquire the mobiles robot trajectory, and Wang et al. combined the PSO algorithm with kinematics equations to trajectory planning in the free-floating space robot, simulating a couple of a spacecraft and a manipulator robot.

Electrical energy has been studied for centuries, and its use has been extended to all possible areas in recent decades. Now, with the renewable production of electrical energy, it is imperative to optimize its use and its distribution and production for maximum utilization. Electricity sometimes comes from a cascade system of hydroelectric generators. Evaporation, irrigation systems, and other factors affect the water level in the reservoirs. Mahor and Saroj [34] used a PSO based algorithm to optimize the generation schedule of a real system of hydroelectric cascade system in India. In [35], Bahrami et al. used the PSO algorithm to determine the optimal parameter in a grey model. This model brings the opportunity to forecast the electric load in the short term, using minimal historical data. The method was used to forecast the load in Iran and New York electrical networks. In 2016, Mesbahi et al. used a hybrid PSO-Nelder-Mead algorithm to identify and model electric vehicle batteries and then used this information to simulate vehicle energy demand, which allows maximum energy harvesting [36]. Finally, Mesbahi et al. compared their results against the power consumption of a physical urban electric vehicle, with a modeling error inferior to 0.5%. Mozafar et al. applied a multiobjective algorithm, using PSO combined with GA to optimize the location and capacity of renewable power sources and electric vehicle charging stations within the same distribution network. [37].

5.4 Summary and Discussion

The PSO is an interesting algorithm that has attracted the attention of researchers and practitioners from different areas over the years. The simplicity of the PSO makes it a popular mechanism for single-objective optimization. Their operators are easy to implement and permit finding solutions to complex problems. Since the PSO has been extensively used, it has also been modified to improve its performance. Multi-objective problems require more sophisticated algorithms that permit handle with them. In this sense, the PSO has been extended for multi-objective optimization. Regarding the importance of the PSO, it is possible to see how this method stills being a source of research from different repositories. The multiple modifications of the PSO algorithm are an excellent example of this. This chapter just considered the

most relevant derivative work of PSO from 2010 to 2020. There exists more, but analyzing them is a complex and irrelevant task to the present work scopes. The quantity of derivative works is similar to the number of applications of the PSO; since its implementation is relatively easy and the code is entirely available, it is possible to find a significant number of articles that reports result from different fields of applications. It is possible to conclude that the PSO is one of the most important meta-heuristic algorithms, and the research related to it is still has lots to promise.

5.4.1 Exercises

1. Program the Rosenbrock function defined as follows:

$$f(x_1 \cdots x_n) = \sum_{i=1}^{n-1} (100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2) \tag{5.19}$$

   where $-2.048 \leq x_i \leq 2.048$ and the minimum at $f(1, 1, \cdots, 1) = 0$. Once you have implemented the Rosenbrock function plot its surface in 2 dimensions.
2. Considering the Rosenbrock function, initialize a random population of 50 particles and plot them in the the 2-dimensional surface of the function.
3. Implement the standard version of the PSO explained in this chapter to minimize the Rosenbrock function. You need to obtain the curve of convergence of the best solution at each iteration and plot it.
4. Implement a code of the Fully Informed PSO that was explained in this chapter. The Fully Informed PSO should be able to minimize the Rosenbrock function. Obtain the convergence curve and compare it with the one obtained by the standard PSO.

5.4.2 Answers to the exercises

1. Rosenbrock function top view

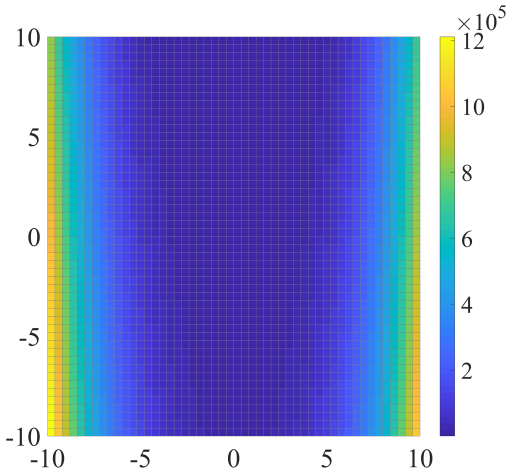


Fig. 5.9: Rosenbrock function

2. Initialization of a random population of 50 particles on the 2-dimensional surface of the Rosenbrock function.

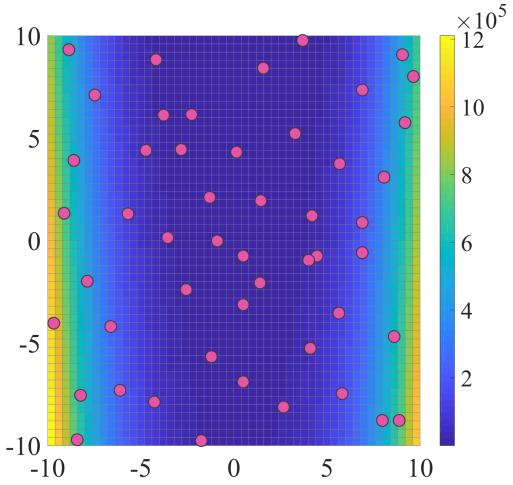


Fig. 5.10: Initialization of 50 particles on the Rosenbrock function

3. Convergence plot of standard particle swarm optimization version

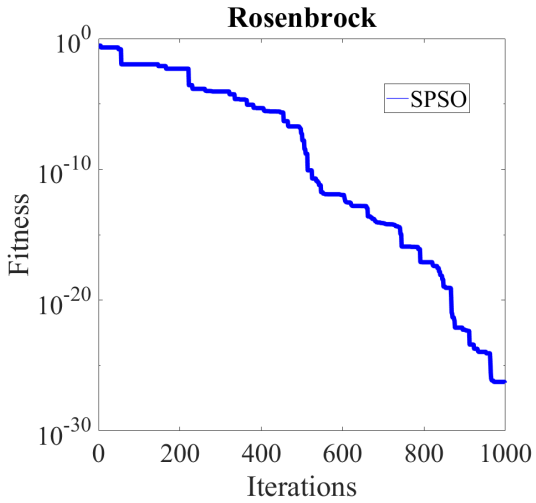


Fig. 5.11: SPSO convergence graph

4. Comparison of convergence plots between standard particle swarm optimization (SPSO) and Fully informed Particle Swarm (FIPS).

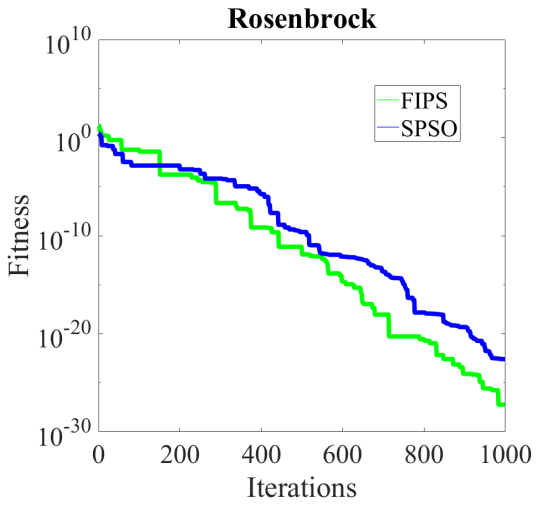


Fig. 5.12: SPSO vs. FIPS convergence plots

References

1. James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.
2. Alexandar Lazinica and Janeza Trdine. *Particle Swarm Optimization*. IntechOpen, 2009.
3. Xin-She Yang. *Nature Inspired Metaheuristic Algorithms Second Edition*. Luniver Press, 2010.
4. Yuhui Shi et al. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, volume 1, pages 81–86. IEEE, 2001.
5. Yuhui Shi and Russell C Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 1945–1950. IEEE, 1999.
6. Russell C Eberhart, Yuhui Shi, and James Kennedy. *Swarm intelligence*. Elsevier, 2001.
7. Muhammad Imran, Rathiah Hashim, and Noor Elaiza Abd Khalid. An overview of particle swarm optimization variants. *Procedia Engineering*, 53:491–496, 2013.
8. Erik Cuevas and Alma Rodirguez. *Metaheuristic Computation with MATLAB*. CRC Press, 2021.
9. Erik Cuevas, Jose Ozuna, Diego Oliva, and Margarita Diaz. *OPTIMIZACION, Algoritmos programados con MATLAB*. Alfaomega, 2016.
10. Esperanza Garcia-Gonzalo and Juan Luis Fernandez-Martinez. A brief historical review of particle swarm optimization (pso). *Journal of Bioinformatics and Intelligent Control*, 1(1):3–16, 2012.
11. Ivo Sousa-Ferreira and Duarte Sousa. A review of velocity-type pso variants. *Journal of Algorithms & Computational Technology*, 11(1):23–30, 2017.
12. Davoud Sedighizadeh and Ellips Masehian. Particle swarm optimization methods, taxonomy and applications. *International journal of computer theory and engineering*, 1(5):486, 2009.
13. Keisuke Kameyama. Particle swarm optimization-a survey. *IEICE transactions on information and systems*, 92(7):1354–1361, 2009.
14. Maurice Clerc. *Particle swarm optimization*, volume 93. John Wiley & Sons, 2010.
15. Maurice Clerc. Beyond standard particle swarm optimisation. In *Innovations and Developments of Swarm Intelligence Applications*, pages 1–19. IGI Global, 2012.
16. William M Spears, Derek T Green, and Diana F Spears. Biases in particle swarm optimization. In *Innovations and developments of swarm intelligence applications*, pages 20–43. IGI Global, 2012.
17. Rui Mendes. Population topologies and their influence in particle swarm performance. *PhD Final Dissertation, Departamento de Informática Escola de Engenharia Universidade do Minho*, 2004.
18. L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Trans. Syst., Man, Cybern*, 3:28–44, 1973.
19. M. Sugeno T. Takagi. Fuzzy identification of systems and its applications to modeling and control. *IEEE Trans. Syst., Man, Cybern*, 15:116–132, 1985.
20. Roger Jang Jyh-Shing. Anfis: Adaptive-network-based fuzzy inference system. *IEEE Trans. Syst., Man, Cybern*, 23(3):665–685, 1993.
21. Yuhui Shi and Russell C Eberhart. Fuzzy adaptive particle swarm optimization. In *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, volume 1, pages 101–106. IEEE, 2001.
22. Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007.

23. CA Coello Coello and Maximino Salazar Lechuga. Mopso: A proposal for multiple objective particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, volume 2, pages 1051–1056. IEEE, 2002.
24. Joshua Knowles and David Corne. The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 1, pages 98–105. IEEE, 1999.
25. Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.
26. Soniya Lalwani, Sorabh Singhal, Rajesh Kumar, and Nilama Gupta. A comprehensive survey: Applications of multi-objective particle swarm optimization (mopso) algorithm. *Transactions on combinatorics*, 2(1):39–101, 2013.
27. Antonio J Nebro, Juan José Durillo, Jose Garcia-Nieto, CA Coello Coello, Francisco Luna, and Enrique Alba. Smpso: A new pso-based metaheuristic for multi-objective optimization. In *2009 IEEE Symposium on computational intelligence in multi-criteria decision-making (MCDM)*, pages 66–73. IEEE, 2009.
28. Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.
29. Vahid Mokarram and Mohammad Reza Banan. A new pso-based algorithm for multi-objective optimization with continuous and discrete design variables. *Structural and Multidisciplinary Optimization*, 57(2):509–533, 2018.
30. Zafer Bingül and Oğuzhan Karahan. Dynamic identification of staubli rx-60 robot using pso and ls methods. *Expert Syst. Appl.*, 38(4):4136–4149, April 2011.
31. Turki Y Abdalla, Ali A Abed, and Alaa A Ahmed. Mobile robot navigation using pso-optimized fuzzy artificial potential field with fuzzy control. *Journal of Intelligent & Fuzzy Systems*, 32(6):3893–3908, 2017.
32. Nianyin Zeng, Hong Zhang, Yanping Chen, Binqiang Chen, and Yurong Liu. Path planning for intelligent robot based on switching local evolutionary pso algorithm. *Assembly Automation*, 2016.
33. Mingming Wang, Jianjun Luo, and Ulrich Walter. Trajectory planning of free-floating space robot using particle swarm optimization (pso). *Acta Astronautica*, 112:77–88, 2015.
34. Amita Mahor and Saroj Rangnekar. Short term generation scheduling of cascaded hydro electric system using novel self adaptive inertia weight pso. *International Journal of Electrical Power & Energy Systems*, 34(1):1–9, 2012.
35. Saadat Bahrami, Rahmat-Allah Hooshmand, and Moein Parastegari. Short term electric load forecasting by wavelet transform and grey model improved by pso (particle swarm optimization) algorithm. *Energy*, 72:434–442, 2014.
36. Tedjani Mesbahi, Fouad Khenfri, Nassim Rizoug, Khaled Chaaban, Patrick Bartholomeues, and Philippe Le Moigne. Dynamical modeling of li-ion batteries for electric vehicle applications based on hybrid particle swarm–nelder–mead (pso–nm) optimization algorithm. *Electric power systems research*, 131:195–204, 2016.
37. Mostafa Rezaei Mozafar, Mohammad H Moradi, and M Hadi Amini. A simultaneous approach for optimal allocation of renewable energy sources and electric vehicle charging stations in smart grids based on improved ga-pso algorithm. *Sustainable cities and society*, 32:627–637, 2017.

Chapter 6
Other Search-Based Optimization Approaches

Roberto Santana

In previous chapters we have seen detailed examples of search-based optimization algorithms. Here, we present an overview of other search-based techniques. This class of methods can be grouped according to different criteria, among them:

- Single solution versus multiple solutions search-based algorithms.
- Black-box versus gray-box optimization algorithms.
- Model-less versus model-based algorithms.

As the name indicates, single-solution search-based algorithms organize the search by maintaining a single solution at each step. The transitions between solutions can be done according to different criteria. Examples of single-solution base-search algorithms are the Iterative Descent (or Hill-Climbing) and Simulated Annealing methods presented in previous sections. The difference between Iterative Descent and Simulated Annealing illustrates that in these algorithms, although a single solution is kept in each iteration, multiple evaluations might be needed (e.g, a neighborhood of solutions for some variants of Iterative Descent) in order to decide which is the next solution to be selected.

The best-known family of algorithms that simultaneously uses multiple solutions are population-based evolutionary algorithms such as genetic algorithms (GAs) [1]. The distinguished characteristic of population-based algorithms is that they use selection and recombination operators as a way to identify and exploit partial solutions of high quality ("fitness") in order to conduct an efficient search of the promising areas of the search space. This goal is more difficult to achieve for search-based methods that use single solutions. We will we present GAs in more detail in Section 6.1.

Black-box optimization algorithms do not assume the existence of any information about the function or problem being optimized. On the other hand,

University of Basque Country, Spain

gray-box optimization methods start from some description of the problem characteristics (e.g., some representation of the problem structure) and use this information to implement specialized and more efficient search operators. Most heuristic optimization methods, including those covered in previous chapters are black-box methods.

Model-less optimization methods are those where no global representation of the characteristics or the best solutions or the search progress is created and exploited. They are commonly a choice for problems for which no information is available a-priori. However, in several optimization domains, modeling the characteristics of the promising solutions can contribute to a more efficient search. Model-based search methods are a family of algorithms that implement this approach. Simple GAs usually apply variation operators on one or two solutions and they are considered model-less methods. Other evolutionary algorithms apply selection to identify high fitness solutions, but instead of applying mutation and recombination operators, they learn a model that captures the relevant information from the selected solutions. The model is then used to generate or sample new solutions and the steps of selection, modeling and sampling are repeated in each generation.

While the three criteria considered in this chapter serve to group optimization algorithms into different classes, a variety of other criteria could be used [2]. The rest of this chapter is organized as follows, Section 6.1 discusses the difference between single-solution and multiple-solutions search-based algorithms using the GA as an example. A comparison between black-box and gray-box search-based methods is presented in Section 6.2. A description of optimization techniques that use modeling of the best solutions is presented in Section 6.3. Finally, Section 6.4 provides an example of how to apply different search-based algorithms to the HP protein model, a simple model of protein folding.

6.1 Single solution versus multiple solutions search-based algorithms

Examples of search-based methods that maintain a solution in each iteration, and of population-based algorithms have been presented in previous sections. Here we focus on GAs [1] since they are one of the most successful optimization methods that have achieved excellent results across a wide range of problem domains. GAs are inspired in the theory of natural selection, e.g., survival of the fittest, and on the application of genetic crossover and mutation operators on a population of candidate solutions.

Algorithm 8 shows the pseudocode of a simple GA. In Step 1 of the algorithm an initial set of solutions is generated. Usually, these solutions are generated randomly. However, seeding strategies can be applied to start the search from a pool of high-fitness solutions. In Step 3, the solutions are eval-

uated and subsequently, in Step 4, a subset of solutions is selected according to a predefined selection method. Such method usually gives higher chance to select solutions of better quality. Among the most used strategies for selection are: proportional, tournament, and truncation selection [3].

In Step 5 of Algorithm 8, genetic operators are applied on the selected population. These are operators that are used to generate new solutions based on the solutions selected in the previous step. In the traditional approach, parents chosen from the selected solutions are combined ("mated") using operators called recombination or crossover operators (e.g., one-point or multiple-point crossover). The offspring then go through another operator called mutation operator (e.g., bit-flip mutation for problems with a binary representation). While the crossover operator produces new solutions, mutation operators modify single solutions. Finally, in Step 6, the offspring population and the previous population are used to create the new population. This is called "replacement". Among the most used replacement strategies is elitism, a strategy in which the best solution of the previous population is guaranteed to be part of newly generated offspring. As termination criterion a maximum number of iterations (generations) or some measure of lack of improvement in the solutions are commonly used.

Algorithm 8 Simple genetic algorithm

```
1: pop = generate_population()
2: repeat
3:     fitness = evaluate_population(pop)
4:     selected_pop = select_solutions(pop, fitness)
5:     offspring = apply_genetic_operators(selected_pop)
6:     pop = apply_replacement(pop,offspring)
7: until Stop criterion is satisfied
```

One of the reasons that have been used to explain GA's success is the systematic creation and parallel recombination of partial solutions or building blocks [1]. Considered from that point of view, GAs can be seen as a class of automatic problem decomposition procedures in which information about promising partial solutions is synthesized in multiple ways and refined along generations. However, it has been reported that in many difficult problems blind crossover recombination is not an appropriate choice to exchange information between solutions since partial solutions tend to be disrupted [4].

6.2 Black-box and gray-box search algorithms

Gray-box optimization methods [5, 6] develop strategies to solve the optimization problems for which some *a-priori* information is available, notably, those problems with a "suitable" structure. The underlying assumption is

that knowing this type of "structural" information can not only serve to improve traditional search-based optimization methods, but also to create significantly novel and more efficient approaches.

Several works assume that the problem structure corresponds to the structure of the interactions among the problem variables. For instance, in additively decomposed functions (ADFs) [7], the objective function is the sum of the evaluations of a number of subfunctions defined on subsets of all the variables. If we consider that these subsets of (interacting) variables define the structure of the problem, and this structure is known, then ADFs can be seen as a gray-box optimization problem. Several real-world problems could be included in the class of gray-box, e.g., MAX-kSAT, Ising model, NK-landscape, etc. [6].

Even if the difference between black-box and gray-box optimization problems seems sufficiently clear, there are situations in which information about the problem exists but it is only partial. For instance, we could know which are the groups of related variables where subfunctions are defined, but not the way in which they are related (i.e., the expression for the subfunctions defined in each group). It is also possible that structural relationships are only known for a limited number of groups, i.e., some definition sets of the function are unknown. Therefore, for addressing real-world problems a finer grain definition of the type of available problem information is required. The information can be: 1) About the structural relationships among variables (definition sets); and 2) About the way in which the interactions are expressed within each group (definition of the subfunctions). A detailed categorization of optimization problems based on these two criteria is presented in [8].

Taking into account the type of available problem information, we can describe different types of gray-box search-based optimization methods. For example, *Partition Crossover* is a specialized genetic operator used by some gray-box GAs [9, 6]. It works by analytically decomposing the parents into recombining components which, in turn, allows the evaluation function to be decomposed into linearly separable subfunctions during recombination. In [9], it was applied to challenging combinatorial problems. While there are no apparent reasons to set constraints on the structural characteristics in the general class of gray-box optimization problems, for feasibility and efficiency reasons, gray-box optimizers assume that the structure of the problem is constrained. For instance, it is assumed that the size of any definition subset of the ADF to be optimized is upper-bounded by a parameter k (e.g., *k-bounded pseudo-Boolean* functions as presented in [6]).

Other population-based optimization algorithms that can be considered as gray-box optimizers are factorized distribution algorithms (FDAs) [7]. However, they can be also covered under the umbrella of model-based optimization methods and therefore we cover them in the next section. Gray-box optimizers can be also local-search optimizers. In particular, efficient variants of Iterative Descent algorithms have been proposed that exploit the information

about the problem structure at the time of exploring the neighborhood of a current solution [10, 11].

6.3 Model-less versus model-based algorithms

Model-based search optimizers build a model of the most promising solutions already evaluated by the algorithm and use this model to generate new solutions. The rationale behind this type of algorithm is that the model will be able to capture the patterns shared by multiple high-fitness solutions, and the methods that generate new solutions from the models, i.e., the sampling methods, will be able to reproduce these patterns in the newly generated solutions.

Algorithm 9 Model-based evolutionary algorithm

```
1: pop = generate_population()
2: repeat
3:     fitness = evaluate_population(pop)
4:     selected_pop = select_solutions(pop, fitness)
5:     model = create_model(selected_pop)
6:     offspring = sample_model(model)
7:     pop = apply_replacement(pop,offspring)
8: until Stop criterion is satisfied
```

Algorithm 9 shows the pseudocode of a typical model-based optimizer. Steps 1-4 and 7 are similar to the simple GA described in Algorithm 8. The main difference is in the way the selected solutions are used to generate the new population. As previously discussed, in GAs, recombination and mutation operators are applied. In model-based evolutionary algorithms, the model is built from the selected solutions in a first step, and only after the model has been learned, new solutions are sampled from it.

Different types of models have been used with this type of algorithms, from simple probability models such as histograms [12], to probabilistic graphical models [13], and neural networks [14]. Estimation of distribution algorithms (EDAs) [15, 13], also known as model-building EAs, are perhaps the most popular family of model-based EAs. They represent the most salient patterns of the selected solutions using a graphical structure that encodes the probabilistic dependencies between the variables, and a set of tables of conditional and marginal probabilities of the variables configurations. The particular choice of the model depends on many factors including the problem representation (e.g., discrete, continuous, mixed) and the number and strength of the problem interactions (when they are known). Typical sampling methods used by EDAs comprise Probabilistic Logic Sampling and Gibbs Sampling [16].

While several EDAs learn the graphical structure and the probabilistic tables from data, some algorithms directly derive the model dependencies from the problem structure. They were originally called factorization-based distribution algorithms (FDAs) [7] and can be considered as gray-box optimizers.

Neural networks have been increasingly applied as models for search-based algorithms [17, 18]. In neural networks, information about the problem structure is usually represented by hidden variables or distributed structures. This representation makes interpretability of the model a difficult task. The interpretability of the representation is one important difference between the way neural networks and probabilistic graphical models are used in model-based EAs. Another important difference is that neural networks require specific sampling methods that have not been applied to practical optimization problems to the same extent that those used by graphical models. This fact has led to the investigation of new methods for generating new solutions from neural networks [19, 20].

6.4 Optimization approaches to the HP protein model

A protein can be represented as a sequence of aminoacids. The properties of these aminoacids determine the way the sequence folds into a three-dimensional structure. Hence, an important problem is to predict this structure from the sequence of aminoacids. In this section we illustrate, using a simplified protein model, the way in which some of the search-based methods presented in this chapter can be applied to this problem.

6.4.1 The hydrophobic-polar (HP) model

The HP protein model considers hydrophobic (H) residues and hydrophilic or polar (P) residues. In the linear representation of the protein sequence, hydrophobic residues are represented with the letter H and polar ones with letter P. The structure in which the protein sequence folds is represented as a possible configuration of the residues in a regular lattice. For the sake of simplicity, we consider a square lattice. Given a pair of residues, they are considered neighbors if they are adjacent either in the protein chain (connected neighbors) or in the lattice but not connected in the chain (topological neighbors). Figure 6.1 shows the linear representation of an HP instance and one of its possible configurations.

An energy function that measures the interaction between topological neighbor residues is defined as $\epsilon_{HH} = -1$ and $\epsilon_{HP} = \epsilon_{PP} = 0$. The HP problem consists of finding a configuration of the sequence in the lattice that is a self-avoided path and minimizes the total energy. The question is then,

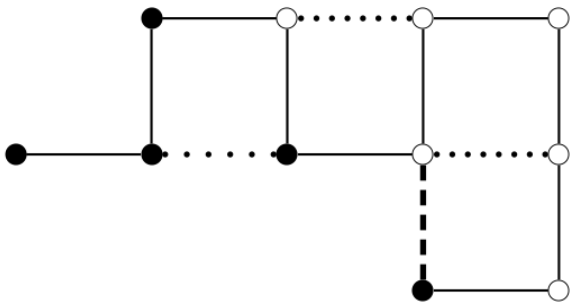


Fig. 6.1: One possible configuration of sequence $HHHPHPPPPPH$ in the HP model. Solid lines indicate a pair of residues neighbors in the sequence. There is one HH (represented by a dotted line with wide spaces), one HP (represented by a dashed line) and two PP (represented by dotted lines) contacts.

how to represent a possible configuration ("solution") of the HP protein sequence.

In order to encode a possible 2D configuration of a sequence, we use the relative encoding [21] which specifies the way in which a walk in a lattice is performed. To represent a walk by means of the relative encoding, an n-dimensional vector $(X_1, X_2, \ldots, X_n)$ is used, where $x_i \in \{0, 1, 2\}$. Creating a path from a vector starts by locating the two first residues in two (usually fixed) adjacent positions of the lattice. Therefore, although variables X_1 and X_2 are part of the representation and correspond to the first two residues, they are meaningless for creating the path.

We can assign an imaginary direction to the segment that joins the two residues in the lattice, the direction goes from the position of the first residue to the position of the second residue. Following this convention, the next residue of the sequence could be located in three possible positions with respect to this segment: in the position ahead of the segment ($x_i = 1$), to the left of this segment and connected to the last residue ($x_i = 0$), or to the right of the segment and connected to the last residue ($x_i = 2$).

The solution encoding the configuration of the sequence shown in Figure 6.1 is $\mathbf{x} = (0, 0, 0, 2, 2, 0, 0, 2, 2, 1, 2)$. x_1 and x_2 are arbitrarily set to zero since residues s_1 and s_2 are always in the same positions. Then, $x_3 = 0$ indicates that residue s_3 will be located left to the initial segment formed by the positions of residues s_1 and s_2, subsequently $x_4 = 2$ indicates that residue s_4 will be located right to the segment formed by the locations of residues s_2 and s_3, and so on.

The computation of the HP energy associated to a given solution $\mathbf{x}$ is not straightforward. Firstly, the configuration has to be constructed simulating

the set of movements encoded by the solution. The energy is computed using the interactions that arise in this configuration. The objective function is the opposite of the energy divided by the number of the sequence's self-intersections.

6.4.2 Model-based approaches to the HP model

The HP model is a focus of research in computational biology [22] and chemical and statistical physics [23, 24]. It has been addressed using local search optimizers [25] and different variants of evolutionary algorithms [26]. We show here the different approaches that can be applied when considering model-based optimizers, in particular, we focus on EDAs that apply two different classes of models as introduced in [27].

Once the problem representation has been set, one of the first questions to consider when addressing real-world optimization problems is whether there exists problem information that could be added to the search. This is what gray-box optimizers do.

For the HP model, we consider two types of information, one is related to the way the 2D configuration is constructed. Since we use a relative representation, it is clear that there is a dependence between the position of residue i and its previous two residues in the sequence. Furthermore, we can expect some dependence to exist also on its previous $k > 2$ residues in the sequence. The other information we know is that some small 2D configurations, where H residues are packed in a small area contribute more to the energy of the proteins.

A key question for model-based optimizer is how to select a model consistent with the interactions of the problem. In the k-order Markov model, the configuration of variable X_i depends on the configuration of the previous k variables, where $k \geq 0$ is a parameter of the model. The joint probability distribution of such a model can be factorized as follows:

$$p_{MK}(\mathbf{x}) = p(x_1, \ldots, x_{k+1}) \prod_{i=k+2}^{n} p(x_i \mid x_{i-1}, \ldots, x_{i-k}) \tag{6.1}$$

where p represents the marginal and conditional probabilities of the variables.

This probabilistic model naturally captures the interaction that arises from the relative encoding representation. Notice, that for a given k, the structure of the interactions relevant for the model is known before the optimization is conducted. However, this is not all the information required by an EDA. In order to implement steps 5 and 6 of Algorithm 9, it is important to define how the model is learned and sampled. For the k-order Markov model, learning consists of computing the marginal and conditional probability tables for the pre-specified interactions using as data the selected solutions. To sample

a solution, first the unconditioned variables are sampled from the marginal probabilities, and then each conditioned variable is sampled, following the order of the sequence.

Another possible approach to the problem is to learn all information about the relevant interactions among the variables directly from the data. A simple model that does this is a tree, where each variable may depend on no more than one variable that is called the parent. A probability distribution $p_{Tree}(\mathbf{x})$ that is conformal with a tree is defined as:

$$p_{Tree}(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid pa(x_i)) \tag{6.2}$$

where $Pa(X_i)$ is the parent of variable X_i in the tree, and $p(x_i \mid pa(x_i)) = p(x_i)$ when $Pa(X_i) = \emptyset$, i.e. when X_i is the root of the tree.

An EDA that uses a tree model [28] could learn the structure of the model from the analysis of the mutual information between every pair of variables. The model uses the marginal and conditional probability tables determined by the structure. For generating a new solution, first the variable corresponding to the root of the tree is sampled, and then each variable is sampled conditioned on the values of its parent in the tree.

6.5 Summary and discussion

In this chapter we have presented different classifications of search-based algorithms that use multiple solutions. We have shown that the decision on whether to apply black-box or gray-box optimization methods should depend on the availability of previous information about the optimization problem and the characteristics of this information. Using a-priori knowledge of the problem as part of the search can make optimization more efficient. Similarly, we have presented the main differences between model-less and model-based evolutionary algorithms and illustrated these differences using the simple GA and EDAs as two paradigmatic examples of each type of methods. At the expense of a higher computational cost, model-based search methods can automatically learn characteristics from the most promising solutions, contributing to a more efficient optimization, and also potentially exploiting and/or unveiling features of the optimization problem.

References

1. D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

2. Jörg Stork, Agoston E Eiben, and Thomas Bartz-Beielstein. A new taxonomy of global optimization algorithms. *Natural Computing*, pages 1–24, 2020.
3. T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.
4. R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In *Parallel Problem Solving from Nature - PPSN V International Conference*, pages 97–106, Amsterdam, The Netherlands, 1998. Springer. Lecture Notes in Computer Science 1498.
5. Francisco Chicano, Darrell Whitley, and Andrew M Sutton. Efficient identification of improving moves in a ball for pseudo-Boolean problems. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 437–444. ACM, 2014.
6. Darrell Whitley. Mk landscapes, NK landscapes, MAX-kSAT: A proof that the only challenging problems are deceptive. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 927–934. ACM, 2015.
7. H. Mühlenbein, T. Mahnig, and A. Ochoa. Schemata, distributions and graphical models in evolutionary optimization. *Journal of Heuristics*, 5(2):213–247, 1999.
8. Roberto Santana. Gray-box optimization and factorized distribution algorithms: where two worlds collide. *CoRR*, abs/1707.03093, 2017.
9. Renato Tinós, Darrell Whitley, and Francisco Chicano. Partition crossover for pseudo-Boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 137–149. ACM, 2015.
10. Francisco Chicano, Darrell Whitley, and Renato Tinós. Efficient hill climber for constrained pseudo-Boolean optimization problems. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 309–316. ACM, 2016.
11. Darrell Whitley, Adele E Howe, and Doug Hains. Greedy or not? best improving versus first improving stochastic local search for MAXSAT. In *AAAI*, 2013.
12. Shigeysoshi Tsutsui, Martin Pelikan, and David E. Goldberg. Evolutionary algorithm using marginal histogram in continuous domain. In *Optimization by Building and Using Probabilistic Models (OBUPM) 2001*, pages 230–233, San Francisco, California, USA, 7 2001.
13. P. Larrañaga, H. Karshenas, C. Bielza, and R. Santana. A review on probabilistic graphical models in evolutionary computation. *Journal of Heuristics*, 18(5):795–819, 2012.
14. L. Marti, J. García, A. Berlanga, and J. M. Molina. Introducing MONEDA: scalable multiobjective optimization with a neural estimation of distribution algorithm. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation GECCO-2008*, pages 689–696, New York, NY, USA, 2008. ACM.
15. H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. In *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lectures Notes in Computer Science*, pages 178–187, Berlin, 1996. Springer.
16. S. Shakya and R. Santana, editors. *Markov Networks in Evolutionary Computation*. Springer, 2012.
17. Shumeet Baluja. Deep learning for explicitly modeling optimization landscapes. *CoRR*, abs/1703.07394, 2017.
18. Malte Probst and Franz Rothlauf. Deep Boltzmann machines in estimation of distribution algorithms for combinatorial optimization. *CoRR*, abs/1509.06535, 2015.
19. Unai Garciarena, Alexander Mendiburu, and Roberto Santana. Envisioning the benefits of back-drive in evolutionary algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.
20. U. Garciarena, J. Vadillo, A. Mendiburu, and R. Santana. Adversarial perturbations for evolutionary optimization. In *International Conference on Machine Learning, Optimization, and Data Science (LOD-2021)*, volume 13164 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 2021.

21. N. Krasnogor, W. E. Hart, J. Smith, and D. A. Pelta. Protein structure prediction with evolutionary algorithms. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1596–1601, Orlando, Florida, USA, 1999. Morgan Kaufmann.
22. A. Gupta, J. Manuch, and L. Stacho. Structure-approximating inverse protein folding problem in 2D HP model. *Journal of Computational Biology*, 12(10):1328–1345, 2005.
23. S. C. Kou, J. Oh, and W. H. Wong. A study of density of states and ground states in hydrophobic-hydrophilic protein folding models by equi-energy sampling. *Journal of Chemical Physics*, 124:244903(1)–244903(11), 2006.
24. H. Abe and H. Wako. Analyses of simulations of three-dimensional lattice proteins in comparison with a simplified statistical mechanical model of protein folding. *Physical Review E. Statistical, Nonlinear, and Soft Matter Physics*, 74:011913, 2006.
25. Hsiao-Ping Hsu, Vishal Mehra, Walter Nadler, and Peter Grassberger. Growth algorithms for lattice heteropolymers at low temperatures. *Journal of Chemical Physics*, 118(1):444–451, 2003.
26. R. Unger and J. Moult. Genetic algorithms for protein folding simulations. *Journal of Molecular Biology*, 231:75–81, 1993.
27. R. Santana, P. Larrañaga, and J. A. Lozano. Protein folding in simplified models with estimation of distribution algorithms. *IEEE Transactions on Evolutionary Computation*, 12(4):418–438, 2008.
28. S. Baluja and S. Davies. Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. In Douglas H. Fisher, editor, *Proceedings of the 14th International Conference on Machine Learning*, pages 30–38, San Francisco, CA., 1997. Morgan Kaufmann.

Part III
Learning Systems

Chapter 7
Introduction

Amanda Cristina Fraga De Albuquerque
Brendon Erick Euzebio Rus Peres
Erikson Freitas de Morais
Gilson Junior Soares
Jose Lohame Capinga
Marcella Scoczynski Ribeiro Martins

Early Artificial Intelligence (AI) studies tried and solved many problems that were considered difficult for humans but relatively easy for computers [1]. These were problems that could be formally described using mathematical rules. Over time, we began to realize that the difficulty did not necessarily reside in these problems, but in those that are easily, even instinctively and intuitively carried out by humans, such as recognizing familiar faces, understanding languages, etc. The point is that human beings, on a daily basis, receive and process huge amounts of information and trying to make computers perform these activities only with pre-defined rules described by us was not viable. Many researchers started to develop techniques where the computer itself, through algorithms, learns to abstract these rules and information on its own by learning from data. This subfield of AI is called Machine Learning. This part of the book is going to introduce several machine learning algorithms.

Two main categories of machine learning algorithms are *supervised* and *unsupervised* learning algorithms. In supervised learning, algorithms have access to data in the form of input-output pairs. The task is to use such data to learn how to predict the outputs given the inputs. For example, one may be interested in predicting whether a patient has cancer (output variable) based on variables describing symptoms that this patient may or may not be displaying (input variables). Or, one may wish to predict the value of a given stock market (output variable) based on its values over the past 5 days (input variables). The function learned by the machine learning algorithm should work well not only for predicting the outputs of previously available data, but also for predicting the outputs of previously unseen inputs. In other words, supervised learning algorithms aim to learn functions able to *generalize* to unseen data. For instance, a function to predict whether patients do or do not have cancer should work well not only for past patients (for which we already know whether or not they have cancer), but also future patients (for

Federal University of Technology - Parana, Brazil

which we do not yet know whether they do or do not have cancer). Similarly, a function to predict the value of the stock market should work not only for past days (for which the stock market value is already known), but also for future days (for which the value of the stock is still unknown).

We can formalize supervised learning as follows: consider a set of examples

$$\mathcal{T} = \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \cdots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\},$$

where $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \in \mathcal{X} \times \mathcal{Y}$ are examples drawn i.i.d. (independently and identically distributed) from a fixed albeit unknown joint probability distribution $p(\mathbf{x}, \mathbf{y})$, $(x_1^{(i)}, x_2^{(i)}, \cdots, x_d^{(i)})^T$ is the vector of input variables of example i, d is the dimensionality of the inputs of the problem, $\mathbf{y}^{(i)} = (y_1^{(i)}, y_2^{(i)}, \cdots, y_{d'}^{(i)})^T$ is the output variable of example i, d' is the dimensionality of the outputs of the problem, and the symbol T indicates the transpose of a vector. Many problems have a single output variable (i.e., $d' = 1$), in which case we express $\mathbf{y}^{(i)}$ directly as a scalar $y^{(i)}$. When $\mathcal{Y}$ is the set of real values, the problem is called a regression problem. When it is a set of categories, it is called a classification problem.

Supervised learning aims at using $\mathcal{T}$ to learn a function $f : \mathcal{X} \to \mathcal{Y}$ able to predict the output values $\mathbf{y} \in \mathcal{Y}$ corresponding to any input $\mathbf{x} \in \mathcal{X}$. This function can be referred to as a predictive model or hypothesis. When dealing with classification problems, it is also common to refer to this function as a classifier. As the data set $\mathcal{T}$ is used to learn this function, it is referred to as the *training set*. Similarly, the examples in this set are referred to as *training examples*. Typically, the learning of f assumes that new (previously unseen) data also follows the joint probability distribution $p(\mathbf{x}, \mathbf{y})$, though some advanced techniques also exist to deal with situations where this may not be the case. When the learned function f is able to predict the outputs of unseen examples very well, it is said that this function *generalizes* well to unseen data.

When dealing with real world problems, it is usually inevitable that the learned function $f(\mathbf{x})$ will make some mistakes, i.e., it will not be able to always correctly predict the output values $\mathbf{y} \in \mathcal{Y}$ corresponding to any input $\mathbf{x} \in \mathcal{X}$. Quite often, when the learning process attempts to create a function f able to perfectly predict the outputs of the examples in the training set $\mathcal{T}$, this model becomes unable to generalize well to unseen examples. This is because the training set $\mathcal{T}$ frequently contains examples with *noise*. These are examples that contain some atypical values for their input or output variables as a result of possible errors in the data collection. When a learning algorithm attempts to predict the outputs of all training examples perfectly, it may end up incorporating such noise, resulting in poor predictions on unseen data. When this happens, it is said that f is overfitting the training examples. Machine learning algorithms thus typically use strategies to avoid overfitting, so that the learned functions f can make the fewest possible mistakes on unseen data.

It is worth noting that, sometimes, the input variables of all training examples are placed in a matrix $\mathbf{X}$ referred to as the design matrix:

$$\mathbf{X} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_d^{(N)} \end{pmatrix} \tag{7.1}$$

Correspondingly, the output variables corresponding to each training example can be placed in an output matrix. When the dimensionality of the outputs of the problem is $d' = 1$, a vector containing the outputs of all training examples can be used instead:

$$\mathbf{y} = \begin{pmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(N)} \end{pmatrix} \tag{7.2}$$

In unsupervised learning, however, the problem has no output, that is, all examples come only with the input data. Therefore, one of the main tasks assigned to these types of algorithms is clustering, where the algorithm learns to find patterns in the input data to separate them into groups called clusters.

Part III-(A) of this book will explain supervised learning algorithms, whereas Part III-(B) will explain unsupervised learning algorithms.

References

1. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
2. Anton Chiang. Ibm deep blue at computer history museum, 2020. Acessado em: 10 de maio de 2021.
3. Santiago Ramon y Cajal. *Comparative study of the sensory areas of the human cortex*. Clark University, 1899.
4. Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.
5. Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 3 edition, 2016.

Part III-(A)
Supervised Learning

Chapter 8
k-Nearest Neighbors

George G. Cabral

Among all supervised classifiers, the family of nearest neighbor (NN) based classifiers are, perhaps, the simplest and most intuitive ones. In a nutshell, given a set of examples $\mathcal{T}$ containing n examples, and considering that we want to find the class of an unlabelled example $\mathbf{z}$, the most basic form of NN (i.e., 1NN) finds the closest example ($\mathbf{x}^{(i)}$) to $\mathbf{z}$, among all n examples in $\mathcal{T}$, and assigns the same class of $\mathbf{x}^{(i)}$ (i.e., $y^{(i)}$) to $\mathbf{z}$.

In order to obtain the closest example, first it's necessary to define a distance, or a similarity, function. The Euclidean distance (Eq. 8.1) stands as the most widely used function. This metric consists in the distance between two examples in an n-dimensional Euclidean space. In Eq. 8.1, the distance between the examples $\mathbf{x}^{(i)}$ and $\mathbf{z}$, located in the j_{th} dimensional space, is being computed.

$$EuclidDist(\mathbf{x}^{(i)}, \mathbf{z}) = \sqrt{(x_1^{(i)} - z_1)^2 + (x_2^{(i)} - z_2)^2 + \cdots + (x_j^{(i)} - z_j)^2} \quad (8.1)$$

In Figure 8.1, the norm of the orange line (i.e., the length/distance between hypothetical examples $\mathbf{x} = [1,1]$ and $\mathbf{z} = [5,4]$), obtained by using Eq. 8.1, is $\sqrt{(1-5)^2 + (1-4)^2}$ resulting in 5 units.

Nevertheless, depending on the domain of the problem, the use of different distance functions might be more adequate. However, these functions must respect the following conditions:

- Non-negativity: $d(x, y) \geq 0$
- Identity: $d(x, y) = 0$ if and only if $x = y$
- Symmetry: $d(x, y) = d(y, x)$
- Triangle Inequality: $d(x, y) + d(y, z) >= d(x, z)$

Department of Computing, Federal Rural University of Pernambuco, BR

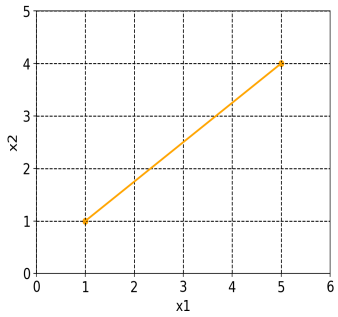


Fig. 8.1: Euclidean Distance illustration between examples located at coordinates (1, 1) and (5, 4).

8.1 Other Distance Metrics

A number of different distances can be used according the domain of the specific application. Some of them are presented in the sequel:

8.1.1 Manhattan Distance

The Manhattan distance [1] (also known as taxicab or cityblock distance) between two examples in j-dimensional space is defined by the sum of the distances in each dimension. Equation 8.2 computes this distance between two examples in the j-dimensional space.

$$ManhDist(x, z) = \sum_{i=1}^{j} \|x_i - z_i\| \tag{8.2}$$

Figure 8.2 shows an example with two unit squares centered at coordinates (1.5, 1.5) and (4.5, 3.5). For this example, the Manhattan distance is $\|1.5 - 4.5\| + \|1.5 - 3.5\|$ resulting in 5 units.

8.1.2 Cosine Similarity

The metric cosine similarity [2] measures the angle between two vectors in the j-dimensional space. The intuition is that the smaller the angle between the two vectors is, more similar they are to each other.

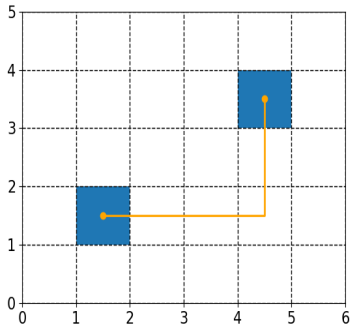


Fig. 8.2: Manhattan Distance illustration between examples defined by coordinates (1.5, 1.5) and (4.5, 3.5).

This metric assumes the value 1 when two vectors are identical and the value -1 when they are completely opposite to each other. These situations take place when the angle between two vectors are 0 and 180 degrees (i.e., the vectors are parallel and have the same direction and the vectors are parallel but have opposite directions, respectively).

Equation 8.3 computes the cosine similarity. In Eq. 8.3, assuming that $\mathbf{x}$ and $\mathbf{z}$ are both vectors, $\mathbf{x} \cdot \mathbf{z} = \mathbf{x}^t \cdot \mathbf{z}$ and $\|\mathbf{x}\| = \sqrt{x_1 \times x_2 \quad \cdots \quad x_n}$.

$$CosSim(x, z) = \frac{x \cdot z}{\|x\| \times \|z\|} \tag{8.3}$$

Among the applications of the cosine similarity is the documents similarity test. Documents can be represented by term-frequency vectors, i.e., each vector position contains the number of occurrences of a given term.

8.1.3 Hamming Distance

The Hamming distance [3] measures the number of characters in disagreement between two words, or two vectors, in general. In other words, the hamming distance is the number of symbols we must change in order to make a vector turn into another. Example: the Hamming distance between the words "**de**ploys" and "**em**ploys" is two. Notice that this distance requires the two words have the same length.

The Hamming distance can be viewed as a special case of the Levenshtein distance where the words must have the same length. In contrast to Hamming Distance, the Levenshtein distance computes the number of substitutions, insertions or deletions necessary to turn one word into another.

8.2 The kNN Algorithm Explained

In contrast to other common machine learning families of algorithms, such as Neural Networks and Decision Trees, the kNN family of algorithms do not possess a learning phase yielding an abstract model representing the problem, instead, these algorithms only store the training examples. Given this fact, kNN-like algorithms are also known as lazy algorithms. In the test phase, once a new unlabelled example $\mathbf{z}$ arrives: (i) its distances (or similarities) to all training examples are computed; (ii) the training examples are sorted accordingly to their distances to $\mathbf{z}$; (iii) the most frequent class among the classes of the k $\mathbf{z}$'s nearest neighbors is then returned (note that for regression problems, the average of the target values from the k $\mathbf{z}$'s nearest neighbors is returned).

Algorithm 10 depicts the test procedure for the original kNN . In Alg. 10, $\mathcal{T}$ is a set of training examples consisting of tuples ($\mathbf{x}$,y), $\mathbf{z}$ is an unlabelled test example and k is the number of neighbors to be considered. In line 1, $neighbors$ is an empty list that shall store tuples containing a distance d (between a training example $\mathbf{x}^{(i)}$ and a test example $\mathbf{z}$) and the label $y^{(i)}$. The loop from lines 2 to 4 fills the $neighbors$ list and in line 6 this list is sorted in ascending order according the stored distances d. The returned class label ($y^{'}$) will be the most frequent class considering the first k items of $neighbors$ (or the average of the k $\mathbf{z}$'s neighbors target values for regression problems, as aforementioned).

Algorithm 10 Simple k Nearest Neighbor

Parameters: $\mathcal{T}$,z,k
Output: $y^{'}$

1: $neighbors \leftarrow []$
2: **for** $\mathbf{x}^{(i)} \in \mathcal{T}$ **do**
3: $\quad d \leftarrow$ distance($\mathbf{x}^{(i)}$,z)
4: $\quad neighbors[i] \leftarrow (d,y^{(i)})$
5: **end for**
6: sort(neighbors)
7: $y^{'} \leftarrow$ mode($neighbors[0:k]$)

For the 1NN ($k = 1$), the Voronoi space for a set of two dimensional examples depicts the boundaries of the areas covered by each training example. Figure 8.3 presents a Voronoi space defined by the depicted blue examples. In this Figure, dashed lines represent infinite boundaries. Since each region is defined by a training example, a new unlabelled example will receive the label of the example that defined the region. By way of illustration, given that the example $\mathbf{p}$ defines the class of the orange region, the same class of $\mathbf{p}$ will be assigned to all unlabelled examples placed in this region.

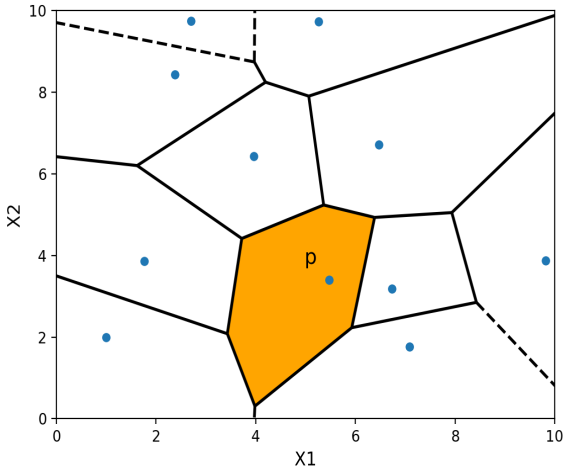


Fig. 8.3: Voronoi space delimiting the 1NN decision boundaries. Dashed lines represent boundaries of infinite length.

Figure 8.4 presents the different decision regions generated by the kNN using different k values applied to a two overlapping classes dataset (blue and orange examples). Firstly, it is important to notice that some examples close to the center of the plot seems to be placed in conflicted regions. These examples, according to the domain of the problem, may be considered noisy or not. Nevertheless, when using $k = 1$ (Figure 8.4.a)), potential noisy examples will have the same importance in building the decision boundaries as any other example (i.e., when $k = 1$ the kNN is completely adjusted to the examples in the training set resulting in an **overfitting** of the data). In Figure 8.4.b) a $k = 3$ was used and the effect of an isolated orange example (placed inside the distribution of blue examples) in the decision process was largely reduced. Also, it is noticeable that in Figures 8.4 c) and d) the decision surface becomes smoother and the importance of noisy examples tends to disappear; as expected for larger values of k.

8.3 Prototype Reduction Schemes

As aforementioned, the training phase of original the kNN consists solely of storing the training examples. On the other hand, the test phase, as shown in Alg. 10, requires the computation of the distance among each training example and the test example to be classified. In addition, it requires the

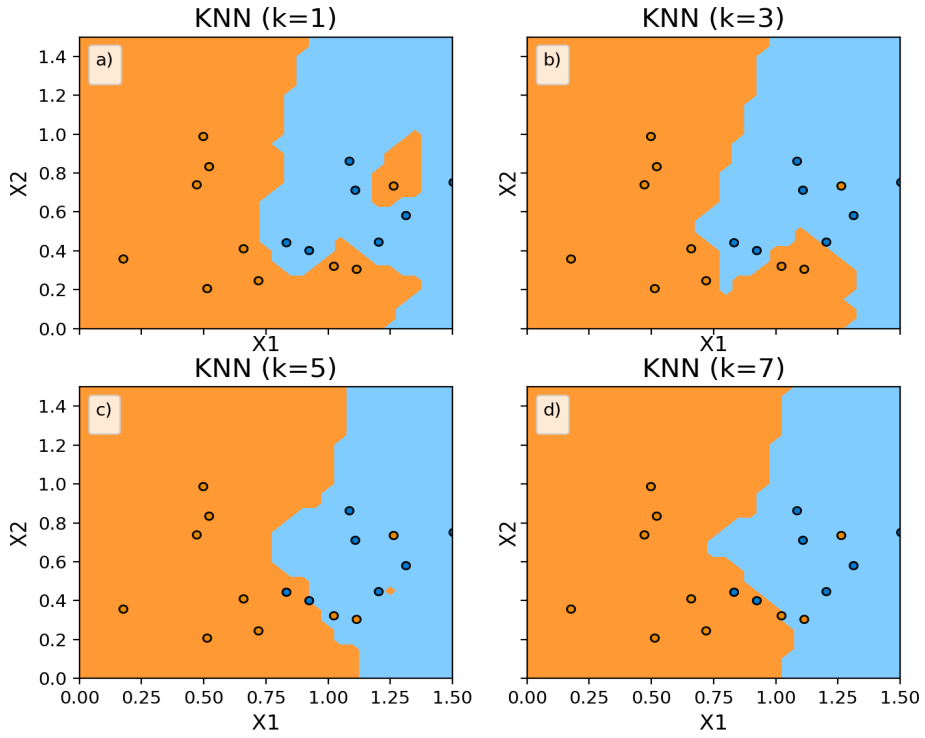


Fig. 8.4: Decision boundaries for different k odd values ranging from 1 to 7.

usage of a sorting algorithm. These two tasks may lead to an unacceptable computational burden in the case of an exceedingly large training set.

With the aim of reducing the computational cost at the test phase, Prototype Reduction Schemes (PRSs) have been used to reduce the size of the training set while keeping, or causing an acceptable loss, in the predictive performance. In sequel, two simple and popular PRSs will be presented: Condensed Nearest Neighbor (CNN) [4] and Nearest Neighbor with Structural Risk Minimization (NNSRM) [5].

8.3.1 Condensed Nearest Neighbor - CNN

The Condensed Nearest Neighbor was initially proposed by P. Hart [4]. The basic intuition behind CNN is to scan all training examples in order to find the

incorrectly classified ones according to the 1NN and to store them in an array S. By doing this, the method is capable of discarding redundant information and, as consequence, keeping only the most representative examples of the problem, i.e., the examples placed in conflict areas.

Algorithm 11 Condensed Nearest Neighbor

Parameters: $\mathcal{T}$
Output: S

```
1: S ← ∅
2: add a random example from T to S
3: while S has changed do
4:    for x^(i) ∈ T do
5:       if 1NN(S,x^(i)) ≠ y^(i) then
6:          S ← S + (x^(i), y^(i))
7:          break
8:       end if
9:    end for
10: end while
```

Algorithm 11 depicts the operation of the CNN method. The aim of the algorithm is to choose the minimum number of training examples that leads to a training error (or empirical risk) equal to zero. Initially, a random training example is added to S (line 2). Notice that, at the end of the CNN procedure, S will store only the most representative examples from $\mathcal{T}$. While S has changed (i.e., new training examples have been added to S) (line 3), all training examples ($\mathbf{x}^{(i)}$,$y^{(i)}$) are classified by an $1NN$ classifier having S as a reference set. If $1NN(S, \mathbf{x}^{(i)}) \neq y^{(i)}$, ($\mathbf{x}^{(i)}$,$y^{(i)}$) is then added to S. If for all $\mathbf{x}^{(i)} \in \mathcal{T}, 1NN(S, x_i) = y_i$, the CNN procedure finishes.

As an illustration of the CNN execution, Table 8.1 shows the coordinates of a two class training set $\mathcal{T}$ comprised of twenty points. The box below presents the execution of Algorithm 11 for the set $\mathcal{T}$ on Tab. 8.1.

Table 8.1: Examples coordinates of a hypothetical two class training set $\mathcal{T}$.

example index	x1	x2	class
1	1.05	0.33	blue
2	1.00	0.50	orange
3	0.58	0.58	orange
4	1.87	0.37	blue
5	0.12	-0.03	orange
6	1.70	0.88	blue
7	0.59	0.20	orange
8	0.24	0.51	orange
9	1.40	0.61	blue
10	1.02	0.36	blue
11	0.70	0.37	blue
12	0.78	0.41	blue
13	0.78	0.04	orange
14	0.74	0.44	blue
15	0.51	0.50	orange
16	0.62	0.46	orange
17	-0.28	0.35	orange
18	0.44	0.74	orange
19	1.30	0.83	blue
20	1.01	0.22	blue

- Initially, $S = \emptyset$. Then it receives a random example, lets say $\mathbf{x}^{(1)}$ ($S = \{(\mathbf{x}^{(1)}, y^{(1)})\}$).
- By running the 1NN on $\mathcal{T}$ and having $S = \{(\mathbf{x}^{(1)}, y^{(1)})\}$ as reference set, example $\mathbf{x}^{(2)}$ is misclassified. $(\mathbf{x}^{(2)}, y^{(2)})$ is then added to S. $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)})\}$.
- Example $\mathbf{x}^{(5)}$ is misclassified, then $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(5)}, y^{(5)})\}$.
- Example $\mathbf{x}^{(6)}$ is misclassified, then $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(5)}, y^{(5)}), (\mathbf{x}^{(6)}, y^{(6)})\}$.
- Example $\mathbf{x}^{(7)}$ is misclassified, then $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(5)}, y^{(5)}), (\mathbf{x}^{(6)}, y^{(6)}), (\mathbf{x}^{(7)}, y^{(7)})\}$.
- Example $\mathbf{x}^{(9)}$ is misclassified, then $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(5)}, y^{(5)}), (\mathbf{x}^{(6)}, y^{(6)}), (\mathbf{x}^{(7)}, y^{(7)}), (\mathbf{x}^{(9)}, y^{(9)})\}$.
- Example $\mathbf{x}^{(11)}$ is misclassified, then $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(5)}, y^{(5)}), (\mathbf{x}^{(6)}, y^{(6)}), (\mathbf{x}^{(7)}, y^{(7)}), (\mathbf{x}^{(9)}, y^{(9)}), (\mathbf{x}^{(11)}, y^{(11)})\}$.
- Finally, example $\mathbf{x}^{(15)}$ is misclassified, then $S = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(5)}, y^{(5)}), (\mathbf{x}^{(6)}, y^{(6)}), (\mathbf{x}^{(7)}, y^{(7)}), (\mathbf{x}^{(9)}, y^{(9)}), (\mathbf{x}^{(11)}, y^{(11)}), (\mathbf{x}^{(15)}, y^{(15)})\}$.

As result, CNN stores only the training examples with indexes $\{1, 2, 5, 6, 7, 9, 11, 15\}$. Figure 8.5 shows the complete original training set (left) and the training set with the reference set S marked (right). Notice that example 2 seems to be a noise. Overall, the main aim of the CNN is to select examples at the classification border, however, depending on the presentation order of the examples in the training set it may select examples placed inside the

class distribution, i.e., examples in non-conflict regions. Additionally, noisy examples are very likely to be added to S since they tend to be misclassified during the training phase.

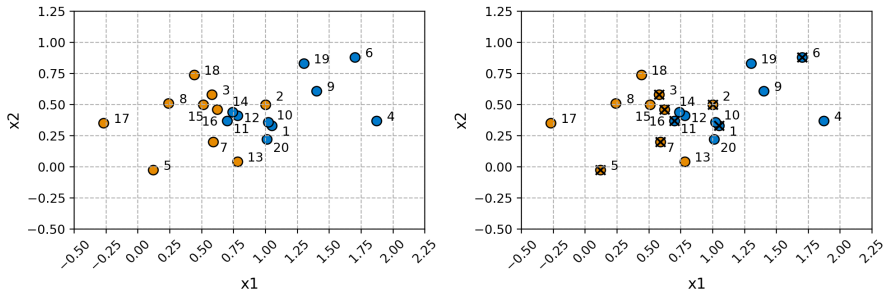


Fig. 8.5: Examples locations of a hypothetical two class training set $\mathcal{T}$ presented in Table 8.1. On the right, the marked examples represent the chosen examples to form the reference set S.

8.3.2 Nearest Neighbor Structural Risk Minimization

NNSRM [5], as CNN, also aims to select examples at the order of the classes, however, NNSRM is not sensitive to the presentation order of the examples in the training set.

NNSRM finds the most significant training examples such that, for all $\mathbf{x}^{(i)} \in \mathcal{T}$, $1NN(\mathbf{x}^{(i)}) = y^{(i)}$, i.e., the R_{emp} (empirical risk or training error) is 0. For a two class problem, let $\rho(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ be the distance between examples $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ so that $y^{(i)} = -1$ and $y^{(j)} = 1$. Let d_k be an ascendant ordered list of distances computed between each two elements of opposite classes for $k = 1, 2, 3 \,..\, \#(y = 1) * \#(y = -1)$.

Algorithm 12 depicts the operation of the NNSRM. While $R_{emp} > 0$ the pair of examples from opposite classes must be found and they must be added to S in case they aren't there. These examples must be found based on the increasing order of distances in d_k.

Algorithm 12 Nearest Neighbor with Structural Risk Minimization

Parameters: $\mathcal{T}$
Output: S

1: $S \leftarrow \emptyset$
2: $k \leftarrow 1$
   {R_{emp} consists in the training error}
3: **while** $R_{emp} > 0$ **do**
4: find (x_i,x_j) such that $\rho(x_i, x_j) = d_k$ and $y_i \neq y_j$
5: **if** $x_i \notin S$ **then**
6: $S \leftarrow S + (x_i, y_i)$
7: **end if**
8: **if** $x_j \notin S$ **then**
9: $S \leftarrow S + (x_j, y_j)$
10: **end if**
11: $k = k + 1$
12: **end while**

Figure 8.6 presents the result of the execution of the NNSRM for the training set of Table 8.1. Based on the result, it is possible to notice that the NNSRM, in contrast to CNN, is not sensitive to the presentation order of the training examples. Nonetheless, the algorithm can still be affected by noisy examples.

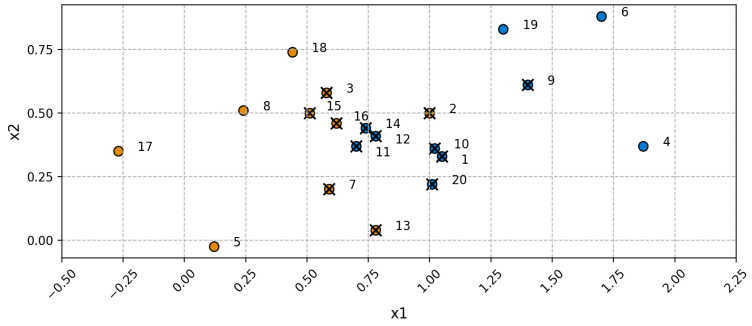


Fig. 8.6: Training examples locations of a hypothetical two class training set $\mathcal{T}$. The examples selected by the NNSRM are marked with a cross.

8.4 Strengths, Weaknesses and Applications of *k*NN

Despite its simplicity, the canonical kNN has several advantages that make it suitable for many domains. In contrast, some characteristics of the method may represent weaknesses for some other domains.

Among the kNN strengths are:

- Simple and intuitive: the training phase consists solely in the storage of the training set in the computer memory. The test phase, basically, consists in computing the distance from each training example to the test example;
- In the case of a proper training set, the kNN is comparable to other state-of-art methods in terms of predictive performance;
- The method does not rely on any statistical assumption of the problem, i.e., it is nonparametric; and
- Easy control of noise robustness via parameter k.

Among the kNN weaknesses are:

- Its computational cost for the test phase is acknowledged as one of main disadvantages;
- Setting a proper value to k is not an intuitive task; and
- For large datasets, as well as the computational cost, the memory requirements for storing the dataset may make prohibitive its use.

The kNN and its variations are general purpose methods and may, virtually, be applied to any problem domain suach as: road traffic prediction [6]; image recognition [7]; software defect prediction [8]; voice recognition [9]; etc.

8.5 Summary and Discussion

The kNN is one of the most well established and popular classifiers. These features are, in part, consequence of its simplicity and its intuitive way to handle the data. Therefore, this classifier is certainly a good starting point to anyone interested in learning computational intelligence (or machine learning, data science, artificial intelligence, etc.). Nevertheless, its popularity resulted as well in a large number of different general or domain specific algorithm versions. This chapter covered the intuition behind the canonical classifier as well as two derived algorithms for data reduction.

8.6 Exercises

All resources necessary for the exact reproduction of the experiments in the exercises below (as well as the exercises' answers) are provided in a python notebook available at: `https://colab.research.google.com/drive/1vzQMXbRJAyrqE7T54G-rOo8TD1bQA1Cs?usp=sharing`

Notice that a basic knowledge about Python language and libraries such as Pandas and Matplotlib is essential.

Question 1: Given the training and testing datasets depicted in Figure 8.7, implement and compute the overall accuracy for the canonical kNN for $k \in \{1, 3, 5, 7, 9\}$.

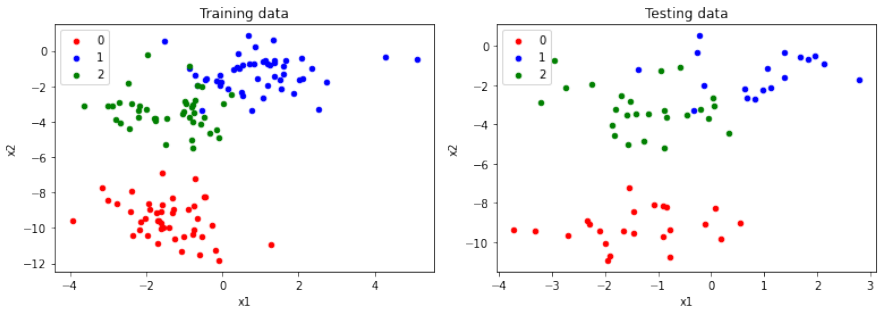


Fig. 8.7: Training (left) and test (right) examples for the exercises.

Question 2: Respecting the order of the training dataset already defined, implement and perform the CNN algorithm and: (i) show the resulting S subset and (ii) the accuracy for a kNN so that $k \in \{1, 3\}$.

Question 3: Considering the training dataset presented in Table 8.1, implement the two class NNSRM algorithm and plot the resulting subset.

References

1. Susan Craw. *Manhattan Distance*, pages 639–639. Springer US, Boston, MA, 2010.
2. Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining concepts and techniques, third edition*. Morgan Kaufmann Publishers, 2012.
3. Mohammad Norouzi, David J. Fleet, and Ruslan Salakhutdinov. Hamming distance metric learning. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, page 1061–1069. Curran Associates Inc., 2012.
4. P. Hart. The condensed nearest neighbor rule (corresp.). *IEEE Transactions on Information Theory*, 14(3):515–516, 1968.

5. A.B. Hamza, H. Krim, and B. Karacali. Structural risk minimization using nearest neighbor rule. In *2003 International Conference on Multimedia and Expo. ICME '03. Proceedings (Cat. No.03TH8698)*, volume 1, 2003.
6. Dongwei Xu, Yongdong Wang, Peng Peng, Shen Beilun, Zhang Deng, and Haifeng Guo. Real-time road traffic state prediction based on kernel-knn. *Transportmetrica A Transport Science*, 16(1):104–118, 2020.
7. Si-Bao Chen, Yu-Lan Xu, Chris H.Q. Ding, and Bin Luo. A nonnegative locally linear knn model for image recognition. *Pattern Recognition*, 83:78–90, 2018.
8. Rinkaj Goyal, Pravin Chandra, and Yogesh Singh. Suitability of knn regression in the development of interaction based software fault prediction models. *IERI Procedia*, 6:15–21, 2014. 2013 International Conference on Future Software Engineering and Multimedia Engineering (ICFM 2013).
9. Lili Chen, Chaoyu Wang, Junjiang Chen, Zejun Xiang, and Xue Hu. Voice disorder identification by using hilbert-huang transform (hht) and k nearest neighbor (knn). *Journal of Voice*, 35(6):932.e1–932.e11, 2021.

Chapter 9
Multilayer Perceptron

Lucas Costa, Márcio Guerreiro, Erickson Puchta, Yara de Souza Tadano, Thiago Antonini Alves, Maurício Kaster, and Hugo Valadares Siqueira

Artificial Neural Networks (ANNs) are computationally intelligent systems inspired by higher organisms' nervous system behavior. They are based on processing units called artificial neurons, capable of calculating mathematical functions [1]. Through these neurons and their connections, ANNs can learn to process information to produce the expected output [2]. ANNs can be seen as general tools for solving different problems. Thus, they are often applied in tasks such as pattern classification, data mining, regression/approximation of functions, and information processing, being useful in several areas of knowledge [1]. This chapter will explain the MultiLayer Perceptron (MLP), the best-known ANN architecture. For that, the concept of artificial neuron, the primary processing structure of an MLP, will first be introduced.

9.1 Artificial Neuron

The artificial neuron is inspired by the biological neuron. In biological systems, neural impulses are received through dendrites and processed in the cell body. Depending on the result of the integration of the received signals, if they are higher than a critical threshold, the neuron may or may not produce a new impulse (action potential), which will, in turn, be transmitted to other neurons connected to its axon terminals [2]. The axon union with dendrites is called a synapse, which works as a valve controlling the flow of information (impulses) between neurons. The synapse is variable, hence, with the ability to adapt and learn.

Figure 9.1 shows the generic artificial neuron scheme used in ANNs. This device receives a set of $\mathbf{x} = [x_1, x_2, ..., x_I]$ inputs, which may come either from other neurons or correspond to the input variables of the problem. When they correspond to the input variables of the problem, $I = d$ is the dimensionality

UTFPR - Universidade Tecnológica Federal do Paraná

of the problem. It then processes this information, and responds with a $\hat{y}_j$ signal, where j the current neuron. This response, which is a transformation of the received inputs, is either propagated to other neurons or given as the output of the ANN.

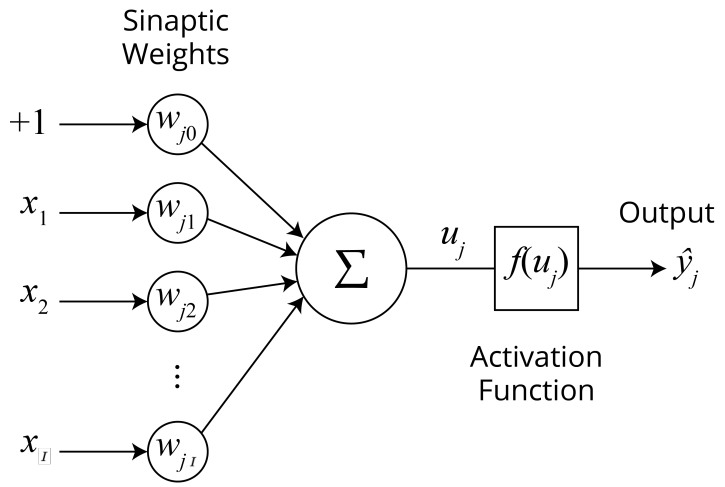


Fig. 9.1: Scheme of an artificial neuron.

The processing performed by neuron j consists in weighting the input signals $x_i, i = 1, 2, ..., I$ and a fixed value bias signal $x_0 = +1$ by the weights w_{ji} of this neuron. Then, the sum of the weighted input signals u_j (a.k.a. activation value) passes through an activation function $f(\cdot)$, generating the output $\hat{y}_j$. The mathematical representation of this model is given by Equation 9.1:

$$\hat{y}_j = f(u_j) = f\left(\sum_{i=0}^{I} w_{ji} x_i\right). \tag{9.1}$$

An activation function $f(\cdot)$ defines the output of a given neuron, given the activation value u. Several functions have already been used or developed. This section gives some examples of such functions.

The linear activation function is a simple function that can be described by equation 9.2:

$$f(u) = \alpha u, \tag{9.2}$$

where α is a real number.

Another standard function is the signal, also known as *Heaviside* [1], which can respond in a binary $\{0, 1\}$ or bipolar $\{-1, 1\}$ way. The former is the one used in the McCulloch and Pitts neuron [4], as shown in Equation 9.3:

$$f(u) = \begin{cases} 0, & u \leq 0 \\ 1, & u > 0 \end{cases}. \tag{9.3}$$

A group of functions most commonly used in the hidden layers of an MLP are the sigmoid functions [1, 2]. Their plot has the peculiar shape of an "S". They represent a balance between linear and nonlinear behavior, which can be obtained from several functions, such as the logistic function and the hyperbolic tangent [5]. The logistic function is defined by Equation 9.4:

$$f(u) = \frac{1}{1 + e^{-u}}. \tag{9.4}$$

The hyperbolic tangent is defined by Equation 9.5:

$$f(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{u}}. \tag{9.5}$$

The wide use of S-Shaped functions is also due to some key features [6]:

- This kind of function is continuous and differentiable at all points, allowing learning with popular derivative-based learning algorithms such as gradient descent;
- They have output saturation, which can prevent the output signal of each neuron from diverging;
- It is possible to use them to create different mappings since these functions have an almost linear character in the region around the origin, while tt the same time, close to saturation, they are strongly nonlinear.

Recently, with the advancement of ANNs studies and, more specifically, *deep learning* methods, new activation functions have been developed to solve vanishing gradients or flat surfaces [7], which are problems that learning algorithms such as gradient descent may face and that may prevent them from being able to successfully learn the weights of the neurons.

One new activation function, the Rectified Linear Unit or ReLU [8], can be expressed by Equation 9.6:

$$f(u) = \begin{cases} u, & u > 0 \\ 0, & u \leq 0 \end{cases}. \tag{9.6}$$

If u is greater than zero, the output will equal the input. So, the ReLU function is similar to the linear activation function (Equation 9.2) for values greater than zero.

Another newly developed function is the Exponential Linear Unit or ELU [9]. Its definition is expressed by Equation 9.6:

$$f(u) = \begin{cases} u, & u > 0 \\ \alpha(e^{u} - 1), & u \leq 0 \end{cases}, \tag{9.7}$$

The definition for u lower than zero allows problems with negative values.

9.2 MLP Architecture

One of the best-known ANN architectures is the MultiLayer Perceptron (MLP), which structurally generalizes the artificial neuron called Rosenblatt perceptron [10] – an artificial neuron that uses the Heaviside activation function. As demonstrated by [11], this ANN has universal approximation capability: an MLP can approximate any continuous, bounded, differentiable, nonlinear function with defined inputs in a compact space with arbitrary precision. This is possible by the additive composition of base functions, which, for MLP, are ridge functions. However, this theorem does not specify the amount of artificial neurons required, nor does it define any method for adjusting the value of the weights so that the optimal configuration of the network is guaranteed.

MLPs organize neurons in several layers chained together, where each layer is the arrangement of parallel neurons. Neurons in a given layer do not communicate with each other, and only send their output signals forward, which is known as a *feedforward* structure. MLPs contain an input layer, one or more hidden layers and an output layer. Figure 9.2 shows an example of possible MLP structure with one hidden layer for a problem with d-dimensional input variables and one output variable. As the neurons are organised in layers, we use superscript numbers in our notation to identify their layer. The first layer ($l = 0$) is the input layer, whose neurons receive as inputs the input variables of the problem. Each neuron in this layer is a special neuron that simply feeds the value of a given input feature to the nodes in the hidden layer, i.e., $\hat{y}_j^{(l=0)} = x_j$, where j. Neurons in the hidden layers ($0 < l < L$) and output layer ($l = L$) are standard artificial neurons that receive as inputs the outputs of the neurons from the previous layer. The last layer is the output layer, whose neurons produce the values of the output variables of the problem.

One or more hidden layers can be used. The hidden layers are responsible for mapping the input signal in a nonlinear way into another space, according to the demand of the problem. As the combination of linear functions is also a linear function, the activation functions of the hidden nodes is usually not a linear function.

When dealing with regression or binary classification problems, a single output node is typically used. When dealing with multi-class classification problems, one output node is used to correspond to each class. After the training process of the MLP is complete, predictions are given by mapping the numeric values given by the output nodes into classes when dealing with classification problems. In particular, sigmoid logistic activations are typically used in the output node for binary classification problems. If the output value is larger than 0.5, a given class could be predicted. If the output value is smaller than or equal to 0.5, the other class could be predicted.

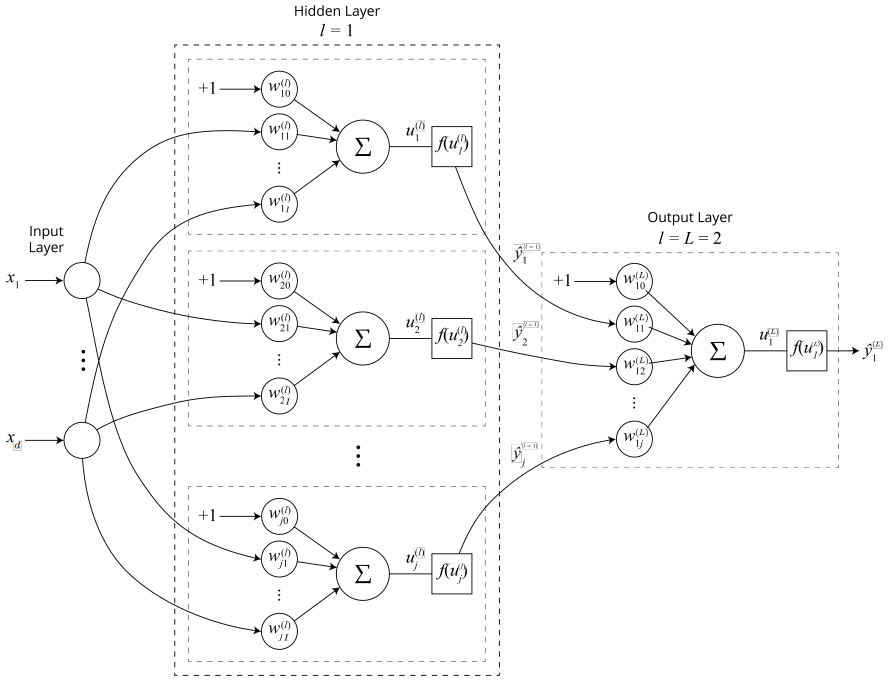


Fig. 9.2: Example of MLP.

9.3 Training

Several training algorithms are aimed at adjusting the weights of an MLP. For that reason, the weights of the MLP are also referred to as "parameters" to be learned by the training process. This chapter will focus on the traditional gradient descent method with the famous backpropagation algorithm [12].

The training process of an MLP is typically supervised, relying on a training set containing labelled training examples with the values of the input variables and the corresponding desired values of the output variables. When using backpropagation, training consists of two steps: forward and backward. In the forward step, the network weights do not change; they are fixed, and the input data (from the training set) is propagated from the first to the last layer to obtain the network output. Then, an error signal is produced by comparing the obtained output with the desired one. After that, the gradient vector of the error function is calculated. Based on it, the backward step applies the gradient descent optimization method from the last to the first layer, such that the weights of the neurons are adjusted in the opposite direction of the gradient, i.e., in the "steepest descent" direction that most decreases the error function [1]. The forward and backward propagation of the whole

training set are applied iteratively until some sopping criterion is reached [2]. Each iteration through the whole training set is typically referred to as an epoch. The most used error function for this task is the Mean Squared Error (MSE). In Section 9.3.1, this procedure is presented in a mathematical format.

It is worth noting that this procedure is nothing more than an optimization algorithm being applied to an unconstrained nonlinear optimization problem. Even though this chapter describes the use of gradient descent to optimize the weights, any unconstrained nonlinear optimization method of 1st or 2nd orders could be used, such as the gradient and Levenberg-Marquardt methods. Note that such a premise is not based on biological inspiration.

There are different learning methods to handle the adjustment of network weights based on gradient descent [1, 13]:

- Batch learning: weight adjustments occur after all training examples are presented to the network, at the end of each epoch during the training;
- Online learning: weight adjustments occur after the presentation of each training example, turning the search for optimal weights into a stochastic search since the examples are presented in random order;
- Mini-batch learning: this an intermediate process between batch and online learning, where weight updates are performed after fixed size batches of examples from the training set are presented.

9.3.1 Backpropagation

This section follows the definitions of Simon Haykin [1] to formalize the backpropagation algorithm. The online learning version of Backpropagation is presented.

Before starting the learning process, the weights of the MLP are typically initialized uniformly at random in the interval $[-1, +1]$. If there is some previous knowledge about the mapping process being worked on, it is possible to initialize the weights with pre-fixed values.

The first step in applying Backpropagation is the propagation of the input signal passing through the input and hidden layers until reaching the output layer (forward step). Consider a given training example $(\mathbf{x}(n), \mathbf{y}(n))$ being introduced in a given iteration n, where $\mathbf{x}(n)$ are the input variables fed to the ANN input layer, and $\mathbf{y}(n)$ is the desired output. The input layer simply feeds the values of the input features to the hidden layer. From the hidden layer to the output layer, the input values propagation is done by calculating the $u_j^{(l)}(n)$ induced by neuron j in layer l (Equation 9.8):

$$u_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n)\hat{y}_i^{(l-1)}(n), \tag{9.8}$$

where the value $w_{ji}^{(l)}(n)$ is the weight connecting neuron j of layer l to neuron i from layer $l-1$, and $\hat{y}_i^{(l-1)}(n)$ is an input of neuron j of layer l corresponding to the output signal of neuron i of the previous layer $l-1$ (or the bias value $+1$, if $i = 0$).

The output of neuron j from layer l can be defined by Equation 9.9:

$$\hat{y}_j^{(l)}(n) = f_j(u_j^{(l)}(n)), \tag{9.9}$$

If the neuron j is in the first hidden layer ($l = 1$), then Equation 9.10 is used:

$$\hat{y}_j^{(0)}(n) = x_j(n), \tag{9.10}$$

where $x_j(n)$ is the jth element of the input vector $\mathbf{x}(n)$.

The second step of the backpropagation (backward) is then performed to propagate the error backward through the MLP, using it to adjust the weights based on gradient descent. The error for each neuron in the output layer is calculated according to Equation 9.11:

$$e_j(n) = \hat{y}_j^{(L)}(n) - y_j(n), \tag{9.11}$$

where $y_j(n)$ is the jth output of the desired output vector $\mathbf{y}(n)$.

Afterwards, the local gradients δ of the network are calculated. The local gradient of the output layer L is obtained by Equation 9.12:

$$\delta_j^{(L)}(n) = e_j(n) f_j'(u_j^{(L)}(n)), \tag{9.12}$$

and those of the other layers are following Equation 9.13:

$$\delta_j^{(l)}(n) = f_j'(u_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n), \tag{9.13}$$

where $f_j'(\cdot)$ is the derivative of the activation with respect to the activation value.

Finally, the adjustment of the weights of the layer l takes place according to the generalized delta rule of Equation 9.14:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \eta \delta_j^{(l)}(n) \hat{y}_i^{(l-1)}(n). \tag{9.14}$$

One way to improve the weight adjustment process and avoid instabilities is to modify Equation 9.14 by adding a term called *momentum*, changing Equation 9.14 to Equation 9.15:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[\Delta w_{ji}^{(l)}(n-1)] + \eta \delta_j^{(l)}(n) \hat{y}_i^{(l-1)}(n), \tag{9.15}$$

where α is a generally positive constant and $\Delta w_{ji}^{(l)}(n) = w_{ji}^{(l)}(n) - w_{ji}^{(l)}(n-1)$. The momentum term can reduce instability in the weights by enabling the previous change in the weight to influence the current change.

The forward and backward steps are repeated, presenting all training data again until some stopping criterion is reached. Backpropagation with gradient descent can be summarized in Algorithm 13.

Algorithm 13 Backpropagation.

Parameters: max_it, η, α, $\mathcal{T}$
Output: $\mathbf{w}$

Initialize $\mathbf{w}(1)$
n $\leftarrow 1$
epoch $\leftarrow 1$
while epoch $<$ max_it **do**
  Shuffle the order of the training data in $\mathcal{T}$
  for each example in $\mathcal{T}$ **do**
    Refer to this example as $\mathbf{x}(n), \mathbf{y}(n)$
    for l from 1 to L **do**
      for each neuron j in layer l **do**
        for each input i of neuron j **do**
          $u_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n)\hat{y}_i^{(l-1)}(n)$ {Equation 9.8}
        end for
      end for
    end for
    for each neuron j of layer L **do**
      $e_j(n) \leftarrow \hat{y}_j(n)^{(L)} - y_j(n)$ {Equation 9.11}
    end for
    for l from L down to 1 **do**
      for each neuron j in layer l **do**
        if $l = L$ **then**
          $\delta_j^{(L)}(n) \leftarrow e_j(n) f_j'(u_j^{(L)}(n))$ {Equation 9.12}
        else
          $\delta_j^{(l)}(n) \leftarrow 0$
          for each neuron k in layer $l+1$ **do**
            $\delta_j^{(l)}(n) \leftarrow \delta_j^{(l)}(n) + f_j'(u_j^{(l)}(n))\delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n)$ {Equation 9.13}
          end for
        end if
        for each input i of neuron j **do**
          $w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[\Delta w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n)\hat{y}_i^{(l-1)}(n)$ {Equation 9.15}
        end for
      end for
    end for
    $n \leftarrow n+1$
  end for
  epoch $\leftarrow$ epoch + 1
end while

The algorithm receives the learning rate η, the momentum constant α, the training data stored in the set $\mathcal{T}$, and a "max_it" variable, which is the stopping criterion of the algorithm. The training stops when it reaches the defined number of iterations. The MLP's weights are then returned. Another common stopping criterion would be to calculate the error at the end of each iteration and check if it is at an acceptable level: if so, the training is stopped. Finally, note that it is necessary to define the number of artificial neurons in the hidden layer a priori.

9.3.2 Pratical Aspects of the Backpropagation Application

The goal of training ANNs is to reach a level of generalization such that it is possible to make correct inferences about data the system does not know. However, the ANN may not perform well with new and unknown data (test set), which were not used during the training phase, characterizing the overfitting phenomenon.

Then, the network's response quality needs to be evaluated during the learning process. A standard procedure separates the available data into three sets: training, validation, and test. As stated, the training set is used to adjust ANN weights. After changing the weights at the end of each training period, the network uses the validation set, which has data not used in training and calculates the output error for this set, keeping the weights fixed. The validation error tends to decrease over the iterations and reach an inflection point when it rises. Thus, an optimal weights array must be saved and overwritten whenever the validation error is reduced during the iterations. When reaching the stopping criterion, the final set of weights of the network will be those saved in the optimal weights array.

The above process is known as cross-validation holdout. Variations can be found in the literature, such as K-fold and leave-one-out [1, 2]. Furthermore, varying the number of neurons (topology) in the hidden layers and comparing the error achieved in the validation set is a strategy to define the number of neurons [14].

A test set whose data were separated from the entire training and validation process is used to assess the final performance of the model. The test set error is expected to be a percentage higher than the training error, although it needs to be an acceptable value. It is a way to measure overfitting even after cross-validating, which is desirable.

Note that the three sets must be meaningful, containing examples that cover all subspaces that were covered in the original data set. There is no rule for dividing the samples, but the literature frequently uses 70% for training, 15% for validation, and 15% for test. Another possibility is 50%, 25%, and 25%, respectively [1].

It is known that the error-based cost function is multimodal (having multiple minima) for real problems and training algorithms are essentially local optimizers. As mentioned, generating the network weights randomly, respecting a uniform distribution in the interval $[-1, +1]$, is the most common way to initialise the neural network. In this way, initializing the weights to different values leads to different weights being learned. As different learned weights can lead to different training and validation errors, the procedure used to evaluate an ANN approach usually runs it several times (typically at least 30 times), so that the dispersion of results can be evaluated [17].

Finally, normalizing the data so that they are in the range of the nonlinearity of the activation function is recommended. For example, the data should be normalized over the interval $[0, +1]$ when using the sigmoid function. Similarly, if a hyperbolic tangent option is used, the desired range is $[-1, +1]$. If we deal with a nonlinear mapping or prediction problem, the normalization needs to be reversed, changing the final results to the original data magnitude.

9.4 Exercises

1. The truth table for the OR logic gate is shown below:

Inputs		Outputs
x_1	x_2	y_{OR}
0	0	0
1	0	1
0	1	1
1	1	1

   For each gate, one could consider a system with desired inputs and outputs. Assume that the inputs are numeric, but treat the output values 0 and 1 as categorical values. Manually create an MLP, including the values of its weights, that is able to perform the OR operation.
2. Haberman's Survival Data Set contains cases from a study conducted at the University of Chicago Billings Hospital between 1958 and 1970 on the survival of patients undergoing surgery to treat breast cancer. The attributes are:

   a. Age of the patient when the procedure was performed;
   b. Year the process took place;
   c. Number of positive nodules detected;
   d. Survivor status, where 1 means the patient survived for five years or more, and 2 represents the patient died before completing five years of surgery.

This database is one of several available ones from the UCI Machine Learning Repository [18]. Use the first three attributes as input data for training and the last one as the desired output (surviving or not). Train an MLP and discuss its performance. Vary the number of epochs, hidden nodes, learning rate and momentum, and discuss the impact of that on the results. You may use any existing software to run the MLP. For instance, you may try WEKA: `https://www.cs.waikato.ac.nz/ml/weka/`.

9.5 Exercise Answers

1. There are many possible answers to this question. One could use an MLP with a single hidden neuron using a sigmoid logistic activation function, with weight of 2 for each of the inputs and weight of -1 for the bias. Interestingly, as this problem is very simple, the hidden node would suffice for providing predictions, without the need for any output node. However, as an MLP should have both a hidden and output layer, we could have a single output neuron using a linear activation function with a weight of 0 for the bias, a weight of 1 for the output of the hidden node, and $\alpha = 1$. Such neuron would simply map the output of the hidden node to the output of the MLP.
2. This is a very open question. Please try it out to see how well different configurations of the MLP can perform!

References

1. Simon Haykin. *Neural Networks and Learning Machines*. Prentice Hall, Hamilton, 3rd edition, 2008.
2. L Castro. Fundamentals of Natural Computing - Basic Concepts, Algorithms, and Applications. In *Chapman and Hall / CRC computer and information science series*, 2006.
3. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.
4. Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943.
5. Alan Jeffrey and Hui-Hui Dai. *Handbook of Mathematical Formulas and Integrals*, volume 44. Elsevier, Burlington, 4th edition, 2008.
6. Anil Menon, Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. Characterization of a class of sigmoid functions with applications to neural networks. *Neural Networks*, 9(5):819–835, 1996.
7. Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. *Neural Networks*, 6(3):1–19, 1988.
8. Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, volume 28, 2013.

9. Djork Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). In *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pages 1–14, 2016.
10. F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.
11. G Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.
12. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
13. Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7700 LECTU:437–478, 2012.
14. Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*, volume 103. Springer New York, 2013.
15. Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference of Artificial Intelligence*, 1995.
16. Brian Ripley. *Pattern Recognition and Neural Networks*. Cambrige University Press, 8th edition, 2005.
17. Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.
18. Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

Chapter 10
Deep Learning

Amanda Cristina Fraga De Albuquerque, Brendon Erick Euzebio Rus Peres, Erikson Freitas de Morais, Gilson Junior Soares, Jose Lohame Capinga, and Marcella Scoczynski Ribeiro Martins

Deep Learning is part of machine learning methods based on artificial neural networks. Part of the theoretical basis underlying Deep Learning initially emerged as models for understanding the human learning process, that is, how the brain works. Thus, these theories are related with Deep Learning that has grown the most in recent years [1]. Deep learning methods have achieved excellent performance over traditional machine learning methods, mainly due to the development of the area, but also by the increase in computational power and the amount of available data [2].

One of the most well known applications of Deep Learning are computer vision problems. Therefore, different from other chapters that discuss potential applications at the end of the chapter, this whole chapter will focus mainly on deep learning for computer vision problems. Computer vision is a field that seeks to reproduce some of the human capabilities through autonomous systems. The main aim of computer vision is to enable computers to perform functions similar to human vision, being able to receive visual data and perform its processing. Examples of computer vision problems include face recognition and object recognition. This field has been using Artificial Intelligence (AI) extensively. The performance improvements in computer vision are strongly related with the evolution of the field of machine learning.

Traditional computer vision techniques were almost entirely pipelined by hand, where the features to be used as input variables were obtained through manually designed methods. This made such techniques more difficult to adapt to different tasks. With the emergence of deep learning, it became possible to use neural networks to automatically learn features to be used. This has helped Deep Learning to outperform other existing methods on several problems such as computer vision problems.

Among the many deep learning architectures, the Convolutional Neural Networks (CNN) is one of the most widely used as it is very similar to a

Federal University of Technology - Parana, Brazil

conventional Multilayer Perceptron (MLP) introduced in a previous chapter of this book.

10.1 Convolutional Neural Networks (CNN)

As with the MLPs, CNNs are also formed by neurons and connections between them to build a model. Backpropagation is also frequently used to learn the weights of the connections (parameters). However, instead of connecting all neurons of a given layer to all neurons of the previous layer like the MLPs, CNNs has some layers connecting a neuron to a limited number of neurons of the previous layer [3]. This helps to reduce the amount of memory and computations required by the CNNs. Moreover, instead of using a separate weight to each connection between neurons, some layers of the CNNs share the same matrix of weights for the incoming connections of all neurons of the same layer [3]. This also helps to reduce the amount of required memory to store the model. Each of such matrices is referred to as a *kernel* or *filter*.

The next subsections explain the types of layer that are used to compose the CNN, namely convolution, pooling and fully connected layers. There are different possible architectures (ways to arrange layers) for CNNs, where different numbers of each type of layer may be used. Typically, the first layer is a convolution layer, which may be followed by other convolution layers or by pooling layers. The last layers are typically fully connected layers, which are similar to the MLP layers. An example of this is shown in Figure 10.1. Different examples of CNN architectures are given in Section 10.1.5.

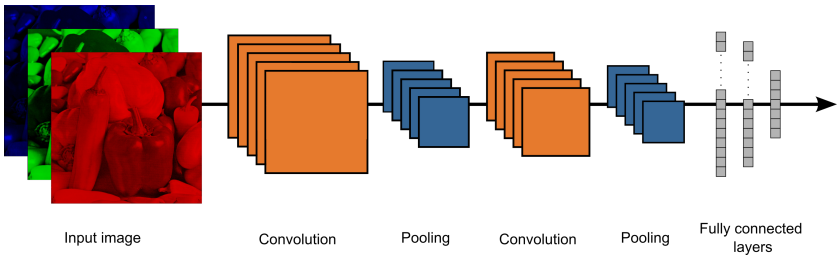


Fig. 10.1: Example of CNN architecture.

10.1.1 Convolution Layers

The operation that gives name to this artificial neural network is the convolution, which is performed in the convolution layers. This operation systematically applies the kernel to an input image to generate a processed output image. Layers that perform the convolution operation are referred to as convolution layers. The processed image produced by a convolution layer can be seen as summarising the presence of features automatically detected in the input image [4] and is commonly referred to as a feature map.

In Deep Learning literature and libraries, it has become common to call this operation as convolution [1], though mathematically this operation is known as correlation. The convolution operation $g(x, y)$ applied to a pixel with coordinates (x, y) is defined in Equation 10.1:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} W_{s,t} F_{x+s,y+t} \tag{10.1}$$

where W is a kernel, which is a matrix of numbers, usually with odd square size $2a+1$ by $2b+1$ (to facilitate operations); a and b are pre-defined values to define the size of the kernel and determine the possible values of its coordinates; and F represents an image in matrix format. For convenience, the first coordinate of the matrix F is set as $(-a, -b)$.

Figure 10.2 presents an example of correlation step taking place. In this figure, $a = b = 1$, and the indices of the elements of the matrices start with -1 instead of starting with 1. For instance, the coordinate x of the matrix F goes from -1 to 3. The convolution operation is being applied to pixel $F_{0,0}$. The operations performed in this convolution step are:

$$\begin{aligned} g(0,0) = \sum_{s=1}^{2} \sum_{t=1}^{2} W_{s,t} F_{0+s,0+t} = \\ w(-1,-1)f(-1,-1) + w(-1,0)f(-1,0) + w(-1,1)f(-1,1) \\ +w(0,-1)f(0,-1) + w(0,0)f(0,0) + w(0,1)f(0,1) \\ +w(1,-1)f(1,-1) + w(1,0)f(1,0) + w(1,1)f(1,1) \\ = (-1)\cdot 5 + (-2)\cdot 7 + (-1)\cdot 0 \\ +0\cdot 6 + 0\cdot 0 + 0\cdot 1 \\ +1\cdot 6 + 2\cdot 2 + 1\cdot 2 = -19 - 0 + 12 = -7 \end{aligned} \tag{10.2}$$

In Figure 10.2, a single matrix F corresponding to an image is being used. In a real world application, this would correspond to a grayscale image. However, most of the time images will be following the RGB (Red, Green Blue) model, where three matrices will be present, each one representing a color channel. These three matrices represent an image in colour. You may also

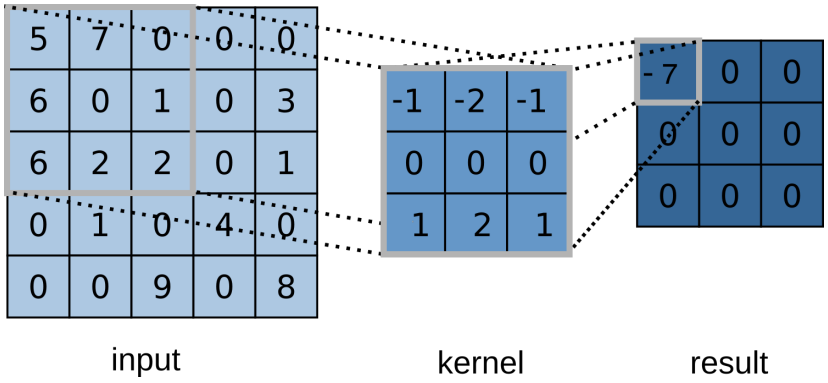


Fig. 10.2: Convolution of a 5x5 sized image with a 3x3 sized kernel and its result.

refer to these three 2-dimensional matrices together as a single 3-dimensional matrix. When using RGB images, there will be one kernel for each channel. You may refer to these three kernels together as a 3-dimensional filter. Using this terminology, applying a 3-dimensional filter to a 3-dimensional image will result in an image represented by a single matrix corresponding to the sum of the three matrices obtained by applying each kernel to its corresponding 2-dimensional matrix. Figure 10.3 depicts an example of convolution for an RGB image.

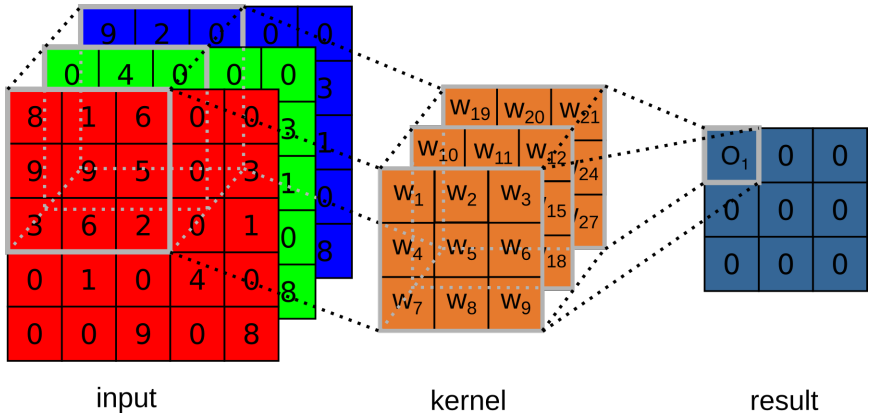


Fig. 10.3: Convolution of a 5x5x3 sized image with a 3x3x3 sized kernel and its result.

In addition, one may wish to apply more than one filter to a given image. This will result in multiple images being produced as output. An example

of that is given in Figure 10.4, which depicts a representation of a convolution layer with three filters. For each filter, there is an output matrix and, consequently, as a final result a dataset where the number of depth layers (also known as feature map, illustrated by the three matrices in orange) will correspond to the number of filters applied to the input. This output can then be sent forward on the network, going through more convolutions and having more features extracted.

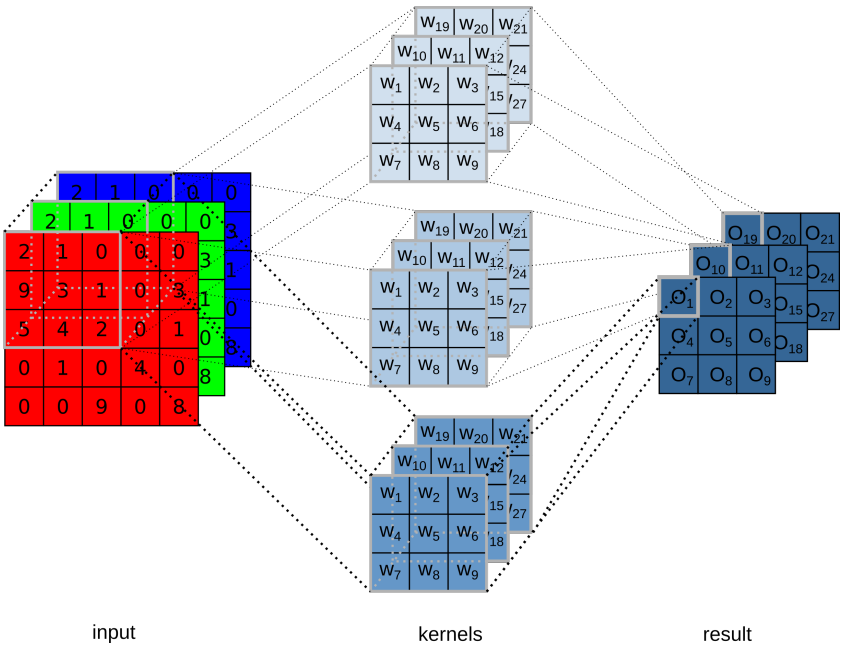


Fig. 10.4: Convolution of a 5x5x3 sized image with three 3x3x3 sized kernel and its result.

These examples also serve to show us one of the features of convolution that makes it a good choice for working with images, what is called feature sparse iterations (also known as sparse connectivity) [1]. This attribute highlights the fact that each output unit, or pixel, is connected to only a fraction of the input units. For example, in Figure 10.4, each output pixel is connected to a region of the 75 input pixels, through a 3x3x3 kernel. This is very useful, as our image can have millions of pixels, and by using smaller sized kernels, we will be able to detect small features such as edges, corners, etc [1]. In the convolution layers, the learning process consists in learning the filters. This results in few parameters to learn and store. Conversely, in a simple neural network, as we saw in the MLP chapter, an image at the input means that

each pixel would be connected to each neuron in the next layer, thus resulting in an excessively large network.

As mentioned earlier, another important feature of CNNs is the ability to share the parameters to be learned, since the same filter is applied to different regions of the image using the same values. This is unlike a neural network without convolution layers, where we have a matrix with weights that are used for only one connection. The sharing of parameters gives us another feature, which is the invariance to translation, i.e., if we move the position of an object in the input image, its representation will also be moved in the resulting image [1].

10.1.1.1 Padding

In the convolution examples, Figures 10.2 to 10.4, we see that as we apply the kernel to the input image, the size of the output image is reduced. In fact, by convolving an image of size $m \times n$ with a kernel of size $k_m \times k_n$ the resulting image will have a height of $m - k_m + 1$ and a length of $n - k_n + 1$. This type of convolution, where the resulting image is smaller, is often called "valid". If we want the output image to be the same size as the input image, we have to add more rows and columns to our image, this is known as padding. In this case, we use the formula $m + 2p_m - k_m + 1$ and $n + 2p_n - k_n + 1$ to represent the size of the new dimensions of the image. Here, p_m represents the number of rows to be added to the image, roughly half at the top and half at the bottom of the image. Similarly, p_n represents the number of columns to be added to the image, roughly half at the left and half at the right size of the image. For example, in the previous Figure 10.2, if the output has to be the same size as the input, we would have to use a padding of $p_m = p_n = 1$. So, padding would increase the dimensions of the input image from 5 by 5 to $6 + 2p_m - 3 + 1$ by $6 + 2p_n - 3 + 1$, i.e., 6 by 6. Typically, the value of the new pixels added to the image is zero.

10.1.1.2 Stride

The convolution examples we saw earlier used unitary steps, i.e., the kernel is applied by sliding through the pixels of the input image one by one. However, we can also use larger steps, as this reduces the computational cost of performing these steps at intervals. Figure 10.5 depicts an example with the steps of a convolution with stride=2 using a kernel of size 3 over an image of size 5 x 5 and no padding. This clearly has an impact on the resulting output, decreasing its resolution, but in cases where we do not need to extract delicate features this becomes a good option [1].

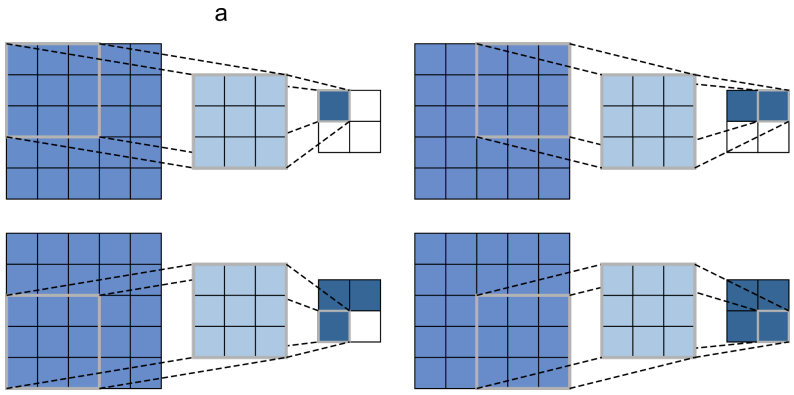


Fig. 10.5: Example using stride=2.

10.1.2 Pooling Layers

This is a very important layer, which aims to subsampling the image to reduce its size, and, consequently, reduce the total memory, processing and parameters needed, in addition to curbing the risk of overfitting [2, 5, 6].

As in convolution layers each output unit is connected to an input region, we must also take into account the size of the kernel, stride and padding. But, unlike the convolution, the "kernel", or, in other words, the region that will connect us to the input, will have no weights – it will perform only one operation, the most common being the maximum or the average [2].

Figure 10.6 depicts an example of max pooling, illustrating how it works (Figures 10.6(a-d)). This example uses a region of 2 x 2 , which is very common [5], and stride=1.

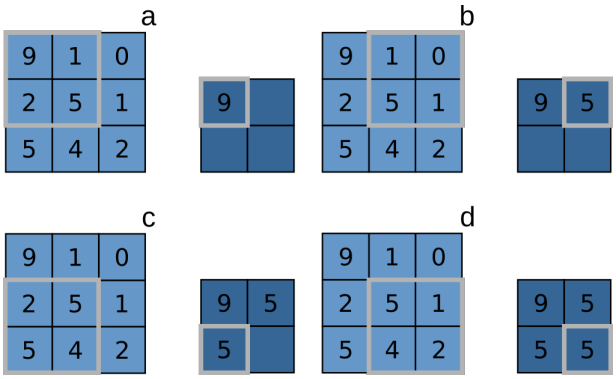


Fig. 10.6: Example of max pooling application.

Figure 10.7 depicts another example of max pooling, but this time performed with a 3-channel input. We can see that the operation is performed on each input channel of the object, and that its output contains the same number of channels as the input, which is what is typically done in this type of operation [2].

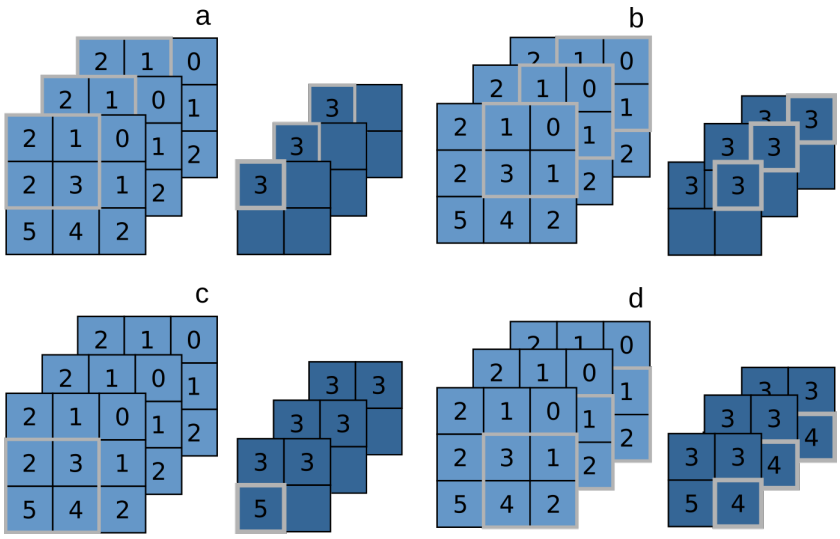


Fig. 10.7: Example of max pooling application in an image with more dimensions.

Although pooling is a very popular technique, we can find scientific works where their authors preferred not to use pooling to perform subsampling, but to use convolution layers with higher stride and appropriate padding values to achieve dimension reduction [6, 5]. This way of working was proposed by Springenberg [7], where they demonstrated that even networks without pooling layers can yield good results for some datasets.

10.1.3 Fully Connected Layers

CNNs usually have several convolution layers followed by activation functions, which in turn are followed by pooling layers, and this process decreases the dimensions $m \times n$ and increases the depth of the CNN model, that is, the number of feature maps generated by it [6, 2]. These feature maps represent the characteristics extracted from the input image, and we need to use this information as features to be given as input to fully connected layers, which are used to map these features to the desired outputs of the neural network.

The fully connected layers are basically an MLP [6], where each neuron of a given layer is fully connected to all neurons of the next layer.

Figure 10.8 illustrates fully connected layers. The input vector is composed of the feature maps produced by the layer that immediately precedes the fully connected layers. In this example, the feature map has a size of 7x7x64, that is, we have 64 feature maps of size 7x7. To provide these features as input to the fully connected layer, we have to flatten these feature maps to a vector of dimensions 1x3136, that passes through the network and ends in the Softmax layer, whose neurons have softmax activation functions (see chapter about MLP for activation functions), resulting in the output vector.

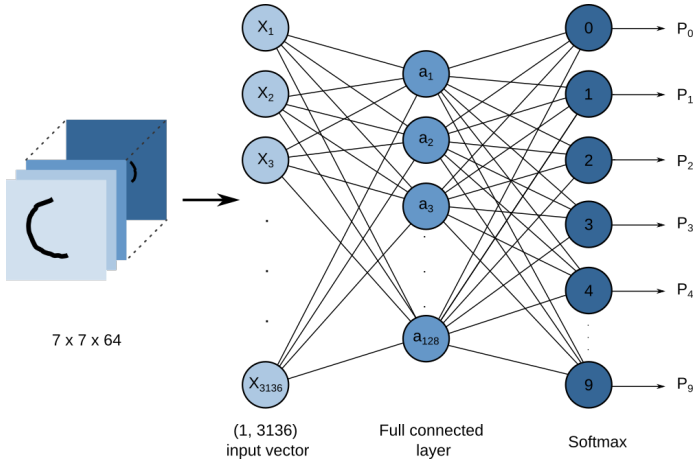


Fig. 10.8: Example of fully connected layers application in an image with more dimensions.

10.1.4 Why Use Convolutions?

So far we understand the building blocks of CNNs and the reasons they are used. The convolution operation is not only used because it is more efficient in image processing, but also because it is inspired by our own visual system.

Like many other neural network topologies, CNNs were bio-inspired by studies on the visual cortex of the human brain that began to take place since 1980 [2], mainly from the work of David H. Hubel and Torsten Wiesel, where experiments conducted on animals allowed them to deduce the functioning of the structure of the visual cortex.

In short, light signals received by the retina are transmitted to the brain through the optic nerve, where they reach the primary visual cortex, which is mainly formed by two types of cells [1]:

- **Simple cells**: these cells have behaviors that can be represented by linear functions in an image with small area known as the receptive field [1, 2]. This type of cell inspired the simplest detector units on CNNs.
- **Complex cells**: they also respond to features of the image, such as simple cells, but are invariant in position, that is, they do not make much distinction from where the feature appears. This type of cell inspired the pooling units [1].

Anatomically, the deeper we go into the layers of the brain, the more layers analogous to convolution and pooling are used, and we find more specialized cells that respond to specific patterns unaffected by input transformations. Until reaching these deeper layers, a sequence of detections followed by pooling layers is performed [1].

10.1.5 Classic CNNs

10.1.5.1 LeNet

After studying the main building blocks of a CNN – convolutional layers, pooling layers and fully connected layers – it becomes easier to compare CNNs architectures and we can realize that even with the differences they present a pattern in the combination of layers. Typically, CNN architectures have an interleaved sequence of convolution layer, followed by a pooling layer, which repeats up to the edge of the network where there are some fully connected layers with similar structure to MLP networks. An example of this basic structure was shown in Figure 10.1.

Figure 10.9 illustrates a LeNet CNN. As the operations are carried out along this network, it is noticed that the feature maps are getting smaller and that the layers deeper, that is, the number of maps in the same layer increases. As the output layer usually presents itself as a vector of probabilities for each class, there is a transition from the representation of the data in maps to a vector, starting from the flattening process, which occurs before the first fully connected layer.

The LeNet network was one of the first CNNs that showed potential application in computer vision. The network was created by Yann LeCun in 1998 for the purpose of handwriting number recognition. LeNet was later adapted to recognize digits for ATM machine deposits, and there are still ATMs that run the code developed by Yann and his colleague Leon Bottou [8]. The network was also widely used for digit recognition of the MNIST dataset [2].

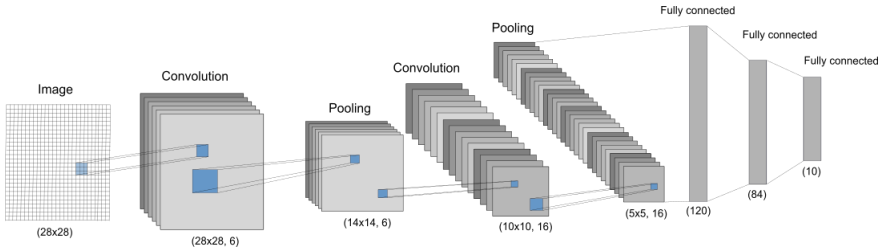


Fig. 10.9: LeNet Convolutional Network - The input is an image of a handwritten number and the output a vector with the probability for each of the ten digits from 0 to 9 [8].

In Figure 10.9, we take as input a standard MNIST grayscale image, of size 28 x 28 pixels. In the general scheme of the LeNet network (Figure 10.10), we have a clearer view of the combination of layers, where after the input layer there are two convolutional layers, each followed by a pooling layer, and at the end there are three fully connected layers.

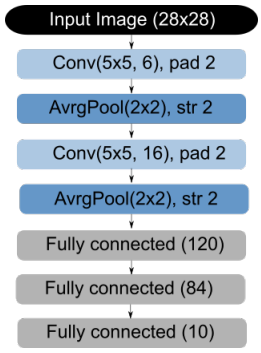


Fig. 10.10: General diagram of layers in the LeNet network - Schematic of the LeNet network with the sequence of convolutional layers ("Conv"), pooling ("AvrgPool") and fully connected layers [8].

Each convolution layer uses a filter 5×5 and a sigmoid activation function (see the chapter about MLPs for a definition of activation functions). The first convolution layer has 6 channels or maps, while the second one has 16. The pooling operation involves a filter 2×2 that calculates the average, so it is identified as "AvrgPool", and uses stride $str = 2$, so that each map from the previous layer is reduced in a half along the width and height, eliminating 75% of the activations. The size of the three fully connected layers are respectively 120, 84, and 10. The last layer, "Fully connected(10)",

corresponds to the possible number of classes, in this case 10 (digits from 0 to 10). The activation function in the last layer is a Gaussian function.

To understand the effects of each layer on the dataset, we present in Table 10.1 the dimensions of the outputs of each layer. Table 10.1 shows that from one convolution block to the other there is an increase in the number of channels (C) from 6 to 16, between the pooling layers these values are not changed, as the process only reduces the width (W) and height (H) of the channels. In fully connected layers, the dimensions are reduced until the size of the number of classes is obtained.

Layer	**Channel (C)**	**Size (H,W)**	**Filter (K)**	**Memory (kB)**	**Parameters**
Inputs	1	28			
Convolutional 1	6	28	5	18	156
Avrg Pooling 1	6	14	2	5	0
Convolutional 2	16	10	5	6	2416
Avrg Pooling 2	16	5	2	2	0
Flatten	400			1.6	0
Fully Connected 1	120			0.5	48120
Fully Connected 2	84			0.3	10164
Fully Connected 3	10			0.04	859
Total				33	61706

Table 10.1: LeNet Layer Settings, Parameters and Information - Summary of LeNet's main layer settings such as number of channels and filter size. Display an estimate of the number of parameters and the amount of memory to train the network.

It is common in CNN's networks that the number of channels practically doubles after a pooling layer since there is a reduction by half in the dimensions of the maps. Thus, it is possible to increase the number of maps, making it more sensitive to identify low-level features such as borders and textures, without drastically increasing the number of parameters and computational resources [8]. As the first convolution layer applies padding = 2, the maps maintain the same dimension in the output as the original image (28×28), however the second layer does not have padding, which reduces the width and height of the maps by 4 pixels.

Over the years, variations of this model have emerged and the most evident difference between the networks is the number of layers, which has increased over the years, making the networks deeper. When increasing the number of layers it was noticed that the performance of the networks tended to improve, however some limitations emerged. The greater the amount of data, the more computational memory is required, and this capacity depends on hardware requirements.

To assess the amount of memory used in training the LeNet network, we will use an approximate calculation based on the amount of output elements in each layer. The number of elements is multiplied by the number of bytes needed to store each element [9]. Considering that floating point data occupy 32 bits, therefore 4 bytes per element, to facilitate the visualization of the results, the measure kilobyte (kB) is used, and for this reason they were divided by the factor 1024 since 1 kB = 1024 B . In Equation 10.3, we exemplify the calculation of the amount of memory for the first convolution layer of the LeNet network:

$$\begin{aligned} \text{Amount of memory} &= \text{CxHxW} \\ &= 6 \times 28 \times 28 \\ &= 4704 \text{ output elements} \\ &= 4704 \times 4 \text{ bytes} = 18816 \text{ bytes} \\ &= 18.38 \text{ kilobytes} \end{aligned} \tag{10.3}$$

The C parameter identifies the number of channels or maps of the layer and the term H and W, the height and width of the layers output element, respectively. As shown in Equation 10.3 and Table 10.1, the approximate amount of memory for the first layer is 18 kB, and in total for the network 33 kB. The first layers tend to need more memory due to the larger dimensions (W and H) of the channels [9].

Increasing the number of layers also requires that more parameters be learned, which affects both the training time and its performance, because if there is not an adequate optimization of the parameters, the probability of overfitting can be higher [6]. To approximately determine the number of parameters related to each layer, the weights related to the filters of each map were considered, calculated as the product between the filter dimensions (K x K), the number of input element channels and the number of output channels of the layer [9]. The biases associated to each output channel were also considered as parameters. On fully connected layers, the number of parameters is determined as the product of the number of input elements and the number of output elements of the layer plus the number of biases. Equation 10.4 exemplifies the calculations for the first convolution layer:

$$\begin{aligned} \text{Weights} &= \text{C}_{\text{output}} \times \text{C}_{\text{input}} \times \text{K} \times \text{K} \\ &= 6 \times 1 \times 5 \times 5 \\ &= 150 \end{aligned}$$

$$\text{Bias} = 6 \tag{10.4}$$

$$\begin{aligned} \text{Parameters} &= \text{Weights} + \text{Bias} \\ &= 150 + 6 \\ &= 156 \end{aligned}$$

Considering that the filter in the first layer is of size 5×5, that the input has only 1 channel and that there are 6 channels in the convolution layer, the first convolution layer considers approximately 156 parameters. In Table 10.1, there is also the number of parameters related to each layer and the approximate total of parameters for the LeNet network is 61706. As the number of channels in the convolution layer increases, more parameters are needed. In general, most parameters are due to fully connected layers due to the greater number of connections [9].

10.1.5.2 AlexNet

Currently there are several CNNs network architectures used for applications in computer vision. The evolution of these networks can be seen in the results of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition. The main objective of the competition was to evaluate algorithms for object detection and image classification. The first edition of the competition, in 2010, involved 1.2 million images for training, being 1000 categories of objects. In the first two years of competition, CNNs networks weren't yet in 1st place, however, from on 2012 CNNs models started to lead the competition [10]. The progress of the networks can be evaluated based on the error rate of the models, which in seven years dropped from approximately 26%, in the second year of the competition, to 2.3% in the last edition of the competition in 2017 [9], as shown in the graph in Figure 10.11.

To learn a little about the different architectures of CNNs networks and notice some differences, and structures that performed well and are still adopted by recent architectures, we will highlight below three additional architectures that become well known and had prominence in the competition. The AlexNet network was the first CNN to win the ImageNet competition in 2012 with an error rate of 16.4%. The VGG network did not lead the competition in 2014, but it is one of the models with great popularity. In 2014, the CNN GoogLeNet won the competition and served as inspiration for the Inceptions networks.

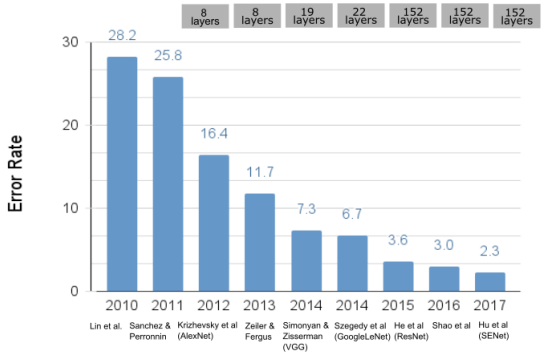


Fig. 10.11: Error rate of the best performing models in the ImageNet competition - The performance of the models in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition was mainly evaluated by the error rate. The graph shows the models that won in each edition of the competition, which ran from 2010 to 2017, and also networks that became popular such as VGG [9].

The AlexNet network was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton [2]. This network is very similar to LeNet, but has more layers. Because it is a deeper network, requiring more memory, the original network had to be physically distributed between two 3GB GPUs [11]. In this way, the network was drawn as in Figure 10.12, with a dual data stream structure so that each GPU would receive half of the model.

As depicted in Figure 10.13, AlexNet has 5 convolution layers, with the first three followed by pooling layers. The most aparent difference between the AlexNet and LeNet architectures are the three additional convolution layers in the AlexNet network, which are followed one after the other with no layer pooling between them. As the input images are bigger than the MNIST dataset approached in the example LeNet network, the input convolution filters are bigger (11×11) and stride str $= 4$ is used. In the second convolution layer, the filters have size 5×5, and in the other convolution layers filters 3×3 are used.

With the discovery that ReLU's activation functions in the convolution layers and that maxpooling improve the performance of networks, most models were built using these functions [8]. Maxpooling filters of size 3×3 and stride str $= 2$ scale down channels based on the largest value of the receive field. With the exception of the first layer, all other convolution layers are padded so that the dimension of the channels is not changed after the convolutions.

The last three layers are fully connected and have sizes 4096, 4096 and 1000, respectively. The output layer has dimension 1000 due to the number of possible classes of ImageNet competition and the activation function is Softmax.

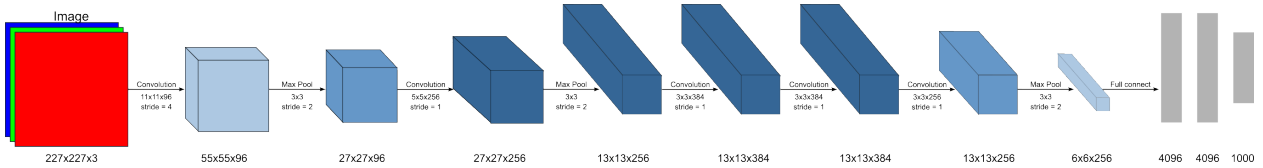


Fig. 10.12: AlexNet network architecture - represented as the combination of two identical networks, as originally the training would occur with the distribution of data between two GPU's [11].

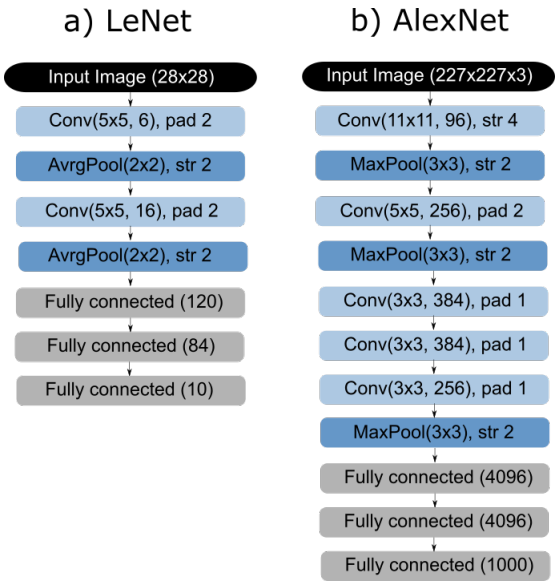


Fig. 10.13: Comparison of AlexNet and LeNet networks. (a) LeNet Network and (b) AlexNet Network. These general layer schemes show that the main difference of networks is that AlexNet is deeper, with three more convolution layers than LeNet [8].

The same pattern for the dimensions of the layer output elements seen in LeNet is seen in Table 10.2 for the AlexNet network. While the size of the channels decreases from one convolution layer to another, the number of channels increases, with 96 in the first, followed by 256, 384, 384 and 256. After the flatten process, the dimension of the layers is reduced until it is established the size of the prediction classes vector.

By comparing the amount of memory and the approximate number of parameters as described in the subsection 10.1.5.1 it is observed that the amount of memory required increases and the number of parameters also increases. Approximate calculations indicate that while the memory required for training the LeNet network would be approximately 33 kB, for AlexNet it would be approximately 3 GB. The number of parameters calculated for LeNet was 62 thousand and for AlexNet 62 million. Generally in both models, the first layers require more memory, while the fully connected layers need more parameters.

Layer	Channel (C)	Size (H,W)	Filter (K)	Memory (kB)	Parameters
Inputs	3	227			
Convolutional 1	96	55	11	1134	35
Max Pooling 1	96	27	3	273	0
Convolutional 2	256	27	5	729	615
Max Pooling 2	256	13	3	169	0
Convolutional 3	384	13	3	254	885
Convolutional 4	384	13	3	254	1327
Convolutional 5	256	13	3	169	885
Max Pooling 3	256	6	3	36	0
Flatten	9216			36	0
Fully Connected 1	4096			16	37753
Fully Connected 2	4096			16	16781
Fully Connected 3	1000			4	4097
Total				3090	62378

Table 10.2: AlexNet Layer Settings, Parameters, and Information - Summary of settings for the main AlexNet layers, such as number of channels and size of filters. Display an estimate of the number of parameters and the amount of memory to train the network.

10.1.5.3 VGG

The VGG network was conceived by the members of the Visual Geometry Group (VGG) at Oxford University by researchers Karen Simonyan and Andrew Zisserman [8]. Compared to the two previous architectures LeNet and AlexNet, VGG adopts principles to establish the structure of the network, which allowed the construction of deeper models [8]. Another characteristic of VGG is the block structure in the part of the network with the convolutional layers, in which each block presents convolutional layers in sequence and at the end a pooling layer. While the AlexNet model, in Figure 10.14, presents 5 convolutional layers, the VGG presents 5 blocks with a variable number of convolution layers, but in general the first blocks have fewer layers. Like AlexNet, at the edge of the network there are three fully connected layers, with equal dimensions on both models, and a Softmax activation function at the output.

In Figure 10.14, there is a representation of the VGG architecture with 16 layers, in which the first two blocks have two convolutional layers and the last three blocks have three convolutional layers. The convolution layers double in size with each block, with each layer in the first block having 64 channels, 128 channels in the next, and so on up to 512 in the last block. The use of the ReLu activation function in the convolution layer and maximum value pooling are strategies that performed well in AlexNet and continued in other models, such as in VGG.

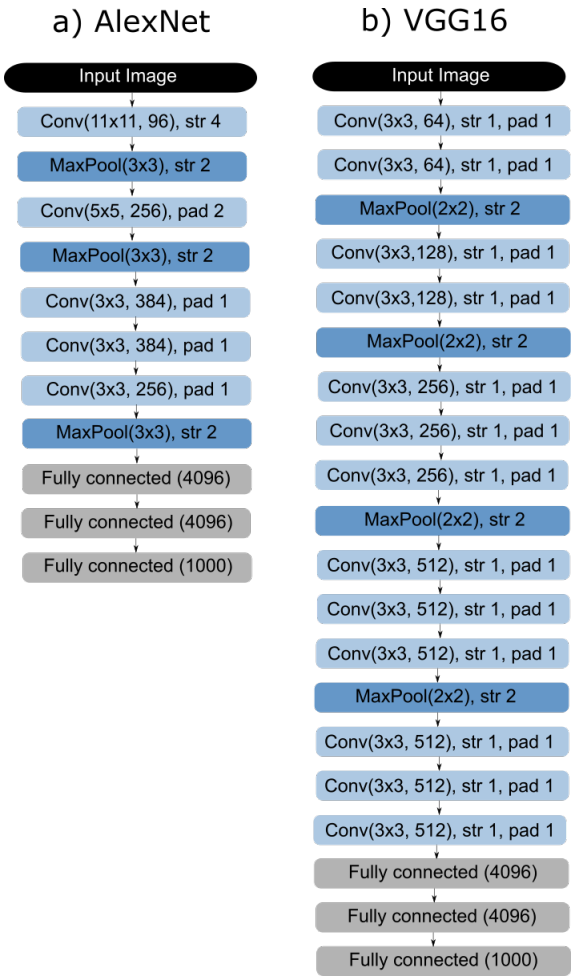


Fig. 10.14: Comparison of VGG and AlexNet networks - Comparison of VGG and AlexNet networks based on the general structure of the layers. While the final part of the networks is similar in relation to the fully connected layers, the VGG differs in that it is deeper and presents a pattern of convolutional layers organized in blocks [9].

In LeNet and AlexNet networks, it is usually necessary to individually select several hyperparameters[1]. For example, in the convolution layers, the number of channels, size of filters, padding and stride are adjustable. In the pooling layer, the hyperparameters are the filter and stride size. In general,

[1] Hyperparameters are parameters that need to be pre-defined, as opposed to learned by the model.

these two networks do not provide a general guide on how to select the parameters. VGG's core design principles state that all convolution filters are 3×3 with stride str $= 1$ and padding pad $= 1$, and that maxpooling filters are 2×2 with str $= 2$. After each pooling layer, the number of channels doubles in the convolution layer. The idea of fixing the size of convolutional filters came from the perception that the combination of two filters 3×3 presents a receptive field equivalent to one filter 5×5, and that three filters 3×3 perform similar to one of 7×7 [6]. Fixing the size of filters and their stride str $= 1$ and pad $= 1$ establishes that the dimension of the channels does not change between the convolutional layers, so the only hyperparameter that needs to be optimized is the number of layers in each block.

By using filters 3×3, which are smaller, but in greater quantity than those used in AlexNet (11×11 and 5×5), more nonlinearity is included, allowing the network to learn more low-level features [6]. Increasing the depth of the network with more layers of convolution adds more non-linear activation functions. Even being deeper networks, this strategy of using smaller filters reduces the number of parameters. Considering that two layers in sequence have C channels each, when using two filters 3×3, the total number of parameters is $2 \times 3 \times 3 \times C^2 = 18C^2$, which is smaller number when compared to the scenario of a single filter 5×5 with $25C^2$ parameters [9].

Of course, doubling the number of channels between blocks should make the number of parameters grow quickly, and that's why maxpooling filters have been standardized to reduce the dimensions of the channels by half. By controlling the number of activations that pass to the next layers, it is possible to keep the number of operations approximately constant. Superficially evaluating that the number of operations is given as the total amount of multiplications and additions, we can calculate for each layer as the product of four parameters in Equation 10.5 [9]: filter size(K x K), input channel dimensions (H x W), quantity of input channels (C_{input}) and output channels (C_{output}).

$$\begin{aligned}\text{Number of operations} &= \text{Number of output elements x Operations by output element} \\ &= (C_{output}\text{ x H x W}) \times (C_{input}\text{ x K x K}) \\ &= (2C\text{ x HW}) \times (2C\text{ x 3 x 3}) \\ &= 36HWC^2\end{aligned} \tag{10.5}$$

In the case of two convolution layers with filters 3×3 and separated by a pooling, reducing by half the size of the channels (2H x 2W $\rightarrow$ H x W) and doubling the number of channels (C $\rightarrow$ 2C), the number of weights increases from $9C^2$ to $36C^2$, but the number of operations remains at $36HWC^2$.

10.1.5.4 GoogLenet and Inception

By following the evolution of CNNs, we can see that the main strategy to increase the performance in the classification of images was to increase the number of layers that keep the weights of the networks. AlexNet and VGG-16 networks were developed with 8 and 16 layers, respectively. As the networks get deeper, the dilemma of how to make the algorithms more efficient arose, since more layers meant more parameters and operations, requiring more computational resources. Comparing the networks in Figure 10.15, it is possible to verify the accuracy of the networks, the number of parameters and the number of operations. It appears that for the VGG-16 network to achieve better results than the AlexNet network, it was necessary to more than twice the number of parameters, from approximately 65 million on AlexNet to just over 130 million on VGG-16.

In the 2014 ImageNet competition, a Google research group led by Christian Szegedy proposed the GoogLeNet architecture that should both ensure good performance and be more efficient than existing models [2]. The model not only won the competition but also met their requirements, as even being a network with 22 layers, more than the VGG-16, it used 12 times less parameters than VGG, 13 million instead of 138 million [6].

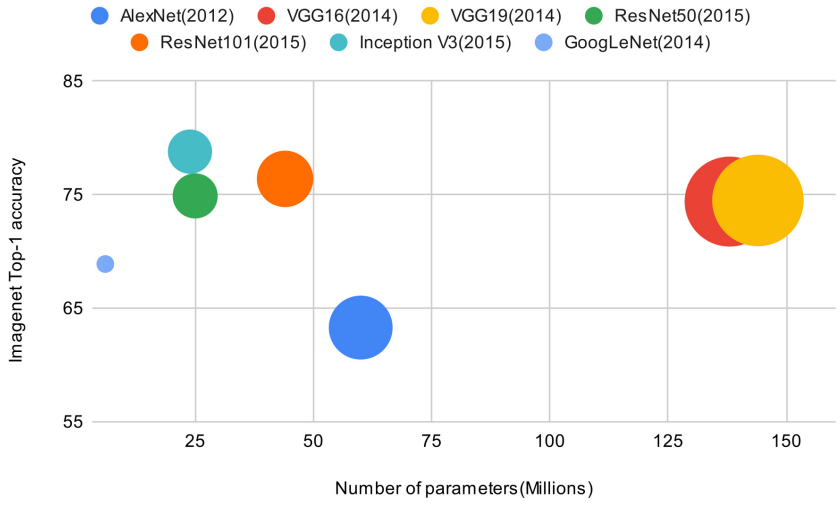


Fig. 10.15: CNN's neural networks evolution graph - Network performance is evaluated by accuracy versus the number of operations required for a single forward step. The radius of the circles is proportional to the number of parameters, with the legend in the lower right corner indicating a reference from 5×10^6 to 155×10^6 [12].

To understand the GoogLenet network we can divide it into three parts (Figure 10.16), in the first part, the input layers are similar to the AlexNet and VGG networks, in the second part, there are the inception blocks characteristic of this network, and the last part refers to the classification structure. The first part contains two blocks with a sequence of convolutional layers followed by pooling 3×3. In the first block, there is only a convolution layer 7×7, with stride str $= 2$ and padding pad $= 3$, and a pooling layer with str $= 2$. At the end of these two layers, the element has 64 channels and was reduced by 4 in its dimension (H and W). In the second block, there are two convolution layers, the first with a filter 1×1 and 64 channels and the second 3×3 with 192 channels, and only the pooling 3×3 at the end of the block changes the dimensions of the channels in half.

The main role of these two blocks is to reduce considerably the dimensions of the image, since most of the memory required is due to the first layers [9]. Whereas, at this stage, there is an 8-fold reduction in image dimensions, an entry 224×224 when reducing to approximately 28×28 use approximately 7.5 MB memory while the same reduction in VGG-16 needs 42.9 MB, almost 6 times more than GoogLenet [9]. Also, when passing a smaller image to the next layers, the number of operations and the number of parameters to train the network are also reduced.

Another technique to make the network more efficient was to include a global AvrgPool layer before the classification layer [2]. In previous CNN models it was common to include flattening to convert the data into a vector, losing the spatial information, to be compatible with the fully connected layers that made the classification. These last layers end up being responsible for most of the parameters. In the VGG-16 model, for example, the 3 fully connected layers generate approximately 123.6 millions of parameters, almost 90% of the total parameters [9].

Instead of adopting flattening, GoogLenet uses an averaging filter of the same dimension as the input element, returning the average of the maps for each position of the vector. As the output vector is already reduced in size, it is necessary to include only one layer fully connected with 1000 classes. Since the global averaging layer does not need parameters, and since it returns a vector 1024, approximately 1 million parameters are needed in the fully connected layer, 100 times less than in the VGG [9]. In the last layer, as in the VGG, a Softmax activation is associated, while in the convolution layers it is a ReLu.

The first and last parts of the GoogLenet network have been explained above. The intermediate section that we will study now includes the Inceptions modules that have become characteristic elements of the most modern networks. Each module is similar to VGG blocks, in that some convolutional layers are present in sequence and at the end a pooling layer. In the case of the VGG, it was seen that to reduce the number of hyperparameters, the size of the filters was set to 3×3, and the variable parameter was the number of convolutional layers. The idea of Inceptions (Figure 10.17) is not to worry

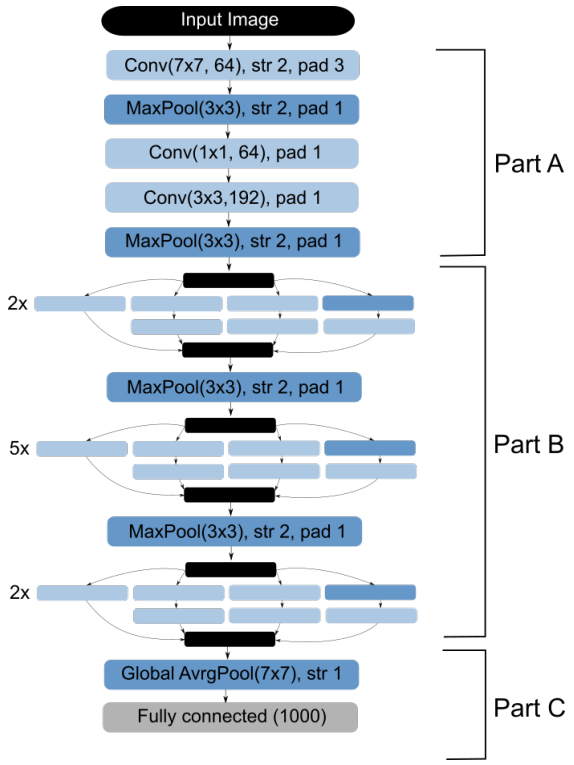


Fig. 10.16: The general structure of the GoogLenet network can be divided into three parts: Part A - Similar to AlexNet and LeNet, contains a sequence of convolutional layers and pooling to reduce image dimensions; Part B - Inceptions modules separated by pooling layers; Part C - Global pooling layer and a Fully Connected for classification [6].

about the size of the filters or the number of layers in the module, as each module consists of a combination of filters with different sizes arranged in a fixed way [6].

From the input of the inception module, copies of the input element go along four paths at the same time. At the end of these paths, the image size does not change, but the number of channels is changed in different ways, with the choice of the number of channels for each layer being a hyperparameter. At the output of the module there is a concatenation of all these channels, forming a single element with the same dimension as at the input and with a number of channels that is the sum of all that resulted from each path.

The first path has only one convolution 1×1, known as the bottleneck, whose main function is to preserve the dimensions (height and width) but reduce the number of channels, which reduces the computational cost and the number of parameters [6]. As this convolution includes more nonlinearity at a

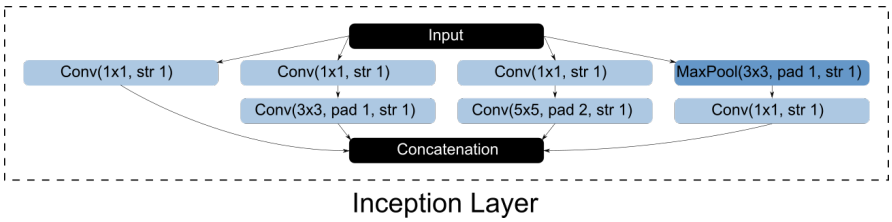


Fig. 10.17: Inception module of the GoogLenet network - The middle part of the GoogLenet network is formed by a sequence of Inceptions modules separated by pooling layers (Figure 10.16). Each module has four paths to the same input data, and on output, where the results are concatenated [8].

low cost, a convolution 1×1 is also included in the input layers, contributing to an optimization in the first part of the network [2]. This same layer has been added at the beginning of each of paths 2 and 3 to reduce model complexity. After reducing the number of layers, larger filters are included, making it possible to process information at different scales, with the filters being in the second way 3×3 and in the fourth way 5×5 [8].

All paths, even the fourth that includes a MaxPool layer, feature padding to keep the same dimension of the channels as in the entrance. As MaxPool does not change the number of channels, a convolution 1×1 is included at the end of the fourth path, reducing the volume.

By concatenating all the channels of each path at the end, the inception module follows the hypothesis that visual information can be processed at various scales and that the aggregated results allow the next level to extract several features from different scales at the same time [6]. In the GoogLenet network in Figure 10.16, we see three groups of inception modules interspersed by Maxpooling 3×3, totalling 9 modules.

The previous GoogLenet diagram (Figure 10.16) is one of the more simplified representations of the model, because as seen in Figure 10.18, the original architecture includes two classifiers that run in parallel with the other blocks described above, one that starts after the third inception module and the other after the sixth module [2]. Each classifier works similarly to the final part of the network, where classification takes place [9]. The classifiers are formed by an AvrgPooling layer, followed by a convolution layer, two fully connected layers and at the output a Softmax activation function.

There is a peculiarity when training deeper networks, because, in the backpropagation of errors, the rates reduce to values very close to zero, making it difficult for the algorithm to converge. One of the techniques adopted by GoogLenet to help convergence was to include the calculation of the error gradient of these intermediate classifications in the error backpropagation [2].

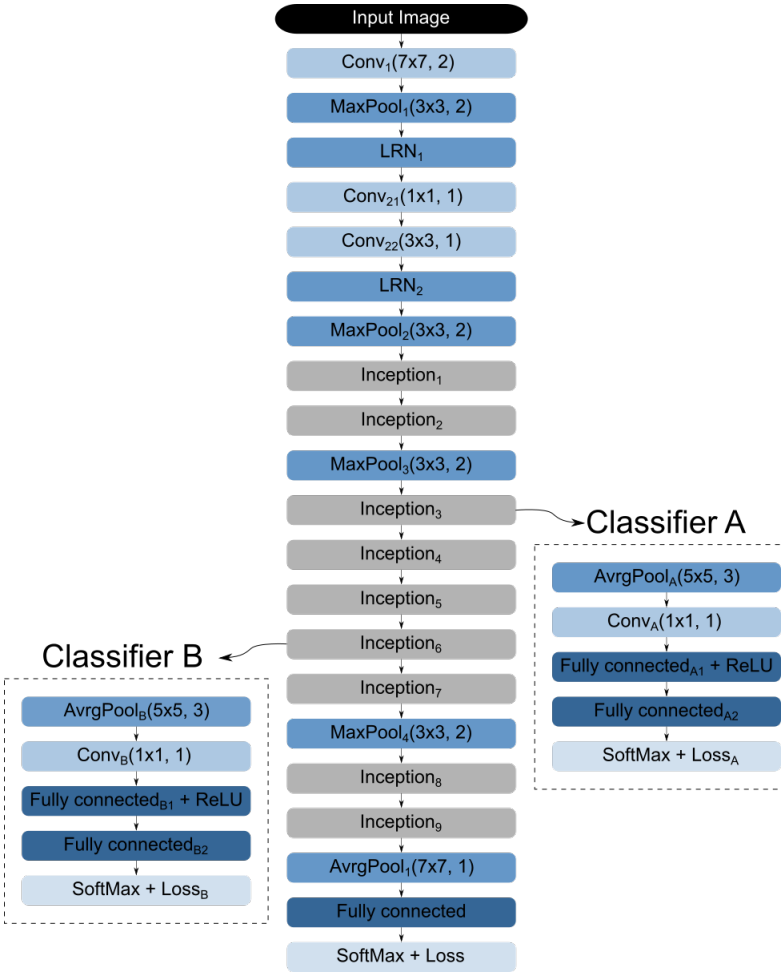


Fig. 10.18: GoogLenet network architecture with intermediate classifiers - The GoogLenet network with two classifiers, one in the third inception module and the other in the sixth module. These intermediate classifiers reduce the fading effect of error gradients [13].

10.2 Transfer learning

Training a neural network from scratch is a complex task, as it requires an enormous amount of data and computational power [6]. In many cases we may not have enough of either, but that doesn't stop us from creating our models. For this we have a technique known as transference learning, which basically consists of benefiting from already trained models to solve

new problems. In this case we assume that there is a relationship between the old and new problems, such that knowledge learnt from the old problem may help learning the new problem. We are able to use this technique for computer vision problems because even models trained for different computer vision problems end up learning to identify similar features of the image in the earlier layers, and then learning greater details in the deeper layers.

Transfer learning in the context of deep learning typically reuses some components of existing neural networks that have already been trained on previous problems using very large datasets, inserting such components into a new neural network that is being built for a new problem. A detailed explanation of how to perform transfer learning is out of the scope of this book chapter, but we refer the interested reader to [6] for further information on that.

10.3 Summary and Discussion

This chapter has introduced one of the most popular types of deep learning approaches – the CNNs. Such approaches have achieved excellent results in the field of computer vision. Many different CNNs architectures exist, and this chapter has introduced four of them. When studying the different models of CNN we realize that the main factor to improve the accuracy of the networks has been the increase in the number of layers, making the networks deeper. It is noteworthy that the construction of deeper models was only possible due to the availability of large datasets and the evolution of hardware performance, especially memory and processing units, and the development of more specific software, known as frameworks for Deep Learning [8].

10.4 Exercises

1. Considering the focus on convolutional neural networks, we can highlight convolution as a basic operator. For a basic understanding of the application and effect of convolution, manually convolute the kernel $\begin{bmatrix} 0 & 2 & 3 \\ 0 & 1 & 0 \\ 3 & 0 & 2 \end{bmatrix}$ onto the image matrix $\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 4 & 1 & 0 \end{bmatrix}$. Use padding with $p_m = p_n = 1$ and stride=1. Use the value of zero for the pixels added through padding.
2. In the context of applications with images, each convolution layer can identify a level of detail in the image. For example, in the initial layers typically identify more general elements of the images, such as lines, edges and corners. Some pre-determined filters, or convolutions, that have the edge detection property are Sobel and Prewitt. Each of these operators is composed of to kernels as shown below:

   - Prewitt Operators.
     $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$
     $G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
   - Sobel Operators.
     $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
     $G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

   By applying each of the two Prewitt kernels on the image and summing the modulus of the two resulting matrices to get a single output matrix, it is possible to determine the edges in the image. A similar procedure can be used with the Sobel operators, which also result in detection of edges. Apply the Prewitt and Sobel operators in the image represented by the matrix below, using no padding and stride=1:
   $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$
3. One of the techniques used in image applications using convolutional neural networks is to reduce the size of inputs across layers, which can reduce computational costs. To perform this reduction we can use strides in the application of convolution. Perform the convolution between kernel $\begin{bmatrix} 0 & 2 & 3 \\ 0 & 1 & 0 \\ 3 & 0 & 2 \end{bmatrix}$ and matrix $\begin{bmatrix} 1 & 2 & 0 & 3 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 1 & 0 & 2 & 1 \end{bmatrix}$, first using stride 1 and then with stride 2 to assess the effect of this technique. Use no padding.
4. During image processing in convolutional neural networks, there are steps in which the inputs and outputs of the convolution layers need to keep the same size, and one way to have this control is to use padding in the application of the convolution. With the same kernel and matrix as in the

previous exercise, determine a padding to ensure that the result remains the same size as the original matrix, and then apply the convolution using this padding.

10.5 Exercise Answers

1. Padding will result in the following new input matrix: $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$.

   The output matrix will then be: $\begin{bmatrix} 1 & 11 & 0 \\ 13 & 16 & 3 \\ 10 & 1 & 0 \end{bmatrix}$.
2. Prewitt result $\begin{bmatrix} 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 4 & 3 & 3 & 3 & 3 \\ 0 & 0 & 2 & 3 & 3 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ Sobel result $\begin{bmatrix} 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 & 4 & 4 & 4 \\ 0 & 0 & 2 & 4 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
3. Using stride = 1: $\begin{bmatrix} 16 & 16 & 9 \end{bmatrix}$.
   Using stride = 2: $\begin{bmatrix} 16 & 9 \end{bmatrix}$.

References

1. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
2. Aurelien Geron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.
3. F. Patel. Textile fabric defect classification using convolutional neural networks. M.s. thesis, Dept. of Computer Science, University of Leicester, Leicester, UK, 2018.
4. J. Brownlee. How do convolutional layers work in deep learning neural networks? `https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/`, 2019. Accessed 5th January 2023.
5. R Dr Adrian. Deep learning for computer vision with python, 2017.
6. Mohamed Elgendy. *Deep Learning for Vision Systems*. Manning Publications, 2020.
7. Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.
8. Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020.
9. Justin Johnson. Deep learning for computer vision - fall 2019, 2019. Last accessed 20 May 2021.
10. Princeton University Stanford Vision Lab, Stanford University. Imagenet large scale visual recognition challenge (ilsvrc), 2020. Last accessed 20 May 2021.
11. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

12. Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.
13. S. P. K. Karri, Debjani Chakraborty, and Jyotirmoy Chatterjee. Transfer learning based classification of optical coherence tomography images with diabetic macular edema and dry age-related macular degeneration. *Biomed. Opt. Express*, 8(2):579–592, Feb 2017.
14. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
15. Aldo von Wangenheim. Deep learning: Introdução ao novo coneccionismo, 2018. Last accessed 14 April 2021.

Chapter 11
Naive Bayes

Leandro L. Minku

The process of learning predictive models from data typically involves uncertainty. For instance, different models could potentially be created to describe the same data, and different predictions could be given to future data. The decision of which model to adopt and which prediction to make thus involves uncertainty [1]. This chapter will introduce you to probabilistic learning, which captures uncertainty through probability theory. In particular, we will cover a machine learning approach called Naïve Bayes. Despite being a relatively simple technique, Naïve Bayes often outperforms more complex approaches [2], being a good place to start learning the topic of probabilistic learning.

This chapter assumes that you are relatively familiar with probability concepts such as random variables; probability distributions; probability mass functions; probability density functions; prior, marginal or unconditional probabilities; conditional or posterior probabilities; and joint probability distributions. If you are unfamiliar with these terms, we suggest you to read Section 13.2.1 and 13.2.2 of [3] or Chapter 6 of [4] for a deeper understanding of this chapter.

Section 11.1 introduces the Bayes theorem upon which Naïve Bayes is based and the relationship that this theorem has with classification. Section 11.2 introduces the Naïve Bayes approach for categorical input variables. Section 11.3 introduces Naïve Bayes for numeric input variables.

School of Computer Science, University of Birmingham, UK

11.1 The Bayes Theorem and Its Relationship With Classification

Consider a machine learning problem with input variables $\mathbf{x}$ and output variable y, where $\mathbf{x} = (x_1, x_2, \cdots, x_d)$ is a d-dimensional vector, $\mathbf{x} \in \mathcal{X}$, $y \in \mathcal{Y}$, $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the output space of the problem. In regression problems, $\mathcal{Y} = \mathbb{R}$, whereas in classification problems $\mathcal{Y}$ is a set of categories. In this chapter, we will focus on classification problems.

In probabilistic learning, we consider that $(\mathbf{x}, y)$ are random variables drawn from a fixed, albeit unknown joint probability distribution $p(\mathbf{x}, y)$ [1]. In particular, in supervised learning, we assume that a training set of examples is available, consisting of N samples[2] drawn from $p(\mathbf{x}, y)$:

$$\mathcal{T} = \{(\mathbf{a}^{(1)}, c^{(1)}), (\mathbf{a}^{(2)}, c^{(2)}), \cdots, (\mathbf{a}^{(N)}, c^{(N)})\}, \tag{11.1}$$

where $\mathbf{a}^{(i)}, c^{(i)}$ represent the values of the input and output variables[3]. A given assignment of values $\mathbf{a}$ to the input variables $\mathbf{x}$ is also referred to as an *event* [3] in probabilistic learning. Supervised learning is then considered to be the problem of using the training set $\mathcal{T}$ to learn a model $f : \mathcal{X} \rightarrow \mathcal{Y}$ able to generalise to unseen examples from $p(\mathbf{x}, y)$.

Based on the product rule from probability theory, the joint probability distribution $p(\mathbf{x}, y)$ can be written as:

$$p(\mathbf{x}, y) = p(y|\mathbf{x})p(\mathbf{x}) = p(\mathbf{x}|y)p(y),$$

where $p(y|\mathbf{x})$ is the conditional distribution of y, $p(\mathbf{x})$ is the unconditional or marginal distribution of $\mathbf{x}$, $p(\mathbf{x}|y)$ is the conditional distribution of $\mathbf{x}$ and $p(y)$ is the prior probability of y. From this, we have that

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

Therefore, given an example with known value of $\mathbf{x} = \mathbf{a}$, one can calculate the probability of it belonging to a given class $y = c$ as

$$p(y = c|\mathbf{x} = \mathbf{a}) = \frac{p(y = c)p(\mathbf{x} = \mathbf{a}|y = c)}{p(\mathbf{x} = \mathbf{a})}. \tag{11.2}$$

[1] There are also machine learning problems where $p(\mathbf{x}, y)$ is not fixed. However, we will not cover them in this book. For more information on those problems, we recommend [5].

[2] In some machine learning problems, the training set may not be fully available beforehand. Instead, training examples may arrive over time.

[3] In this chapter, we will use letters such as $\mathbf{a}^{(i)}, c^{(i)}$ to represent the values of the input and output variables instead of using their corresponding indexed variable names $\mathbf{x}^{(i)}, y^{(i)}$ in sample i. This will enable us to distinguish more explicitly between the variables and their values, which can be useful when discussing Naïve Bayes.

This is known as the Bayes Theorem. To simplify the writing, whenever it is not ambiguous, we will write this as

$$p(c|\mathbf{a}) = \frac{p(c)(\mathbf{a}|c)}{p(\mathbf{a})}.$$

Such probabilities can be used to make inferences, i.e., to predict the class c given the observed input values $\mathbf{a}$. In particular, one can compute $p(c|\mathbf{a})$ for every possible class c, and predict the class associated to the highest probability $p(c|\mathbf{a})$. This means that these probabilities can work as our model $f : \mathcal{X} \rightarrow \mathcal{Y}$. But how to compute these probabilities? For that, in supervised learning, we can rely on the fact that we have access to a training set $\mathcal{T}$. This training set can be used to create frequency tables that enable us to compute these probabilities.

Let's have a look at a simplified problem to illustrate how this works. Consider a problem where we have a single input variable x_1 representing whether a person eats healthy foods and a output variable y corresponding to whether that person is developing cancer. The problem is to predict whether the person is developing cancer based on whether they eat healthy foods. In this problem, we will refer to x_1 as "healthyFood" and y as "cancer", to facilitate reading.

Assume we have access to the training set corresponding to six people shown in Table 11.1. We can compute the number of times that each different value of the input and output variables occur together as shown in the frequency table given in Table 11.2. Then, let's say that we have received a new test instance corresponding to a person who does not eat healthy foods (i.e., healthyFood = *no*) and wish to predict whether this person is developing cancer by applying the Bayes Theorem[4].

Table 11.1: An Illustrative Dataset With A Single Input Variable

x_1 (healthyFood)	y (cancer)
no	*yes*
no	*yes*
yes	*yes*
yes	*no*
yes	*no*
no	*no*

From Eq. 11.2, we have that

[4] We could potentially compute p(cancer = *yes*|healthyFood = *no*) and p(cancer = *no*|healthyFood = *no*) directly from the frequency tables without having to apply the Bayes Theorem. However, the intention of this example is to show that we can compute them by applying the Bayes Theorem, which will be useful to learn Naïve Bayes in the next section.

Table 11.2: An Illustrative Frequency Table For The Single Input Variable Dataset From Table 11.1

	cancer = *no*	cancer= *yes*	total
healthyFood = *no*	1	2	3
healthyFood = *yes*	2	1	3
total	3	3	6

$$p(\text{cancer} = yes|\text{healthyFood} = no) = \frac{p(\text{cancer} = yes)p(\text{healthyFood} = no|\text{cancer} = yes)}{p(\text{healthyFood} = no)}$$

and

$$p(\text{cancer} = no|\text{healthyFood} = no) = \frac{p(\text{cancer} = no)p(\text{healthyFood} = no|\text{cancer} = no)}{p(\text{healthyFood} = no)}.$$

Based on the frequency table given in Table 11.2, we have that

- $p(\text{cancer} = yes) = 3/6$, as 3 in 6 people had cancer;
- $p(\text{healthyFood} = no|\text{cancer} = yes) = 2/3$, as 2 in 3 of the people with cancer did not eat healthy foods;
- $p(\text{cancer} = no) = 3/6$, as 3 in 6 people did not have cancer;
- $p(\text{healthyFood} = no|\text{cancer} = no) = 1/3$, as 1 in 3 of the people with cancer ate healthy foods;
- $p(\text{healthyFood} = no) = 3/6$, as 3 in 6 people did not eat healthy foods.

Therefore,

$$p(\text{cancer} = yes|\text{healthyFood} = no) = \frac{3/6 \times 2/3}{3/6} \approx 0.67.$$

$$p(\text{cancer} = no|\text{healthyFood} = no) = \frac{3/6 \times 1/3}{3/6} \approx 0.33.$$

As $p(\text{cancer} = yes|\text{healthyFood} = no) > p(\text{cancer} = no|\text{healthyFood} = no)$, it would be reasonable to predict that the class is cancer = *yes*.

It is worth noting that the denominator $p(\mathbf{a})$ of the Bayes Theorem ($p(\text{healthyFood} = no)$ in the example above) works as a normalisation factor to ensure that the probabilities of the test instance to belong to each different possible class sums to one, i.e.:

$$\sum_{c \in \mathcal{Y}} p(c|\mathbf{a}) = 1$$

We could replace $p(\mathbf{a})$ by the factor β shown below and achieve the same normalisation effect:

$$\beta = \sum_{c \in \mathcal{Y}} p(c)p(\mathbf{a}|c).$$

If we set $\alpha = 1/\beta$, then the Bayes Theorem becomes:

$$p(c|\mathbf{a}) = \alpha p(c)p(\mathbf{a}|c). \tag{11.3}$$

In the next section, it will be useful to explicitly use α instead of $p(\mathbf{a})$, as the assumptions made by Naïve Bayes will cause $p(\mathbf{a})$ not to work as a normalising factor anymore, potentially resulting in probabilities that would not sum to 1 (i.e., in values that are not really probabilities).

11.2 Naïve Bayes for Categorical Input Variables

The Bayes Theorem can be written as follows for d input variables:

$$p(c|\mathbf{a}) = \alpha p(c)p(\mathbf{a}|c) = \alpha p(c)p(a_1, a_2, \cdots, a_d|c).$$

As the probability $p(a_1, a_2, \cdots, a_d|c)$ corresponds to the probability of a given combination of values (event) $a_1, a_2, \cdots, a_d$ for the input variables being observed together given a class c, each row of our frequency table would correspond to a different combination of values for the input variables. For example, consider a problem with two input variables, where the first input variable x_1 represents whether the person eats healthy foods and the second input variable x_2 represents whether this person is suffering from pain. Assume that we would like to predict whether a person is developing cancer (output variable y) based on these input variables. For a problem with the dataset shown in Table 11.3, we could have the frequency table shown in Table 11.4. For problems with large numbers of input variables and many possible values for such variables, the number of rows in this kind of frequency table would quickly become intractable.

Naïve Bayes is a machine learning approach that deals with this issue by making the assumption that the input variables are conditionally independent of each other given the output variable. A variable x_1 is conditionally independent of x_2 given the output variable y if the following condition is satisfied:

$$p(x_1|x_2, y) = p(x_1|y).$$

Table 11.3: An Illustrative Dataset With Two Input Variables

x_1 (healthyFood)	x_2 (pain)	y (cancer)
no	*yes*	*yes*
no	*yes*	*yes*
yes	*yes*	*yes*
yes	*no*	*no*
yes	*no*	*no*
no	*no*	*no*

Table 11.4: An Illustrative Frequency Table For The Two Input Variable Dataset From Table 11.3

	cancer = *no*	cancer= *yes*	total
healthyFood = *no* and pain = *no*	1	0	1
healthyFood = *no* and pain = *yes*	0	2	2
healthyFood = *yes* and pain = *no*	2	0	2
healthyFood = *yes* and pain = *yes*	0	1	1
total	3	3	6

This means that, if we know the value of y, we do not need to know the value of x_2 to determine the value of x_1. By making this assumption, we can say that:

$$P(a_1, a_2, \cdots, a_d|c) = \prod_{i=1}^{d} p(a_i|c).$$

By replacing this in the Bayes Theorem (Equation 11.3), we get:

$$p(c|\mathbf{a}) = \alpha p(c) \prod_{i=1}^{d} p(a_i|c), \tag{11.4}$$

where

$$\alpha = \frac{1}{\sum_{c \in \mathcal{Y}} \left(p(c) \prod_{i=1}^{d} p(a_i|c) \right)}.$$

Naïve Bayes uses Eq. 11.4 to calculate $p(c|\mathbf{a})$ in order to predict the class associated to a given example whose input variables have value $\mathbf{a}$. This means that a separate frequency table can be created for each input variable, leading to a total number of rows that grows linearly with the number of values that the input variables can assume.

For example, let's consider again our problem of predicting whether a person is developing cancer based on whether they eat healthy foods and are experiencing pain. The training set for this problem is shown in Table 11.3. The frequency tables corresponding to this training set would be the ones shown in Table 11.5.

Table 11.5: An Illustrative Frequency Table For The Two Input Variable Dataset From Table 11.3

(a) Frequency Table For x_1 (healthyFood)

	cancer = *no*	cancer= *yes*	total
healthyFood = *no*	1	2	3
healthyFood = *yes*	2	1	3
total	3	3	6

(b) Frequency Table For x_2 (Pain)

	cancer = *no*	cancer= *yes*	total
pain = *no*	3	0	3
pain = *yes*	0	3	3
total	3	3	6

Let's assume that we wish to predict whether a person who eats healthy foods (healthyFood = *yes*) and is experiencing pain (pain = *yes*) is developing cancer. We would need to compute the following:

$$p(\text{cancer} = yes|\text{healthyFood} = yes, \text{pain} = yes) =$$
$$\alpha p(\text{cancer} = yes) \times p(\text{healthyFood} = yes|\text{cancer} = yes) \times p(\text{pain} = yes|\text{cancer} = yes) =$$
$$\alpha \times 3/6 \times 1/3 \times 3/3 = \alpha \times 1/6$$

$$p(\text{cancer} = no|\text{healthyFood} = yes, \text{pain} = yes) =$$
$$\alpha p(\text{cancer} = no) \times p(\text{healthyFood} = yes|\text{cancer} = no) \times p(\text{pain} = yes|\text{cancer} = no) =$$
$$\alpha \times 3/6 \times 2/3 \times 0/3 = 0$$

where

$$\alpha = \frac{1}{3/6 \times 1/3 \times 3/3 + 3/6 \times 2/3 \times 0/3} = 6.$$

As $p(\text{cancer} = yes|\text{healthyFood} = yes, \text{pain} = yes) > p(\text{cancer} = no|\text{healthyFood} = yes, \text{pain} = yes)$, Naïve Bayes predicts that the person is developing cancer.

Note that in the calculations above, the whole probability of the person not developing cancer became zero, because $p(\text{pain} = yes|\text{cancer} = no) = 0$. In fact, even if there had been many other input variables in this problem, all with values suggesting that this person was not developing cancer, the whole probability of the person not developing cancer would still become

zero, just because $p(\text{pain} = yes|\text{cancer} = no) = 0$. This would lead to several inaccuracies on Naïve Bayes' predictions.

One way to avoid this problem is to adopt Laplace Smoothing. This consists in summing a small value $\epsilon > 0$ to every frequency value in the frequency table, and adjusting the total cells accordingly. And example for $\epsilon = 1$ is shown in Table 11.6.

Table 11.6: An Illustrative Frequency Table With Laplace Smoothing For The Two Input Variable Dataset From Table 11.3

(a) Frequency Table For x_1 (healthyFood)

	cancer = *no*	cancer= *yes*	total
healthyFood = *no*	1+1=2	2+1=3	5
healthyFood = *yes*	2+1=3	1+1=2	5
total	5	5	10

(b) Frequency Table For x_2 (Pain)

	cancer = *no*	cancer= *yes*	total
pain = *no*	3+1=4	0+1=1	5
pain = *yes*	0+1=1	3+1=4	5
total	5	5	10

When computing the probabilities for Eq. 11.4, the values of $p(a_i|c)$ should be computed based on the updated tables, whereas $p(c)$ is still calculated based on the original tables (without Laplace Smoothing) [5]. For instance, the calculations to predict whether a person who eats healthy foods and is experiencing pain would be as follows:

$$p(\text{cancer} = yes|\text{healthyFood} = yes, \text{pain} = yes) =$$

$$\alpha p(\text{cancer} = yes) \times p(\text{healthyFood} = yes|\text{cancer} = yes) \times p(\text{pain} = yes|\text{cancer} = yes) =$$

$$\alpha \times 3/6 \times 2/5 \times 4/5 = \alpha \times 8/50$$

$$p(\text{cancer} = no|\text{healthyFood} = yes, \text{pain} = yes) =$$

$$\alpha p(\text{cancer} = no) \times p(\text{healthyFood} = yes|\text{cancer} = no) \times p(\text{pain} = yes|\text{cancer} = no) =$$

$$\alpha \times 3/6 \times 3/5 \times 1/5 = 3/50$$

where

$$\alpha = \frac{1}{3/6 \times 2/5 \times 4/5 + 3/6 \times 3/5 \times 1/5} = 50/11.$$

[5] Try out Exercise 4 to understand why.

As $p(\text{cancer} = yes|\text{healthyFood} = yes, \text{pain} = yes) > p(\text{cancer} = no|\text{healthyFood} = yes, \text{pain} = yes)$, Naïve Bayes still predicts that the person is developing cancer in this example. However, in other examples the predicted class could potentially change when adopting Laplace Smoothing.

11.3 Naïve Bayes for Numeric Input Variables

The Naïve Bayes frequency tables explained in the previous section contain one row for each possible value of the input variables. When dealing with numeric input variables, it is infeasible to have one row for each possible numeric value. How can Naïve Bayes deal with numeric input variables?

To gain some insight into that, it is worth observing that, when we are collecting frequency values for the frequency tables and transforming them into probabilities, we are actually learning probability mass functions. These are functions that give the probability of observing each possible value of a discrete random variable [3, 4], e.g., of a categorical value in our machine learning problems. For instance, the figure below shows an example of probability mass function for $p(\text{healthyFood}|\text{cancer} = yes)$, obtained through the frequency table given in Table 11.6a.

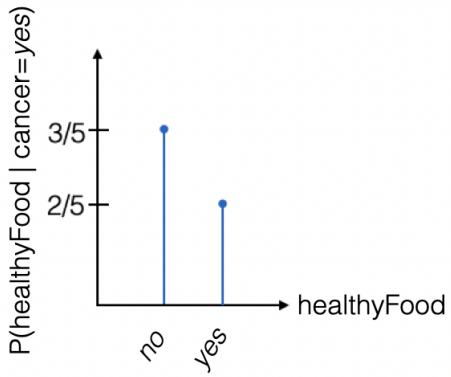


Fig. 11.1: Probability Mass Function Obtained Through The Frequency Table Given in Table 11.6a

For numeric input variables, we can learn probability density functions, instead of probability mass functions. Probability density functions represent the relative likelihood of observing different values of a given continuous random variable. Different from probability mass functions, the values of the relative likelihoods do not sum to one. Instead, the area under the curve formed by the function is equal to one [4].

To be able to adopt probability density functions in Naïve Bayes, one must choose what kind of probability density function to adopt. In most cases, a univariate Gaussian probability density function is used for each numeric input variable. This function can be written as follows for a given numeric input variable x_i:

$$p(x_i|\mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$$

where μ is a parameter corresponding to the mean of the distribution, σ^2 is a parameter corresponding to its variance, $\pi \approx 3.14159$ and $e \approx 2.71828$.

Learning then corresponds to setting appropriate values for the parameters μ and σ^2. In classification problems, every numeric input variable x_i needs to be associated to a probability density function $p(x_i|c)$ for each of the possible values $c \in \mathcal{Y}$. In order to learn the parameters of these probability density functions, the training examples can be used. In particular, for a Gaussian probability density function associated to the input variable x_i and the class value c, the mean μ is set as the mean of the x values values of all training examples whose class value is c. The variance σ^2 is set as the variance of these values. Considering the training set defined in Eq. 11.1 with N training examples, the mean and variance of a variable x_i is calculated as follows, respectively:

$$\mu = \frac{1}{N} \sum_{(a^{(j)}, c^{(j)}) \in \mathcal{T} \text{ and } c^{(j)}=c} a_i^{(j)},$$

and

$$\sigma^2 = \frac{1}{N-1} \sum_{(a^{(j)}, c^{(j)}) \in \mathcal{T} \text{ and } c^{(j)}=c} (a_i^{(j)} - \mu(x_i))^2.$$

Let's have a look at an example of how to calculate this. Assume that we need to predict whether a person is developing cancer (output variable y) based on whether they eat healthy foods (categorical input variable x_1) and on the amount of alcohol in units that they consume per month (numeric input variable x_2). The training set is shown in Table 11.7.

Table 11.7: An Illustrative Dataset With A Categorical and A Numeric Input Variable

x_1 (healthyFood)	x_2 (alcohol)	y (cancer)
no	40	*yes*
no	35	*yes*
yes	60	*yes*
yes	20	*no*
yes	30	*no*
no	17	*no*

Consider that we decide to use Gaussian probability density functions for the numeric input variable alcohol (x_2). As this is a binary classification problem (where $\mathcal{Y} = \{yes, no\}$ is a set of size 2), this means that we need to learn two Gaussian conditional probability density functions for alcohol: one for cancer = *yes* and one for cancer = *no*.

For cancer = *yes*, we calculate the mean and variance of the alcohol values of all training examples where cancer = *yes*:

$$\mu = \frac{40 + 35 + 60}{3} = 45,$$

$$\sigma^2 = \frac{1}{3-1}[(40-45)^2 + (35-45)^2 + (60-45)^2] = 175.$$

For cancer = *no*, we calculate the mean and variance of the alcohol values of all training examples where cancer = *no*:

$$\mu = \frac{20 + 30 + 17}{3} \approx 22.33,$$

$$\sigma^2 = \frac{1}{3-1}[(20-22.33)^2 + (30-22.33)^2 + (17-22.33)^2] \approx 46.34.$$

Therefore, one of the Gaussian conditional probability density functions is $p(\text{alcohol}|\mu = 45, \sigma^2 = 175)$ and the other is $p(\text{alcohol}|\mu = 22.33, \sigma^2 = 46.34)$. Note that we are omitting cancer = *yes* and cancer = *no* when we write $p(\text{alcohol}|\mu = 45, \sigma^2 = 175)$ and $p(\text{alcohol}|\mu = 22.33, \sigma^2 = 46.34)$ because the μ and σ^2 values already capture cancer = *yes* and cancer = *no*. However, we could also write cancer = *yes* and cancer = *no* explicitly if we wanted. These two functions are plotted in Figure 11.2.

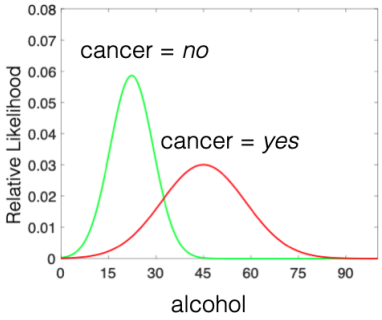


Fig. 11.2: Conditional Probability Density Functions For The Alcohol Variable Illustrative Dataset From Table 11.7. $P(\text{alcohol}|\text{cancer} = no)$ is shown in green and $P(\text{alcohol}|\text{cancer} = yes)$ is shown in red.

Once the parameters of the probability density functions are set, the values of $p(a_i|c)$ used in Equation 11.4 can be taken from these functions. It is worth noting, though, that these values are relative likelihood instead of probabilities. However, the resulting $p(c|a_i)$ will still be probabilities due to the use of the normalising factor α.

An example of prediction for this problem would be as follows. Consider that we wish to predict whether a person who does not eat healthy foods (healthyFood = *no*) and consumes 20 units of alcohol per month (alcohol = 20) is developing cancer. The frequency tables for x_1 (healthyFood) with Laplace smoothing, the total number of training examples for cancer = *yes* and cancer = *no* and the parameters of the conditional probability density functions for x_2 (alcohol) computed based on the training set are shown in Tables 11.6a, 11.8 and 11.9, respectively.

Table 11.8: Number of Training Examples From Each Class In The Training Set From Table 11.7

cancer = *no*	cancer = *yes*	total
3	3	6

Table 11.9: Parameters Of The Gaussian Conditional Probability Density Functions For The Training Set From Table 11.7

	cancer = *no*	cancer= *yes*
μ	22.33	45
σ^2	46.34	175

The probabilities $p(\text{cancer} = yes|\text{healthyFood} = no, \text{alcohol} = 20)$ and $p(\text{cancer} = no|\text{healthyFood} = no, \text{alcohol} = 20)$ are shown below:

$$p(\text{cancer} = yes|\text{healthyFood} = no, \text{alcohol} = 20) =$$
$$\alpha p(\text{cancer} = yes)p(\text{healthyFood} = no|\text{cancer} = yes)p(\text{alcohol} = 20|\text{cancer} = yes) =$$
$$\alpha \times 3/6 \times 3/5 \times 0.0051 = \alpha \times 0.00153 = 12.15\%$$

$$p(\text{cancer} = no|\text{healthyFood} = no, \text{alcohol} = 20) =$$
$$\alpha p(\text{cancer} = no)p(\text{healthyFood} = no|\text{cancer} = no)p(\text{alcohol} = 20|\text{cancer} = no) =$$
$$\alpha \times 3/6 \times 2/5 \times 0.0553 = \alpha \times 0.01106 = 87.85\%$$

where $\alpha = 1/(0.00153 + 0.01106) \approx 79.43$.

11.4 Strengths, Weaknesses and Applications of Naïve Bayes

Naïve Bayes has several strengths and has been successfully applied in many different domains. Strengths include:

- Training is fast, as it requires only one pass through the data.
- The relative probabilities computed by Naïve Bayes have achieved good results for making predictions for many applications, such as text categorisation (e.g., spam detection [6]), medical diagnosis [7], software defect prediction [8], among others.

Naïve Bayes' weaknesses include:

- It assumes conditional independence, which may be violated in many real world problems.
- Requires the choice of a probability density function for numeric input variables, which may not correspond to the true probability density function underlying the data.
- Despite being frequently good for prediction purposes, the probability values themselves outputted by Naïve Bayes may not be good estimates of the actual probabilities.

11.5 Summary and Discussion

Naïve Bayes is a simple probabilistic learning approach that has demonstrated success in many applications. Its learning process involves creating frequency tables that correspond to probability mass functions for categorical input variables given the values of the output variable, or parameters of probability density functions corresponding to numeric input variables given the values of the output variable. Laplace smoothing is typically adopted to avoid misclassifications resulting from frequencies of zero associated to certain input and output values in the training set. Based on these functions, on the Bayes Theorem and on the conditional independence assumption, Naïve Bayes can then compute the probabilities of examples to belong to each given class, given the observed values of their input variables. These probabilities can then be used to decide which class to predict for these examples.

11.6 Exercises

1. Consider the illustrative training set below, which was inspired by the real Heart Disease dataset from the UCI Machine Learning Repository[6].

x_1 (gender)	x_2 (smoker)	x_3 (painType)	y (heartDisease)
$female$	$smokes$	$angina$	yes
$male$	$smokes$	$angina$	yes
$male$	$doesnt$	$angina$	no
$male$	$smokes$	$nonangina$	no
$male$	$doesnt$	$nopain$	no
$female$	$doesnt$	$nopain$	no
$female$	$doesnt$	$angina$	yes

   Create the frequency tables that compose the Naïve Bayes classifier for this training set using Laplace Smoothing with $\epsilon = 1$.
2. Draw a plot for the probability mass function $p(\text{painType}|\text{heartDisease} = no)$ represented by the frequency table for the input variable painType created in Exercise 1.
3. Determine the prediction that Naïve Bayes would give to the instance (gender = $female$, smoker = $smokes$, painType = $nonangina$, heartDisease=?) using Laplace Smoothing with $\epsilon = 1$, given the training set from Exercise 1.
4. Compute the probability $p(\text{heartDisease} = yes)$ based on the three different smoothed frequency tables that you have obtained in Exercise 3, rather than based on the original frequency values of each class. Reflect about the results.

11.7 Exercise Answers

1.

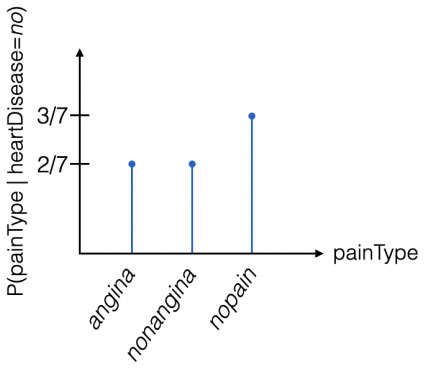
	heartDisease=yes	heartDisease=no	total
gender=$female$	3	2	5
gender=$male$	2	4	6
total	5	6	11

	heartDisease=yes	heartDisease=no	total
smoker=$smokes$	3	2	5
smoker=$doesnt$	2	4	6
total	5	6	11

[6] `https://archive.ics.uci.edu/ml/datasets/heart+disease`

	heartDisease=*yes*	heartDisease=*no*	total
painType=*angina*	4	2	6
painType=*nonangina*	1	2	3
painType=*nopain*	1	3	4
total	6	7	13

2. The probability mass function is given below:



3. In order to predict whether the person does or does not have disease, we need to compute $p(yes|female, smokes, nonangina)$ and $p(no|female, smokes, nonangina)$. Then, we give a prediction corresponding to the higher probability.
For compute these probabilities, we need to instanciate Eq. 11.4:

$$p(yes|female, smokes, nonangina) =$$
$$= \alpha P(yes)P(female|yes)P(smokes|yes)P(nonangina|yes)$$
$$= \alpha * 3/7 * 3/5 * 3/5 * 1/6$$
$$\approx \alpha * 0.0257 = 1/(0.0257 + 0.01814) * 0.0257 \approx 0.59$$

$$P(no|female, smokes, nonangina) =$$
$$= \alpha * P(no)P(female|no)P(smokes|no)P(nonangina|no)$$
$$= \alpha * 4/7 * 2/6 * 2/6 * 2/7$$
$$\approx \alpha * 0.01814 = 1/(0.0257 + 0.01814) * 0.01814 \approx 0.41$$

The $p(a|c)$ probabilities above are computed based on the frequency values from Exercise 1's answer. For example, $P(female|yes) = 3/5$ because, according to the first column of the first table, 3 in 5 people who had disease were female. The probabilities $p(c)$ are still calculated based on the original frequencies (without Laplace smoothing).
As $p(yes|female, smokes, nonangina) > p(no|female, smokes, nonangina)$, Naïve Bayes would predict *yes*, the person has the disease. However, as

$p(yes|female, smokes, nonangina)$ is very close to $p(no|female, smokes, nonangina)$, we can consider that Naïve Bayes is not very certain on the class being *yes*.

4. We have three input variables. Two of them (gender and smoker) can assume two possible values ($female/male$ and $smokers/doesnt$). However, the other input variable (painType) can assume three possible values ($angina/nonangina/nopain$). If we compute $p(\text{heartDisease} = yes)$ based on the first two smoothed tables, we get the value of $5/11 \approx 0.4545$. However, if we compute $p(heartDisease = yes)$ based on the third smoothed table, we get the value $6/13 \approx 0.4615$. As we can see, these values are not the same, hindering the calculation of $p(yes)$.

References

1. Z. Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521:452–459, 2015.
2. T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, second edition, 2017.
3. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, third edition, 2010.
4. M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.
5. A. Bifet, R. Gavalda, G. Holmes, and B. Pfahringer. *Machine Learning for Data Streams: With Practical Examples in MOA*. MIT Press, 2018.
6. M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk e-mail. In *AAAI Workshop on Learning for Text Categorization*, 1998.
7. I. Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in Medicine*, 23(1):89–109, 2001.
8. B. Turhan and A. Bener. Analysis of naive bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, 68(2):278–290, 2009.

Chapter 12
Evaluation of Supervised Learning Models

George G. Cabral and Leandro L. Minku

As explained in the introduction to Part III of this book, supervised learning aims at creating functions (models) able to predict output variables given the values of the input variables. Such functions should be able to generalize to new examples that are unseen at training time. If a predictive model works well on the training data, this does not mean that it works well for unseen data. But how to evaluate how well a given predictive model is likely to perform on unseen data? This chapter will explain a number of performance metrics (Section 12.1) and evaluation procedures (Section 12.2) that can be used for this purpose.

12.1 Evaluation Metrics

This section explains some evaluation metrics that can be computed over a set of examples to show how well a given predictive model performs on this set of examples. In other words, they can be used to compute the predictive performance of a given model on a set of examples. Section 12.1.1 presents evaluation metrics for classification problems, whose output variables are categories. Section 12.1.2 presents evaluation metrics for regression problems, whose output variables are numeric.

12.1.1 Classification Problems

Consider a set of examples with known output values

G. Cabral is with the Department of Computing, Federal Rural University of Pernambuco, BR. L. Minku is with the School of Computer Science, University of Birmingham, UK.

$$\mathcal{E} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(M)}, y^{(M)})\},$$

where $\mathbf{x}^{(i)} \in \mathcal{X}$ are the input variables coming from any input domain $\mathcal{X}$, $y^{(i)} \in \mathcal{Y}$ is the output variable coming from the categorical domain $\mathcal{Y}$, $1 \leq i \leq M$, and M is the number of examples. Consider also the predictions $\{\hat{y}^{(1)}, \hat{y}^{(2)}, \cdots, \hat{y}^{(M)}\}$ given by a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ to these examples. Each prediction $\hat{y}^{(i)}$ can be either correct or incorrect. It is correct when $\hat{y}^{(i)} = y^{(i)}$ and incorrect when $\hat{y}^{(i)} \neq y^{(i)}$.

When we are dealing with binary classification problems (i.e., problems where $\mathcal{Y}$ is a set containing two categories), it is common to refer to one of the categories as the "positive" category and the other one as the "negative" category. A matrix called *confusion matrix* can be used to count the number of positive and negative examples from $\mathcal{E}$ that are correctly and incorrectly classified by the function f being evaluated as follows:

Confusion Matrix:

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

The true positive (TP) is the number of examples that are predicted as positive and are actually positive. The false positive (FP) is the number of examples that are predicted as negative but are actually positive. The True Negative (TN) is the number of examples that are predicted as negative and are actually negative. The False Negative (FN) is the number of examples that are predicted as negative but are actually positive.

The confusion matrix can give an idea of how well the function f is predicting examples in $\mathcal{E}$. However, it is typically easier to interpret the predictive performance of a function when we represent the values above in the form of ratios, or when we aggregate these values into single metric values. We explain below some examples of evaluation metrics that can be computed based on the confusion matrix.

Accuracy is the proportion of examples that are correctly classified. It can be calculated as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}.$$

This metric has values in $[0, 1]$, with 0 being the worst possible value and 1 being the best possible value. However, this metric is only suitable when the number of positive and negative examples in $\mathcal{E}$ is similar. When that is not the case (i.e., when the data are *class imbalanced*), this metric focuses on the predictions given for examples of the majority class. Therefore, a function f that is able to predict the majority class very well, but fails to correctly identify examples of the minority class, gets a misleadingly high value for the accuracy metric. For example, consider a problem where $\mathcal{E}$ contains 100 examples – 10 belonging to the positive class and 90 belonging

to the negative class. Consider that f always predicts negative, no matter the values of the input variables of the examples. The accuracy of f would be $(90 + 0)/100 = 90\%$, which appears to be a very good value. However, f is not a reasonable predictive model, as it always predicts the same category regardless of the example being predicted.

Classification Error is the opposite of Accuracy:

$$\text{Classification Error} = 1 - \text{Accuracy}.$$

This metric has values in $[0, 1]$, with 1 being the worst possible value and 0 being the best possible value.

Precision is the proportion of positive predictions that are correct predictions. It can be calculated as follows:

$$\text{Precision} = \frac{TP}{TP + FP}.$$

This metric has values in $[0, 1]$, with 0 being the worst possible value and 1 being the best possible value. This metric has also been discouraged for class imbalanced problems, as it is a biased metric [1].

Recall (a.k.a., True Positive Rate (TPR) or Sensitivity) is the proportion of positive examples that are successfully predicted as positive. It can be calculated as follows:

$$TPR = \frac{TP}{TP + FN} = 1 - FNR,$$

where FNR is the False Negative Rate:

$$FNR = \frac{FN}{TP + FN}.$$

Recall has values in $[0, 1]$, with 0 being the worst possible value and 1 being the best possible value, whereas FNR has values in $[0, 1]$, with 1 being the worst possible value and 0 being the best possible value. Recall and FNR are suitable metrics to be used no matter if the data are class imbalanced or not. However, they only represent how well f performs on the positive category. Therefore, they should never be reported in isolation. Other metrics to also capture performance on the negative category, such as the specificity explained below, should be reported to complement them.

Specificity (a.k.a., True Negative Rate) is the proportion of negative examples that is successfully predicted as negative. It can be calculated as follows:

$$TNR = \frac{TN}{TN + FP} = 1 - FPR,$$

where FPR is the False Positive Rate (a.k.a., false alarm rate):

$$FPR = \frac{FP}{TN + FP}.$$

Specificity has values in $[0, 1]$, with 0 being the worst possible value and 1 being the best possible value, whereas FPR has values in $[0, 1]$, with 1 being the worst possible value and 0 being the best possible value. Similar to recall and FNR, Specificity and FPR are suitable metrics to be used no matter if the data are class imbalanced or not, but should be complemented by the report of other metrics to also capture performance on the positive category.

F1-Score is the harmonic mean between precision and recall:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

It has values in $[0, 1]$, with 0 being the worst possible value and 1 being the best possible value. F1-Score is a useful metric when one needs to aggregate precision and recall into a single value. However, similar to Precision, F1-Score is also a biased metric that has been discouraged for class imbalanced data [1].

G-Mean is the geometric mean between recall and specificity:

$$\text{G-Mean} = \sqrt{TPR \cdot TNR}.$$

It has values in $[0, 1]$, with 0 being the worst possible value and 1 being the best possible value. Similar to Recall and Specificity, G-Mean is suitable for both class balanced and imbalanced data. It is a useful metric when one needs to aggregate the performance on the positive and negative categories into a single unbiased metric.

When dealing with multi-class classification problems (i.e., problems where $\mathcal{Y}$ is a set containing more than two categories), the confusion matrix can be generalized as follows, where c_i, represents a given category in $\mathcal{Y}$, $i \in \{1, 2, \cdots, n\}$, n is the number of categories, and $C_{j,k}$ is the number of examples of category c_j that were predicted by f as being of category c_k, $\forall j, k \in \mathcal{Y}$:

	Predicted c_1	Predicted c_2	$\cdots$	Predicted c_n
Actual c_1	$C_{1,1}$	$C_{1,2}$	$\cdots$	$C_{1,n}$
Actual c_2	$C_{2,1}$	$C_{2,2}$	$\cdots$	$C_{2,n}$
$\vdots$	$\ddots$		$\cdots$	$\vdots$
Actual c_n	$C_{n,1}$	$C_{n,2}$	$\cdots$	$C_{n,n}$

Metrics for binary classification problems can also be generalized to multi-class problems.

The following correspond to the overall accuracy, and to the precision, recall and specificity for a given category c_i:

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} C_{i,i}}{\sum_{j,k=1}^{n} C_{j,k}},$$

$$\text{Precision}_{c_i} = \frac{C_{i,i}}{\sum_{j=1}^{n} C_{j,i}},$$

$$\text{Recall}_{c_i} = \frac{C_{i,i}}{\sum_{j=1}^{n} C_{i,j}},$$

$$\text{Specificity}_{c_i} = \frac{\sum_{j \neq i} C_{j,j}}{\sum_{j \neq i,k} C_{j,k}}.$$

Based on these, the F1-Score for a given category c_i and the overall G-Mean for all categories can be computed as follows:

$$\text{F1-Score}_{c_i} = 2 \cdot \frac{\text{Precision}_{c_i} \cdot \text{Recall}_{c_i}}{\text{Precision}_{c_i} + \text{Recall}_{c_i}},$$

$$\text{G-Mean} = \sqrt[n]{\prod_{i=1}^{n} C_{i,i}}.$$

12.1.2 Regression Problems

Consider a set of examples with known output values

$$\mathcal{E} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(M)}, y^{(M)})\},$$

where $\mathbf{x}^{(i)} \in \mathcal{X}$ are the input variables coming from any input domain $\mathcal{X}$, $y^{(i)} \in \mathcal{R}$ is a real-valued output variable, $1 \leq i \leq M$, and M is the number of examples. Consider also the predictions $\{y'^{(1)}, y'^{(2)}, \cdots, y'^{(M)}\}$ given by a function $f : \mathcal{X} \to \mathcal{R}$ to these examples. Different from classification problems, instead of considering predictions as "correct" or "incorrect", it is common practice to check how different the predicted value is from the actual one, i.e., how large the error of the predictions is. We show below some evaluation metrics that can be used to evaluate f based on $\mathcal{E}$.

Mean Absolute Error (MAE) is the average absolute value of the difference between the predicted and actual outputs:

$$MAE = \frac{1}{M} \sum_{i=1}^{M} |y^{(i)} - \hat{y}^{(i)}|.$$

This metric has positive or zero values, where the smaller the value the better.

Mean Squared Error (MSE) is the average squared value of the difference between the predicted and actual outputs:

$$MSE = \frac{1}{M}\sum_{i=1}^{M}(y^{(i)} - \hat{y}^{(i)})^2.$$

This metric has positive or zero values, where the smaller the value the better. The squared function has the effect of emphasizing larger differences between the actual and predicted output values. However, the use of the squared function makes the values of this metric more difficult to interpret, as their unit of measurement is different from the unit of the output variable. The metric presented next overcomes this problem.

Root Mean Squared Error (RMSE) is the squared root of the MSE:

$$RMSE = \sqrt{\frac{1}{M}\sum_{i=1}^{M}(y^{(i)} - \hat{y}^{(i)})^2}.$$

This metric has positive or zero values, where the smaller the value the better.

The metrics above are measured in the unit of the output variable (or the square unit, in the case of MSE), which may not be very easy to interpret depending on the problem. The metrics explained below may be helpful to improve interpretability.

Mean Absolute Percentage Error (MAPE) is the absolute value of the difference between the actual and predicted output values as a proportion of the actual value:

$$MAPE = \frac{1}{M}\sum_{i=1}^{M}\left|\frac{y^{(i)} - \hat{y}^{(i)}}{y^{(i)}}\right|.$$

This metric has positive or zero values. Values smaller/larger than one correspond to prediction errors that are in general smaller/larger than the actual output values. The smallest the value of this metric, the better. However, this metric may place less emphasis on errors performed on examples whose output values are larger. This is because larger output values will result in a larger denominator, and as a result a smaller relative error.

Relative Absolute Error (RAE) is a measure of the error relative to a simple model that would predict the average of the actual output values of the examples in $\mathcal{E}$:

$$RAE = \frac{\sum_{i=1}^{M}|y^{(i)} - \hat{y}^{(i)}|}{\sum_{i=1}^{M}|y^{(i)} - \bar{y}|},$$

where $\bar{y}$ is the average of the actual outputs:

$$\bar{y} = \frac{1}{M}\sum_{i=1}^{M} y^{(i)}.$$

This metric has values that are positive or zero. RAE's numerator represents the MAE of the function f being evaluated, whereas RAE's denominator represents the MAE of the simple model. Therefore, values smaller/larger than 1 mean that the predictions are in general better/worse than those provided by the simple model.

Root Relative Squared Error (RRSE) is also a measure of the error relative to a simple model that would predict $\bar{y}$. However, similar to RMSE, it further emphasizes larger errors by squaring the differences between the actual and predicted output values:

$$RRSE = \sqrt{\frac{\sum_{i=1}^{M}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{M}(y^{(i)} - \bar{y})^2}}.$$

This metric also has values that are positive or zero, where smaller values are better values. Values that are smaller/larger than 1 mean that the predictions are in general better/worse than those provided by the simple model.

12.2 Evaluation Procedures

During the modeling phase of a problem, i.e., when training the predictive model, practitioners have access to a limited amount of training data (i.e., a subset of examples of the problem) and the aim is to create a predictive model that behaves well when deployed (i.e., that can generalize well to unseen data). In an unreal scenario, practitioners would have all possible data available during the modeling (training) phase. In such unreal scenario, theoretically, one could learn a classifier able to produce the best achievable predictive performance (e.g., accuracy of 100% or MSE of 0). However, in real world problems, the data shortage for modeling the problem is often an issue that must be seriously taken into account. Moreover, data typically includes noisy examples, whose input or output values have been potentially collected with some mistakes. Learning algorithms should try to make the most of the available data while avoiding the negative effect of noise.

Figure 12.1 presents a bi-dimensional two-class dataset where each of the classes contains 20 examples to illustrate this. Assume that these data are used as the training set to learn a classifier. By taking a visual analysis of this dataset we can see some particularities that, intuitively, would be beneficial if incorporated or not incorporated into the classifier:

- Example 20 of the orange class may be interpreted as a noise. It may be desirable for the classifier not to incorporate it into its learned function.
- Examples 14, 8 and 15 of the orange class are placed in a conflict area. It may be desirable for the classifier to be capable of separating these examples from those of the blue class.

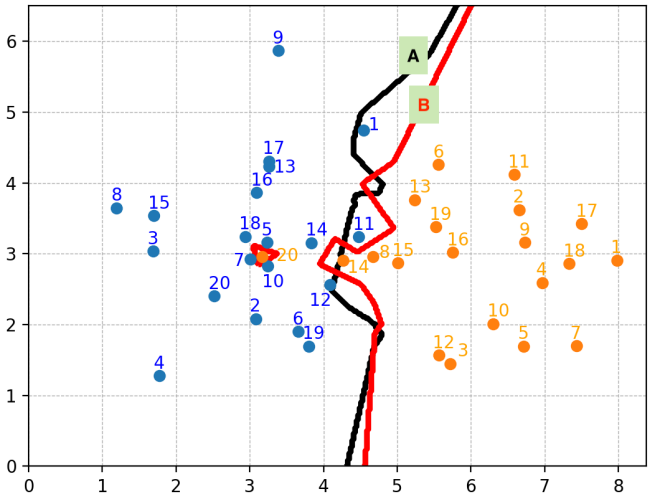


Fig. 12.1: Hypothetical 2-class dataset containing 20 examples for each class.

As this is a two dimensional problem (with only two input variables), one could attempt to plot the decision boundary learned by the predictive model to get an idea of how well it may generalize to unseen data. The decision boundary are lines learned by a predictive model to separate examples of different classes in a classification problem. As we can see from Figure 12.1, classifier A was able to make the most of the data while preventing the negative effect of example 20. This classifier is likely to generalize well. However, classifier B learned the noise of example 20. This classifier may not generalize well. Similarly, predictive models for regression problems can also be affected by noise.

In contrast to the dataset presented in Figure 12.1, most real world problems are composed of several input variables. For these problems we cannot visually assess how good the learned model is. Instead, we need to compute the predictive performance based on a set of examples using evaluation metrics such as the ones presented in Section 12.1. However, if we compute the evaluation metrics on the training set, we would be considering a predictive model to be good when it learns noise. For instance, classifier B makes no mistakes on any training example and would get the best possible predictive performance on the training set. In contrast, classifier A misclassifies example 20 and would get worse predictive performance on the training set. Nevertheless, classifier A should be considered as better than classifier B in terms of its generalization capability.

Evaluation procedures can be used to train predictive models and evaluate their generalization capabilities by splitting the available data into different sets for training and evaluation (testing) purposes. Different evaluation procedures create such splits in different ways. Sections 12.2.1, 12.2.2 and 12.2.3 present three popular evaluation procedures.

12.2.1 Stratified k-fold Cross-Validation

As explained above, when building a classifier, it is imperative that the data used to train the classifier is not used to assess its performance. If a classifier learns a supervised problem represented by a training set $\tau = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(M)}, y^{(M)})$, examples contained in τ may be well learned, but examples not present in τ may still be misclassified. Therefore, it is important to separate the available data into distinct datasets called training and test datasets. The idea is to train a classifier on the training dataset and validate its predictive performance on the test dataset. However, depending on the split of the data used to create the training and test sets, a different predictive model and a different estimation of the generalization capability may be obtained.

For example, let's consider the dataset in Figure 12.1 and suppose that the examples (orange class: 20, 14, 8, 15, 13, and 6) were used for testing a classifier f whereas the remainder of the orange examples were used to train f. In such case, some key examples were left behind in the learning phase and the decision surface may be negatively affected. Had example 14 been included in the training set, a better model could have been obtained. Similarly, the predictive performance calculated on the test set may also be influenced by the examples included in this set. The estimate of the generalization of the predictive model may assume a larger or smaller value depending on which examples are included in the test set.

The idea of $k-$fold cross-validation is to systematically create several different splits of the data into training and test sets, such that each example is used once for testing. By ensuring that each example is used once for testing and by using a large enough value for k so that models can be trained on more examples, a less biased estimate of generalization ability may be obtained. To apply this procedure, the order of the whole available data is randomized. Then, the data are divided into k subsets called folds. The classifier can then be trained on $k-1$ folds and tested on the remaining fold. This process is iterated k times, where each iteration selects a different fold for testing. In a 10-fold cross validation procedure, for example, the available data is divided into ten folds. Then 10 classifiers are learned (based on 9 folds each time) and tested in the fold not used for training, as depicted in Figure 12.2. The estimation of the performance of a classifier that would be trained on the full

dataset is then computed as the average test performance of the classifiers trained on each of the k iterations, as depicted in Equation 12.1.

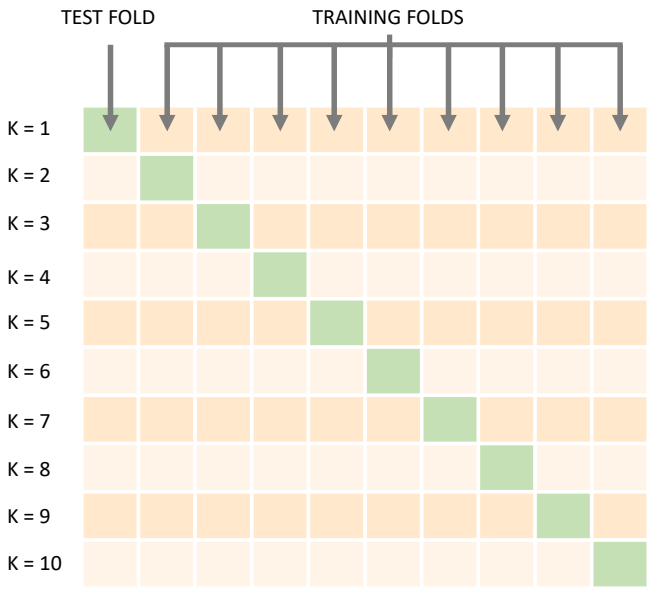


Fig. 12.2: 10-fold cross validation scheme. The test fold is shown in green, whereas the training folds are shown in (lighter or darker) orange.

$$Perf = \frac{1}{k'} \sum_{k'=1}^{k} Perf(k') \tag{12.1}$$

where $Perf(k')$ represents any performance metric that is suitable for the problem in hands, computed on the test fold k', using the classifier trained on folds $k'' \in \{1, 2, \cdots, k\} \backslash k'$.

Note that k-fold Cross Validation does not guarantee that the best training examples will be chosen to build the classifier, however, it can be repeated as many times as necessary to reduce the chances of picking examples that do not represent well the problem. It is also common to create the splits such that each fold preserves the same proportion of examples of each class as the proportion presented by the full data set. This process is called stratification and may help to improve the estimation of generalization.

12.2.2 Leave-one-out cross-validation (LOOCV)

Leave-one-out Cross Validation (LOOCV) can be defined as a particular case of the k-fold Cross Validation (Section 12.2.1) where the value of k is set to n, i.e., the total number of available labeled examples. For large datasets, using the LOOCV procedure means to create a large number of classifiers, which may be very time consuming. For small datasets, this procedure may have a feasible computational cost while enabling the models to be trained on larger training sets than when $k < n$.

Table 12.1 shows an example of dataset containing 12 different tonalities of the colors red, green and blue. Considering this data, creating a classifier that properly generalizes this problem may be a challenging task and the k-fold cross validation using typical values for the k may be unsuitable. Conversely, for this problem, the LOOCV will build twelve classifiers (f) such that each of these will be trained using 11 training examples and will be tested on the remaining one. The estimate of the generalization can then be calculated based on Eq. 12.1 with $k = 12$.

Table 12.1: Dataset containing different tonalities for the RGB color pattern.

R	G	B	Class
134	10	20	Red
210	31	5	Red
198	15	2	Red
189	8	21	Red
11	214	3	Green
7	193	20	Green
20	207	4	Green
13	221	20	Green
3	17	179	Blue
21	10	200	Blue
12	18	215	Blue
15	4	196	Blue

12.2.3 Repeated Hold-out Validation

The Hold-out Validation procedure simply consists in randomly selecting a given number of examples for training and using the remaining ones for testing. The training examples are used to train a predictive model f, whereas the test examples are used to estimate its generalization capability. However, unless the dataset is very large, using a single split of the data into training and test sets would mean that the estimate of the generalization is likely

biased. By chance, the split may have been a "lucky" one that results in a good value for the generalization estimate, when in reality the performance on other unseen examples would not be so good. Conversely, the split may have been an "unlucky" one that results in a poor value for the generalization estimate, when in reality the performance on other unseen examples would not be so poor.

In an attempt to overcome this issue, the Hold-out Validation procedure is typically repeated multiple times with different random splits of the data into training and test sets. The estimate of the generalization of a model trained on the full dataset is then set as the average of the test performances obtained on all repetitions. A disadvantage of this procedure compared to $k-$fold Cross Validation is that it does not ensure that every example is used for testing once. Some examples may be used multiple times for testing, whereas others may not be used at all. This could potentially lead to a more biased estimate of generalization when the dataset is not very large.

12.3 Summary and Discussion

This chapter presented evaluation procedures that can be used to evaluate the generalization ability of a predictive model f learned by a given machine learning algorithm with a fixed and pre-defined set of hyperparameter values (when applicable) [1]. Several different performance metrics that can be used with these evaluation procedures have also been presented, where each metric captures and/or focuses on different performance aspects. Several other performance metrics exist. As future reading, you may wish to learn about the Area Under the Receiver Operating Characteristic Curve (AUC) [2].

The evaluation procedures presented in this chapter are for estimating generalization. One may think of using the average test performance metrics reported by these procedures for model selection, i.e., to choose among different models created by different machine learning algorithms and/or hyperparameter values. This is possible. However, it is important to note that once the average performance metric values computed based on certain test data are used to select a model, they do not work as measures of the generalization capability of this model on unseen data anymore. This is because the model was chosen so as to optimize the value of predictive performance computed on the given test data, i.e., we chose the model (among a set of models) that does best on those test data. Therefore, such average test performance can become biased towards this chosen model [3]. Another separate test set that was used neither for training nor for model selection would thus

[1] Hyperparameters are parameters that have to be set before running the learning algorithm. For example, the learning rate used by backpropagation for multilayer perceptrons is a hyperparameter.

be necessary to further evaluate the generalization capabilities of the selected model.

12.4 Exercises

1. Consider the following data set:

$$\mathcal{E} = \{([1, 2, 3], \text{positive}), ([1, 1, 1], \text{negative}), ([3, 3, 3], \text{positive})\}.$$

   Consider also a classifier function f that gives the following predictions to these examples:

$$\{\text{negative}, \text{positive}, \text{positive}\}.$$

   Calculate all the performance metrics presented in Section 12.1.1 of this chapter. Reflect about the differences in the results obtained when using different metrics.
2. Consider the following data set:

$$\mathcal{E} = \{([1, 2, 3], 1), ([1, 1, 1], 0.1), ([3, 3, 3], 7)\}.$$

   Consider also a classifier function f that gives the following predictions to these examples:

$$\{2, 4.1, 8\}.$$

   Calculate all the performance metrics presented in Section 12.1.2 of this chapter. Reflect about the differences in the results obtained when using different metrics.
3. Given the widely known dataset Iris, a kNN classifier with $k = 3$, and the accuracy evaluation metric, perform: (i) 10-fold cross validation, (ii) Leave-one-out cross validation and (iii) repeated hold-out validation with 30 repetitions. Analyze the results for each procedure in terms of predictive performance and training time. You may use WEKA as a machine learning tool: `https://www.cs.waikato.ac.nz/ml/weka/`.

12.5 Answers

1. The performance values are as follows:

$$\text{Accuracy} = \frac{1}{1 + 1 + 0 + 1} \approx 0.33$$

$$\text{Classification Error} \approx 1 - 0.33 = 0.67$$

$$\text{Precision} = \frac{1}{1+1} = 0.5$$

$$TPR = \frac{1}{1+1} = 0.5$$

$$FNR = \frac{1}{1+1} = 0.5$$

$$TNR = \frac{0}{0+1} = 0$$

$$FPR = \frac{1}{0+1} = 1$$

$$\text{F1-Score} = 2 \cdot \frac{0.5 \cdot 0.5}{0.5+0.5} = 0.5$$

$$\text{G-Mean} = \sqrt{0.5 \cdot 0} = 0$$

Note that there are twice more positive than negative examples in $\mathcal{E}$, i.e., there is class imbalance. As most examples were misclassified, the accuracy and classification errors are poor. However, we can see that the performance on the negative class (TNR) is much worse than that on the positive class (TPR). In particular, no example of the negative class was correctly classified, leading to very low TNR. Correspondingly, the FPR was also very high. The performance on the positive class (TPR) was also poor, but not so poor (TPR = 0.5, FNR = 0.5). As the performance on the negative class was so poor, this had a dramatic impact on the value of the G-Mean (G-Mean = 0), though the impact on the F1-Score was not so large.

2. The performance values are as follows:

$$MAE = \frac{1}{3}[|1-2| + |0.1-4.1| + |7-8|] = 2$$

$$MSE = \frac{1}{3}[(1-2)^2 + (0.1-4.1)^2 + (7-8)^2] = 6$$

$$RMSE = \sqrt{6} \approx 2.45$$

$$MAPE = \frac{1}{3}\left[\left|\frac{1-2}{1}\right| + \left|\frac{0.1-4.1}{0.1}\right| + \left|\frac{7-8}{7}\right|\right] = 14$$

$$RAE = \frac{|1-2| + |0.1-4.1| + |7-8|}{|1-2.7| + |0.1-2.7| + |7-2.7|} \approx 0.70$$

$$RRSE = \sqrt{\frac{(1-2)^2 + (0.1-4.1)^2 + (7-8)^2}{(1-2.7)^2 + (0.1-2.7)^2 + (7-2.7)^2}} \approx 0.79$$

Note how the large error on the second example was magnified by the RMSE and RRSE compared to the MAE and RAE metrics. Note also

how the relative error on the first example ($|(1-2)/1| = 1$) was much larger than the relative error on the third example ($|(7-8)/7| = 0.14$) when calculating MAPE, despite the absolute errors (differences between the actual and predicted values) being the same for these two examples.

References

1. A. Luque, A. Carrasco, A. Martin, and A. de las Heras. The impact of class imbalance in classification performance metrics based on the binary confusion matrix. *Pattern Recognition*, 91:216–231, 2019.
2. T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006.
3. G. C. Cawley and N. L. C. Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11:2079–2107, 2010.

Part III-(B)
Unsupervised Learning

Chapter 13
k-Means

Harish Tayyar Madabushi

The k-means clustering algorithm (k-means for short) provides a method of finding *structure* in input examples. It is also called the Lloyd–Forgy algorithm as it was independently introduced by both Stuart Lloyd [1] and Edward Forgy [2]. k-means, like other algorithms you will study in this part of the book, is an *unsupervised learning* algorithm and, as such, does not require labels associated with input examples. Recall that unsupervised learning algorithms provide a way of discovering some inherent structure in the input examples. This is in contrast with supervised learning algorithms, which require input examples *and* associated labels so as to fit a hypothesis function that maps input examples to one or more output variables.

While there are different *structures* that can be extracted from input examples the most intuitive is a *cluster*. Very simply, a cluster is a set of examples that are grouped together, often as a consequence of those examples sharing some similarities. The k-means algorithm is one method of finding clusters in input examples. It is important to note that k-means requires as input the number of clusters (k) and the method of selecting this number is not always obvious. While there are methods of estimating a 'good' k, some of which we will discuss later in Section 13.5, this might not always be feasible and, in such instances, one might use algorithms which do not require selecting a value of k. You will study some of these other algorithms, such as DBScan, in subsequent chapters of this book. Although clustering provides us with methods of grouping input examples, it does not provide information on the relationship between these resulting groups. For those cases where we might require such relations, one might use algorithms that can extract what is called connectivity information such as "Hierarchical Clustering", yet another unsupervised learning algorithm that is also detailed in subsequent chapters.

This chapter begins by exploring some applications of clustering algorithms, of which k-means is by far the most favored because it is simple,

The University of Bath, UK

intuitive and also effective. The chapter then provides an intuitive overview of the algorithm, before stepping into the details of its implementation including its optimization objective and computational complexity. Finally, we explore some methods of picking k, the number of clusters, before then exploring possible extensions of this algorithm. At the end of this chapter you will

(a) know what k-means is and when to use it,
(b) know how to implement k-means, and
(c) know how to interpret the results.

13.1 Motivation and Applications

Why might one want to cluster input examples together and where might such clustering come in handy? Consider the scenario wherein a manufacturer of sweaters is attempting to come up with some fixed sizes. Garment manufacturers must come up with a fixed (and feasible) number of sizes in which to offer their products to ensure that the manufacturing process is tractable. One method of doing this might be to sample the height and weight of potential customers and group them by those who can wear the same size. Notice how, in this example, the height and weight are the input variables and clusters of input examples will provide an estimate of the heights and weights of people who might fit into the same size.

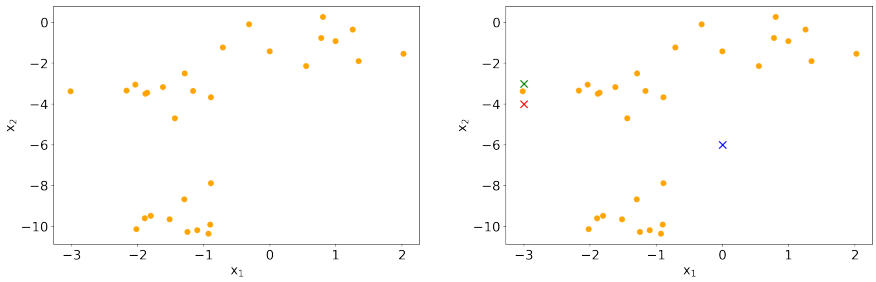
The applications of extracting structure from data are wide ranging: online retailers, for example, might want to cluster customers into groups who are shown the same offers, computer files that are often required together might be clustered so they can be loaded together, thus reducing load time. Similarly, students might be clustered based on performance so certain groups can be provided additional support.

Clustering is also used to analyze and group DNA sequences extracted from fragments of biological material [3] to allow biologists to both identify species and also find relationships between species' and where a species fits in the overall taxonomy of biological organisms. Notice that in this case, we require methods that additionally provide information regarding the connections between clusters, such as hierarchical-clustering. As such, it is important to choose an algorithm based on the nature of the problem at hand.

13.2 An Intuitive Overview of *k*-means

Consider a set of input examples each consisting of two input variables as illustrated by Figure 13.1a. Be mindful of the fact that both the axes of the figure represent input variables (neither of them are output variables in

contrast with supervised learning). The goal is to cluster these examples into k clusters (i.e., $k = 3$). Take note of the input examples illustrated in the figure below. Think about what the resultant clusters might look like and if 3 is a reasonable value for k.



(a) Input examples

(b) with cluster centroids added.

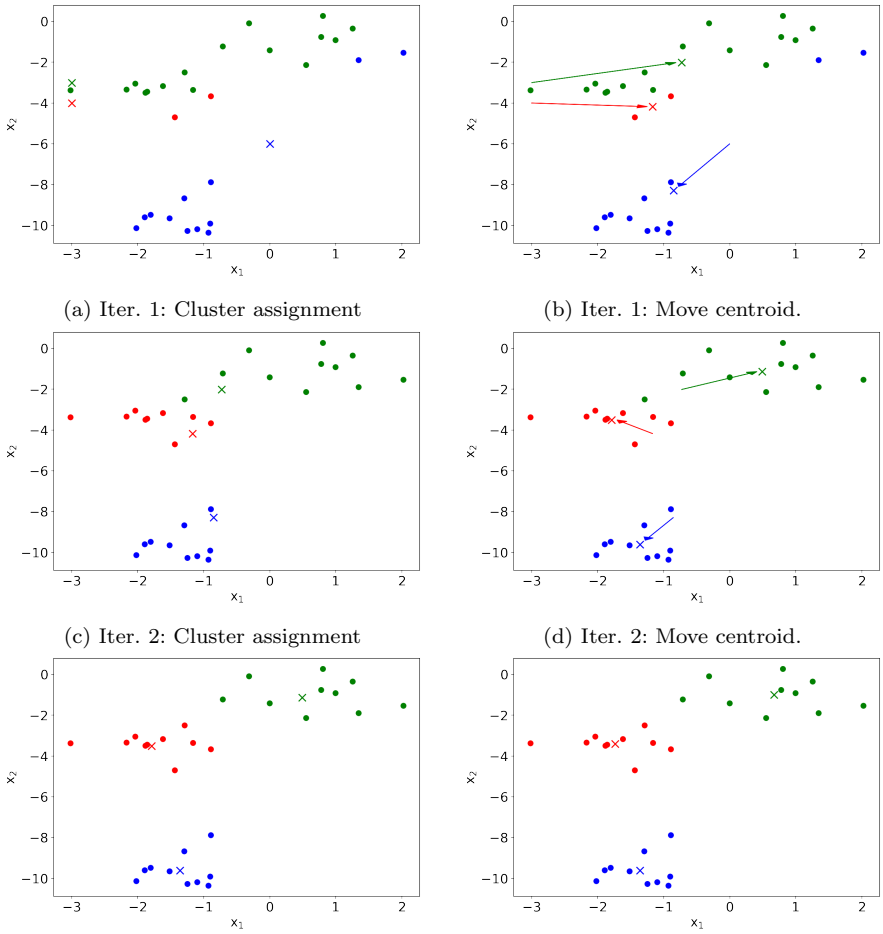
Fig. 13.1: A visualization of examples in two input variables x_1 and x_2

The k-means algorithm takes as input k, the number of clusters, and the input examples. It starts by randomly initializing what we call *cluster centroids*, which will eventually be the centroids of our final clusters. Figure 13.1b represents a visualization of these centroids alongside the input examples. Notice that each of these randomly initialized centroids has an associated color which we use to represent different clusters.

k-means then performs two steps iteratively: the first is what is called the *cluster assignment* step, wherein each input example is assigned to the cluster centroid that it is "closest" to (while we define 'close' formally later on, for now, think *close* in two and three dimensions), and the second is the *move centroid* step wherein each cluster centroid is moved to the centroid of the input examples that were assigned to it in step 1. Figure 13.2 illustrates three iterations over these steps. At the end of each iteration the centroid that an example is associated with (its color) can switch based on how the centroids themselves moved.

Notice how, on the third iteration, there is very little movement of the centroids (Figure 13.2f) and how any subsequent iterations will result in no change at all. When this occurs, k-means terminates and clusters associated with each cluster centroid are returned. Note that in practice, k-means is run for a fixed number of iterations (e.g. 15 – 20) as the clusters change only minimally after this point (also see Section 13.3.2).

It is important to remember that k-means is sensitive to both the initialization of the cluster centroids [4] and the value of k. What this implies is that the output clusters will change (sometimes dramatically) based on either of these. So far, we've gained a high level understanding of k-means. In the next section, we will formalize this intuition and, additionally, discuss the



(a) Iter. 1: Cluster assignment

(b) Iter. 1: Move centroid.

(c) Iter. 2: Cluster assignment

(d) Iter. 2: Move centroid.

(e) Iter. 3: Cluster assignment.

(f) Iter. 3: Move centroid.

Fig. 13.2: A visualization of k-means performing *cluster assignment* and *move centroid* over 3 iterations. Notice the negligible movement of centroids in iteration 3.

optimization objective for k-means, which provides a way of evaluating each of the different cluster assignments associated with different initializations and values of k. As a result, we can pick the "best" cluster assignment from amongst several different ones.

13.3 k-means: The internals

Now that we have an intuitive overview of the k-means algorithm, let us explore the algorithm more formally. To begin with, let us revisit the terminology we will be using: The n input examples are denoted as $\mathcal{T} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ...\mathbf{x}^{(n)}\}$ each consisting of m input variables (each person's details in the sweater example above consists of height and weight which are the input variables), and $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ...\boldsymbol{\mu}_k\}$ denote the k cluster centroids. $\{\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}\}$ denote the *sets* consisting of the indexes of input examples assigned to the corresponding cluster centroid. Specifically, $\mathcal{C}^{(1)}$ is a set that contains the indexes of input examples assigned to the cluster centroid $\boldsymbol{\mu}_1$ and so on – for example, $\mathcal{C}^{(1)}$ might be $\{1, 4\}$, which would imply that the input examples $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(4)}$ are assigned to the cluster centroid $\boldsymbol{\mu}_1$.

Algorithm 14 k-means algorithm

Parameters: Input examples $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(n)}$ and the number of clusters, k.
Output: k clusters ($\{\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}\}$).

1: Initialize (randomly) k cluster centroids $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ...\boldsymbol{\mu}_k\} \in \mathbb{R}^m$
2: **repeat**
3: **for** $i = 1$ to k **do**
4: $\mathcal{C}^{(i)}$:= indexes of input examples whose "closest" cluster centroid is $\boldsymbol{\mu}_i$
5: **end for**
6: **for** $i = 1$ to k **do**
7: $\boldsymbol{\mu}_i$:= center of input examples assigned to $\mathcal{C}^{(i)}$
8: **end for**
9: **until** No change to clusters ($\{\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}\}$) between iterations.

Consider the k-means clustering algorithm presented above (Algorithm 14). The algorithm takes as input the number of required clusters k and the input examples ($\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(n)}$), and returns the k clusters ($\{\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}\}$) consisting of indexes associated with input examples. Notice that the first *for* loop (lines 3 – 5) performs cluster assignment and second (lines 6 – 8) performs the move centroid step. It should be noted that some implementations of k-means use an array of length n to keep track of the index of the closest centroid to each input example instead of, as we have done, k sets to keep track of the indexes of input examples closest to each centroid. Note that these are equivalent.

The cluster assignment step requires us to find input examples that are closer to a given cluster centroid than any of the other cluster centroids. We do this by finding the *Euclidean distance* between each input example and all of the cluster centroids and picking the one that is closest. The combination of cluster assignment and move centroid allow k-means to generate clusters with minimal *within-cluster variance*. It is possible to use other distance measures (e.g. cosine, cityblock or squared Euclidean distance) in place of Euclidean distances and some implementations of k-means do provide these options.

However, these other measures do not result in clusters with minimal within-cluster variance and the reasoning for convergence of the algorithm that we provide in Section 13.3.3 relies on the use of Euclidean distance. As such, these other measures might result in the algorithm not converging. In the next section, we will discuss the optimization objective associated with k-means and use it to understand how the two steps of the algorithm translate to minimizing within-cluster variance.

13.3.1 Optimization Objective

Let's define $\mu(\mathbf{x}^{(i)})$ to represent the cluster centroid that input example $\mathbf{x}^{(i)}$ is assigned to. Recall that $\mathcal{C}^{(i)}$ is a set that contains the indexes of input examples assigned to the cluster centroid $\boldsymbol{\mu}_i$. Therefore, $\mu(\mathbf{x}^{(i)})$ can be written as $\mu(\mathbf{x}^{(i)}) = \boldsymbol{\mu}_j \iff i \in \mathcal{C}^{(j)}$, which expresses the fact that $\mu(\mathbf{x}^{(i)})$ will be equal to $\boldsymbol{\mu}_j$ (a given cluster centroid j) *if and only if* i is an element of $\mathcal{C}^{(j)}$. Notice that i being an element of $\mathcal{C}^{(j)}$ would in turn imply that the input example $\mathbf{x}^{(i)}$ is assigned to $\boldsymbol{\mu}_j$.

Notice that if a centroid is a *good* representation of a set of input examples in a cluster, then it would be "close" to each of them. As such, a good measure of this is the sum of the squared distance between the cluster centroid of a cluster and each of the constituent input examples associated with that cluster. This measure is called the *residual sum of squares* (RSS) and is expressed with respect to the cluster centroids ($\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ...\boldsymbol{\mu}_k\}$) and the cluster assignments ($\{\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}\}$) by Equation 13.1 below.

$$RSS(\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k) = \sum_{i=1}^{n} (\mathbf{x}^{(i)} - \mu(\mathbf{x}^{(i)}))^2 \tag{13.1}$$

Since the RSS represents how well our cluster assignment is doing, we can use it to build an *optimization objective* where we aim to minimize the RSS with respect to the centroids and the cluster assignments. We normalize the RSS by the number of input examples so as to make this value comparable across different sets of inputs. The resultant optimization objective is given by Equation 13.2.

$$\min_{\substack{\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)} \\ \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k}} \frac{1}{n} RSS(\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k) \tag{13.2}$$

In fact, the k-means algorithm presented in Algorithm 14 does exactly what is required by Equation 13.2, that is, it minimizes the RSS. This is because the first step – the cluster assignment step – minimizes the RSS with respect to the cluster assignment ($\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, ..., \mathcal{C}^{(k)}$), while holding the

cluster centroids constant, and the second step – the move centroid step – minimizes the RSS with respect to the centroids $(\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k)$, while holding the cluster assignment constant. Also, notice that, for a given cluster, the right hand side of Equation 13.1 is the within-cluster variance that k-means is implicitly minimizing.

13.3.2 Time Complexity

Let us now consider the time complexity of the k-means algorithm. The algorithm relies heavily on computing vector distances and adding vectors (to calculate the centroids), both of which have a time complexity of $\mathcal{O}(m)$, where m is the number of input variables. The cluster assignment step requires the calculation of distances between k centroids and n input examples, and so has an overall complexity of $\mathcal{O}(mkn)$. The move centroid step requires that we perform a total of n additions (across different clusters), each a vector of length m resulting in a complexity of $\mathcal{O}(nm)$. Therefore, the complexity of each iteration is $\mathcal{O}(mkn)$ and the overall complexity of the algorithm over i iterations is $\mathcal{O}(mkni)$.

Notice that the time complexity we've defined is reliant of the number of iterations i, which is relatively difficult to calculate. In the worst case, the complexity of k-means, when run until convergence, is superpolynomial [5]. However, in practice, cluster assignments change very little after a relatively small number of iterations and so k-means is usually run for a fixed number of iterations [6, Chapter 16].

13.3.3 Convergence

Does k-means always reach a "stable" state wherein further iterations do not lead to changes to either the cluster centroids or the cluster assignments? Is it possible that the cluster centroids continue to move around indefinitely, thus leading to a scenario wherein k-means does not terminate? It turns out that this is not the case and that k-means does always converge (as long as we use Euclidean distance). We can show that k-means converges by showing that the RSS monotonically decreases (or remains the same) as a result of the combination of cluster assignment and move centroid. Remember that, in practice, k-means is not run until convergence, but for a fixed number of iterations (see also Section 13.3.1).

Observe that if the cluster assignment step changes the centroid that an input example is assigned to, then the distance between the input example and that new centroid it is now assigned to (which contributes to the RSS) will necessarily decrease. If the new distance was going to be larger, the cluster

assignment step would not have performed this assignment. With regard to the move centroid step, intuitively, it should be clear that the point from which the sum of the distance to all points in a cluster is minimal is its center and so such a move will never increase the RSS. For a more formal analysis, see [6, Chapter 16].

Given the monotonic decrease of the RSS and the fact that there are only a finite number of possible clusterings, k-means is guaranteed to converge to a *local minimum* and terminate *as long as input examples that are equidistant from multiple cluster centroids are allocated to one in a consistent manner.* This is important as not doing so could result in k-means not terminating as such examples oscillate between centroids. Now that we have established that k-means does in fact terminate and that it does so at some local minimum, the next section discusses methods of attempting to find the solution that is globally optimal (or as close to it as is practical).

13.4 Random Initialization

As mentioned earlier, k-means is highly sensitive to random initialization of the cluster centroids – different initialization can result in very different clusters. Each clustering represents one possible local minimum of the RSS as described in Section 13.3.1 above. Finding a globally optimal solution requires running the algorithm multiple times using different random initialization and picking the one with the least RSS. Notice that this method is not guaranteed to provide a global optimum – it only increases the likelihood of finding one, and this likelihood continues to increase as the number of attempts increases. In practice, k-means, like most machine learning algorithms, is run 10 times, and we pick the clustering with the least RSS.

There are methods of randomly initializing the centroids that can sometimes speed up convergence. One such method is to randomly partition the input examples into k sets and to use the centroids of these sets as initial cluster centroids. Another method is to similarly partition *some* input examples into k sets each containing a relatively small number of examples (e.g. 10) and then use their centroids as the initial cluster centroids. For more initialization methods and a formal analysis of their characteristics, see [4, 7].

13.5 Optimal Number of Clusters

Given that the k-means algorithm requires the number of clusters as input, we must be able to (roughly) determine this number independently. Notice that we cannot rely solely on "minimizing RSS" as we did when selecting the best initialization. This is because the value of the RSS will reduce with the

increase in the value of k (up to some point). Therefore, the standard method of establishing the optimal number of clusters is by using the *Elbow Method*. This requires one to plot the RSS against a varying number of clusters as in Figure 13.3 and to pick a k at a point where the curve bends the most (the *elbow*), as highlighted in the plot. Notice that this allows us to pick a k beyond which the 'gains' obtained by adding a cluster are not worth the 'cost'. Importantly, for each k, the k-means algorithm must be run multiple times (about 10) using different initialization and the best clustering for that k is chosen to be the run with the minimal RSS.

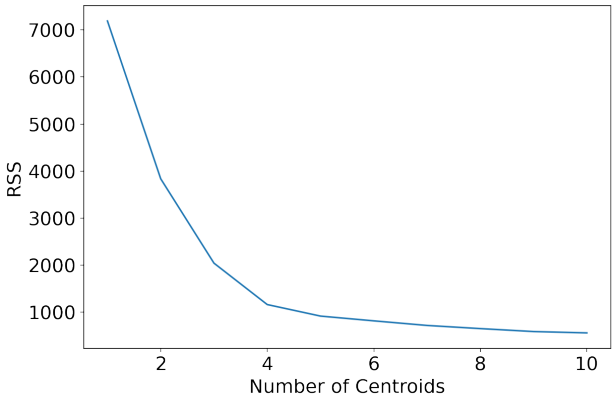


Fig. 13.3: Plot of how RSS varies with change in the number of cluster centroids. Notice how $k = 4$ provides an "Elbow".

The other way of establishing the number of clusters is based on what these clusters are going to be used for in a downstream task. For example, if we are clustering customers (by a combination of their heights and weights) so as to come up with a fixed number of sweater sizes (e.g. XS, S, M, L, XL), then we could use the number of sizes we intend to create (in this case 5) as our k.

13.5.1 Empty assignments

As the number of clusters increases, the cluster assignment step of k-means might result in one or more cluster centroids being assigned no input examples. As a result, the move centroid step will lead to errors because the centroid of zero input examples does not exist. There are a couple of ways to handle this: a) to ignore such centroids in all future steps of k-means, thus resulting in less than k clusters or, b) keeping such centroids in their prior positions and continuing with the algorithm. What you choose to do will

depend on the requirements of your problem, but not handling this case will result in your algorithm failing.

13.6 Summary and Discussion

In this chapter, we discussed the need for methods of discovering both standalone and connected clusters in input examples. Given this motivation, we studied the k-means algorithm, which is only one, albeit the most popular algorithm for finding clusters amongst input examples. We discussed the algorithm both intuitively and more formally with a particular emphasis on those aspects of the algorithm that are important to be aware of so as to effectively implement and use k-means. Remember that k-means requires the definition of k, which is its biggest handicap. Additionally, notice that k-means can only capture clusters that are *hyperspherical* (i.e. circle in two dimensions, sphere in three, ...) and so might not always be appropriate in certain situations. However, k-means is an intuitive algorithm, which, when run for a fixed number of iterations as is generally the practice, is also efficient.

13.7 Exercises

1. The k-means algorithm is run with k set to 2. During a particular iteration, the cluster centroids of the two clusters are A: $(15, 3)$ and B: $(12, 18)$. Which of the two centroids will the following input examples be assigned to and why: $(1, 2)$ and $(8, 9)$
2. This exercise is for those who are familiar with time complexity analysis of algorithms.
   Assume that a special chip, designed to optimize vector manipulation, is able to calculate the distance between vectors and to add vectors in constant time. What would the time complexity of the k-means algorithm be, assuming we always run it for a fixed number of iterations (say 15)?
3. Implement the k-means algorithm and additionally ensure that, for a given input, your implementation:
   a. Executes k-means for a varying number of clusters (e.g. between 2 and 10).
   b. Executes k-means multiple times (e.g. 10) using different initializations for each k and picks the best based on the value of RSS.
   c. Plots RSS vs k so as to identify the "Elbow" and so the most suitable k.
   d. Optionally plot the final clusters.

You might find the following functions helpful (although you are not required to use them):

- *euclidean_distances* from sklearn.metrics.pairwise
- *argmin* from numpy
- *mean* from numpy
- *seed* from random
- You might also like to maintain your data as numpy arrays so you can splice data more easily.
- Try to use matrix operations where possible, instead of loops over individual elements – this will keep your code compact and make it easier to implement.

Finally, your algorithm must cluster the input examples (X) generated by the following code snippet:

```
from sklearn.datasets import make_blobs
X, _ = make_blobs(n_samples=30, centers=3,
                  cluster_std=0.7, random_state=2)
```

13.7.1 Solutions

1. The Euclidean distance between two points (X_1, Y_1) and (X_2, Y_2) is given by: $\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$

   The distance between:

   a. A $(15, 3)$ and $(1, 2)$ is 14.04
   b. A $(15, 3)$ and $(8, 9)$ is 9.22
   c. B $(12, 18)$ and $(1, 2)$ is 19.42
   d. B $(12, 18)$ and $(8, 9)$ is 9.85

   Since both points are closer to A, they will both be assigned to the cluster centroid A.
2. The cluster assignment step requires the calculation of distances between k centroids and n input examples, and so has an overall complexity of $\mathcal{O}(kn)$. The move centroid step requires that we perform a total of n additions (across different clusters) resulting in a complexity of $\mathcal{O}(n)$. Therefore, the complexity of each iteration is $\mathcal{O}(kn)$. Since we run the algorithm for a fixed number of iterations, this is also the complexity of the algorithm itself.
3. Before you take a look at the solution, you are strongly encouraged to attempt the exercise yourself. The solution is available on Google Colaboratory at:

- https://colab.research.google.com/drive/1QNoSZpySciaBmik4eMKf3IaqTiRadWH0?usp=sharing

or

- https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/tree/main/Part%203%20-%20Learning%20Systems/Unsupervised%20Learning/k-Means/code.

The notebook also contains a step-by-step run through of the k-means algorithm which should be very helpful in understanding it. Feel free to make a copy of the notebook and play around with it.

References

1. Stuart Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.
2. Edward W. Forgy. Cluster analysis of multivariate data : efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.
3. Thomas Palys, L. K. Nakamura, and Frederick M. Cohan. Discovery and classification of ecological diversity in the bacterial world: The role of dna sequence data. *International Journal of Systematic and Evolutionary Microbiology*, 47(4):1145–1156, 1997.
4. M. Emre Celebi, Hassan A. Kingravi, and Patricio A. Vela. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems with Applications*, 40(1):200–210, 2013.
5. David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*, SCG '06, page 144–153, New York, NY, USA, 2006. Association for Computing Machinery.
6. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.
7. Paul S. Bradley and Usama M. Fayyad. Refining initial points for k-means clustering. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, page 91–99, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

Chapter 14
Hierarchical Clustering

Shuo Wang

Hierarchical clustering is a type of unsupervised methods that partitions data examples into clusters without asking for the number of clusters before training starts. Unlike K-means, hierarchical clustering instead requires specifying a dissimilarity measure between groups of examples, and produces a hierarchical representation (a dendrogram) of examples showing which two groups of examples are closer at each level of the hierarchy. Strategies for hierarchical clustering divide into two types: *agglomerative* (bottom-up) and *divisive* (top-down). Agglomerative clustering starts at the bottom treating each single example as a cluster, and recursively merges the closest pair of clusters into a single cluster. Divisive clustering acts in the opposite direction. It starts at the top with one big cluster and recursively splits into two new groups with the largest between-group dissimilarity. Divisive clustering is less popular than agglomerative clustering due to its complexity of looking for the best split at each round, but it can be made faster. Most agglomerative clustering methods have time complexity $\mathcal{O}(N^3)$, while the fastest divisive clustering takes only $\mathcal{O}(N)$ time. In addition, divisive clustering makes splitting decisions in view of all the data examples, while the agglomerative clustering makes myopic merge decisions [1].

14.1 Agglomerative Clustering

Agglomerative clustering begins with every data example representing a singleton cluster. For a dataset with N examples, the closest two clusters are merged into one cluster at each step, which repeats $N - 1$ steps in total. When the training process ends, there is one single cluster containing all the examples. To decide the closeness of two clusters, two parameters are required: 1) a distance measure indicating how far one example is from another

School of Computer Science, The University of Birmingham

in a dataset; 2) a dissimilarity measure indicating the distance between two clusters. The calculation of the dissimilarity measure for clusters needs the distance measure for examples. More details about these two parameters are given in Section 14.1.1. The pseudocode of agglomerative clustering is given below:

Algorithm 15 Agglomerative clustering

Parameters: dissimilarity measure d for clusters and distance measure d' for examples
Output: a hierarchical data representation.

1: Initialize N clusters $C_1, C_2, \ldots, C_i, \ldots, C_N$, where each cluster contains only one example.
2: **repeat**
3: Find two closest clusters C_j and C_k with smallest d to merge based on d'.
4: Create a new cluster $C_l \leftarrow C_j \cup C_k$.
5: Remove C_j and C_k from the cluster set.
6: Add C_l into the cluster set.
7: **until** no more clusters are available for merging.

14.1.1 Dissimilarity Measures Between Clusters

To find the closest (i.e. least dissimilar) clusters at each step, a dissimilarity measure between two clusters must be defined. Different measures can give quite different results. The 3 most commonly used measures are single linkage (SL), complete linkage (CL) and group average (GA). Others exist, such as the centroid method and the Ward's method [2].

For any pair of clusters C_j and C_k, the *single linkage* measure d_{SL}, also called nearest neighbour, is defined as the distance between the two closest examples of each cluster (see Fig. 14.1):

$$d_{SL}\left(C_j, C_k\right) = \min_{\mathbf{x}^{(t)} \in C_j, \mathbf{x}^{(t')} \in C_k} \{d_{t,t'}\}$$

where $d_{t,t'}$ can be any distance measure between two examples $\mathbf{x}^{(t)}$ and $\mathbf{x}^{(t')}$, such as Euclidean, Manhattan and Minkowski distances. Euclidean distance is the most commonly used one. It is defined as the length of the line between two points, and can be calculated using the Pythagorean theorem from the Cartesian coordinates of the points. The single linkage measure simply chooses the nearest examples and does not take in any account the internal cohesion of the clusters. If a small cluster is initially formed, it can lead to the progressive merging of adding one example at a time to this cluster, which is called a chain effect [3]. In fact, two clusters to be merged based on single

linkage do not have to be "compact". Compactness here means that all the examples within a group are similar to each other.

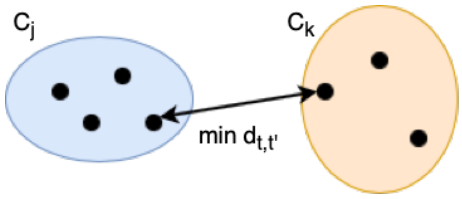


Fig. 14.1: Single linkage

The *complete linkage* measure d_{CL}, also called furthest neighbour technique, is defined as the distance between the two most distant examples of each cluster (see Fig. 14.2):

$$d_{CL}\left(C_j, C_k\right) = \max_{\mathbf{x}^{(t)} \in C_j, \mathbf{x}^{(t')} \in C_k} \{d_{t,t'}\}$$

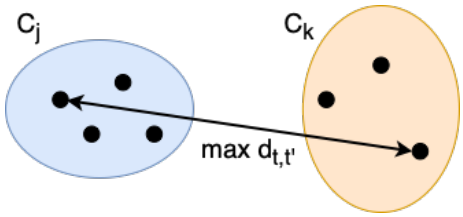


Fig. 14.2: Complete linkage

Complete linkage works the opposite to single linkage. Two clusters with the smallest "furthest example distance" will be merged together at each iteration. It means that the examples from these two clusters are more compact and relatively similar.

The *group average* measure (or average linkage), uses the average distance between all pairs of examples from the two clusters (see Fig. 14.3):

$$d_{GA}\left(C_j, C_k\right) = \frac{1}{N_j N_k} \sum_{\mathbf{x}^{(t)} \in C_j} \sum_{\mathbf{x}^{(t')} \in C_k} \{d_{t,t'}\}$$

where N_j and N_k are the number of examples in clusters C_j and C_k respectively. The group average is a measure in between single and complete linkage. It tends to produce more compact clusters than single linkage and further apart clusters than complete linkage.

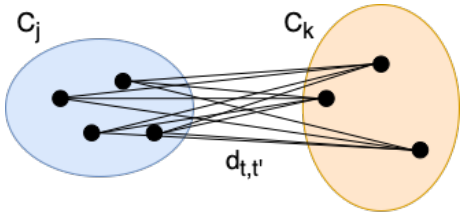


Fig. 14.3: Group average

14.2 Divisive Clustering

Divisive clustering begins with the whole dataset as a single cluster, and then recursively splits each existing cluster into two children clusters, which have the largest between-group dissimilarity, in a top-down fashion. The pseudocode of divisive clustering is given below for a dataset with N examples:

Algorithm 16 Divisive clustering

Parameters: cluster choosing scheme m, cluster splitting scheme s
Output: a hierarchical data representation.

1: All the examples $\mathbf{x}^{(i)}$ $(i = 1, 2, \ldots, N)$ form one cluster.
2: **repeat**
3: Pick one existing cluster C_i according to m.
4: Find two furthest sub-clusters C_j and C_k according to s.
5: Create two new clusters C_j and C_k, where $C_j \cup C_k = C_i$.
6: Remove C_i from the cluster set.
7: Add C_j and C_k into the cluster set.
8: **until** the desired number of clusters is obtained.

There are various ways to choose which cluster to be split at step 3, such as the cluster with most examples or the one with the least overall similarity among its examples. The existing research has shown that the differences between the choosing schemes were very small [4].

The splitting scheme can be any of the combinatorial methods, such as K-means with $K = 2$. This clustering procedure combining divisive clustering and K-means is called the bisecting K-means algorithm [4]. However, such schemes would depend on the algorithm initialization specified at each step. In addition, they do not necessarily hold the monotonicity property required for dendrogram representation [5]. A method called *dissimilarity analysis* proposed by Macnaughton-Smith et al. avoids these problems [6]. Instead of considering all divisions, it starts with a single cluster C_j containing all the data examples, then measures the average dissimilarity of each example $\mathbf{x}^{(t)}$ to all the other examples in C_j. The example that has the largest average dissimilarity is then moved to a second cluster C_k. The above is repeated,

i.e. moving examples from C_j to C_k, until the example in C_j with the largest average dissimilarity to the examples in C_j has a smaller average dissimilarity than its average dissimilarity to the examples in C_k. That is, there is no more examples in C_j that are on average closer to C_k. This process results in two children clusters. Each successive hierarchical level is produced by applying this process to each of the clusters at the previous level. Based on the dissimilarity analysis method, Kaufman and Rousseeuw proposed DIANA (Divisive ANAlysis). It splits a cluster by moving examples from a larger sub-cluster to a smaller one. The move occurs when the average dissimilarity of an example $\mathbf{x}^{(t)}$ in the larger sub-cluster to the remaining examples in that sub-cluster is larger than the average dissimilarity of $\mathbf{x}^{(t)}$ to the examples in the smaller sub-cluster [7]. A comparison between agglomerative and divisive clustering can be found here [8].

14.3 Interpreting a Dendrogram

The binary tree produced by hierarchical clustering is called a dendrogram. It is a graphical display of how examples are grouped at each hierarchy level. A dendrogram is highly interpretable, which is one of the main reasons for the popularity of hierarchical clustering.

Let's begin with a simple example. It is a simulated data set with 5 two-dimensional examples, as shown in Fig. 14.4.

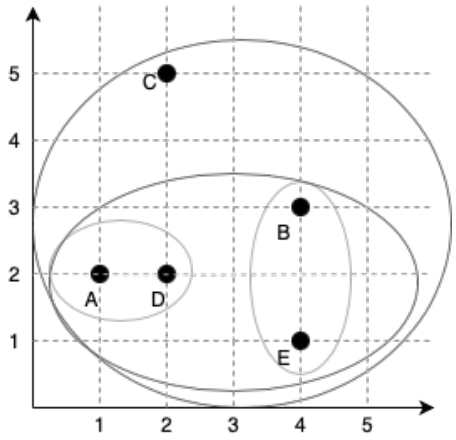


Fig. 14.4: Two-dimensional data examples

If we apply agglomerative clustering with single linkage, the clustering happens in the following order:

1. Examples A and D are grouped with distance 1.
2. Examples B and E are grouped with distance 2.
3. Cluster (A, D) and Cluster (B, E) are grouped with distance $\sqrt{5}$.
4. Example C is merged to Cluster (A, D, B, E) with $2\sqrt{2}$.

The resulting dendrogram is shown in Fig. 14.5. The horizontal axis indicates the examples in the dataset. The links between the examples represent which examples are merged into one cluster. The vertical axis represents the height of two merging clusters. *Height* is referred to as the distance/dissimilarity between the clusters. In this case, the distance between A and D is 1; thus, their connecting line is at 1 along the vertical axis. Similarly, B and E are connected at the height of 2. This height is monotonically increasing with the level of the dendrogram. In other words, the examples that merge at the very bottom of the tree are quite similar, whereas the examples that merge close to the top of the tree tend to be very different.

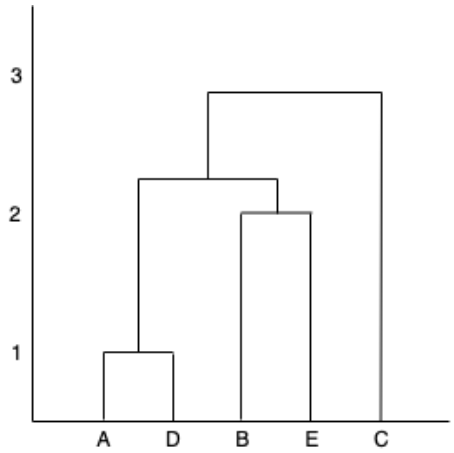


Fig. 14.5: Resulting dendrogram

Dendrograms should be read with caution. First, different hierarchical methods and dissimilarity measures can lead to quite different dendrograms. Second, when reading a dendrogram, we cannot conclude about the similarity of two examples based on how far they are along the horizontal axis [9]. For example, example C in Fig. 14.5 locates next to example E, but it does not mean C closer to E. All the other examples are in fact closer to C. Third, hierarchical clustering does not tell you the "right" number of clusters. Instead, it creates a hierarchical representation of data, even if data has absolutely no hierarchical meanings. By cutting off a dendrogram at various heights, different numbers of clusters are obtained. In practice, people often look at the dendrogram and select by eye a sensible number of clusters, based on

the heights of merging clusters and the desired number of clusters [9]. Alternatively, other approaches have to be used, such as Bayesian hierarchical clustering [10].

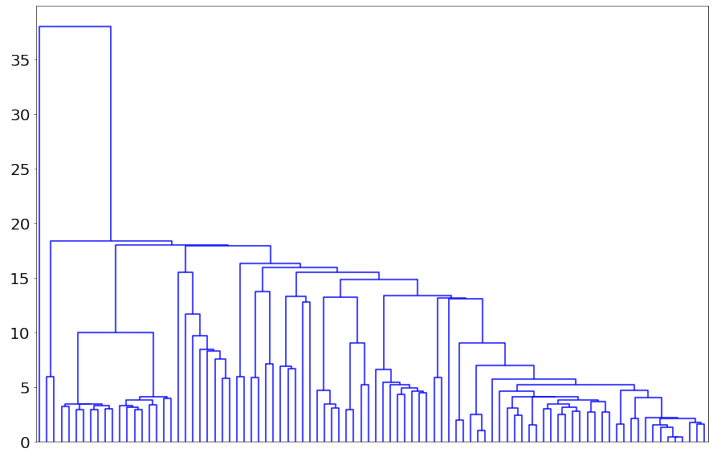
14.4 Example: Yeast Gene Data

Agglomerative clustering is popular in bioinformatics, because it provides visualization of clusters as dendrogram and gives explainability to the task. Here, we use a yeast gene expression dataset [11] and aim to cluster similar yeast samples together. It contains 92 samples described by over 6000 genes.

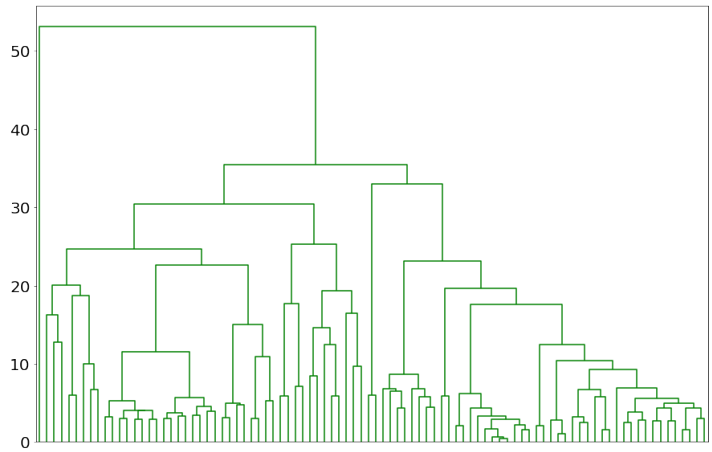
Fig. 14.6 shows the dendrograms resulting from agglomerative clustering with single linkage, complete linkage and group average respectively. Depending on which dissimilarity measure we use, the results are quite different. The generated binary trees are more balanced in the complete linkage and group average cases than in the single linkage case. In practice, the choice of dissimilarity measure should take into consideration the type of data and the learning task at hand [9]. The vertical axis in each dendrogram shows the height of two clusters when they are merged.

14.5 Summary and Discussion

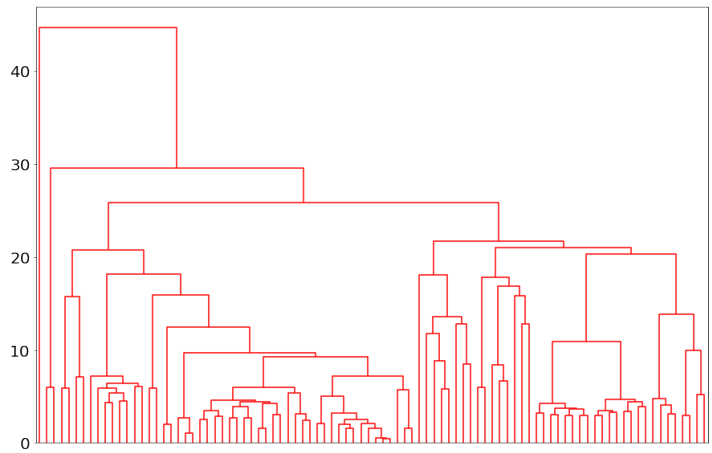
The core concepts of hierarchical clustering have been introduced. There are two types of hierarchical clustering – agglomerative and divisive, differing in the direction of forming clusters. Agglomerative clustering starts at the bottom treating each single example in the dataset as a cluster, and recursively merges the closest pair of clusters into a single cluster. Divisive clustering starts at the top with one big cluster and recursively splits into two new groups with the largest between-group dissimilarity. Agglomerative clustering is more widely used due to its easier implementation. Same as the other clustering algorithms, hierarchical clustering requires a distance measure to decide how far an example is from another. In addition, agglomerative clustering needs to choose a dissimilarity measure to decide the distance between two clusters. Three commonly used dissimilarity measures have been discussed and compared through a yeast gene data example, which are single linkage, complete linkage and group average. The resulting clusters from hierarchical clustering are represented by a dendrogram. It is highly interpretable by showing how examples are grouped at each hierarchy level and the height of two clusters when being merged. This is also the main reason for the popularity of hierarchical clustering in the field of bioinformatics. With good explainability, it will be interesting to see how hierarchical clustering contributes to many other fields, for the readers to explore. Some inspirations



(a) single linkage



(b) complete linkage



(c) group average

Fig. 14.6: Agglomerative clustering of yeast gene expressions data with (a) single linkage, (b) complete linkage and (c) group average. Figures are generated by Python.

may be found here [12]. If you are familiar with Python and would like to implement different versions of hierarchical clustering and use them on some datasets, here are some useful resources [13] [14].

Exercise

1. Is agglomerative hierarchical clustering a deterministic or non-deterministic clustering algorithm? A deterministic algorithm always gives the same results regardless of the starting states of the algorithm.

2. You are using agglomerative clustering to cluster an 1-dimensional data set. At some point, you obtain 2 clusters: cluster 1 with numbers [7, 11] and cluster 2 with numbers [12, 16, 20]. What is the distance between these 2 clusters using single linkage? And what about using complete linkage and group average?

3. Use single and complete link agglomerative clustering to group the data described by the following distance matrix. Draw the dendrograms.

	A	B	C	D
A	0	5	2	3
B		0	1	6
C			0	4
D				0

4. As hierarchical clustering does not explicitly tell you the number of clusters of data in results, how would you decide the most appropriate number of clusters?

Exercise Answers

1. Agglomerative clustering is a deterministic algorithm, as the result does not change with the starting states of the algorithm.

2. The distance will be 1 when using the single linkage, because 11 from cluster 1 and 12 from cluster 2 are the nearest examples from the 2 clusters, and the distance between these 2 examples will be the distance of these 2 clusters. Similarly, when using complete linkage, the distance between the 2 clusters will be 13, which is the distance between examples 7 and 20. When using group average, we calculate the average distance between all pairs of examples from the two clusters, so the cluster distance will be

7.

3. When using the single linkage, the dendrogram is:

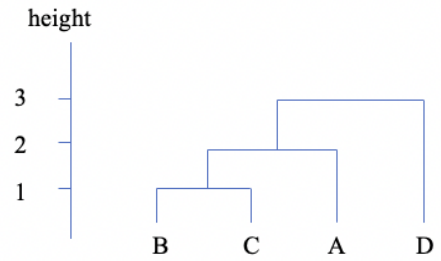


Fig. 14.7: Resulting dendrogram from the single linkage

When using the complete linkage, the dendrogram is:

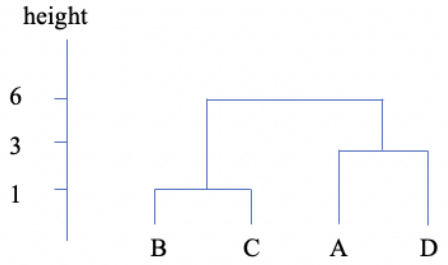


Fig. 14.8: Resulting dendrogram from the complete linkage

4. The best choice is always the pre-knowledge, if you know the number of clusters in advance. If not, a common way specific to hierarchical clustering is to observe the dendrogram you obtain. See and compare the height of every two merging clusters at each hierarchy level. Find two adjacent heights that give the largest distance increase, and draw a horizontal line between these two hierarchy levels. The best choice can be the number of vertical lines intersect by the horizontal line. Other general methods include Silhouette coefficient [15], Elbow method [16], Calinski-Harabasz Index [16], Davies-Bouldin Index [17], etc.

References

1. Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA, 2012.
2. Joe H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.
3. Federico Marini and José Manual Amigo. Unsupervised exploration of hyperspectral and multispectral images. *Data Handling in Science and Technology*, 32:93–114, 2020.
4. Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *Proceeding of the KDD Workshop on Text Mining*, pages 1–20, 2000.
5. Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, New York, 2009.
6. P. Macnaughton-Smith, W. T. Williams, M. B. Dale, and L. G. Mockett. Dissimilarity analysis: a new technique of hierarchical sub-division. *Nature 202*, pages 1034–1035, 1964.
7. Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, 1990.
8. Satyam Kumar. Hierarchical clustering: Agglomerative and divisive – explained. `https://towardsdatascience.com/hierarchical-clustering-agglomerative-and-divisive-explained-342e6b20d710`, 2020. [Online; accessed April 29, 2022].
9. Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, New York, 2013.
10. Katherine A. Heller and Zoubin Ghahramani. Bayesian hierarchical clustering. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pages 297–304, 2005.
11. Kaggle. Transcriptomics in yeast. `https://www.kaggle.com/costalaether/yeast-transcriptomics`, 2016. [Online; accessed August 8, 2021].
12. Claire Whittaker. Seven innovative uses of clustering algorithms in the real world. `https://datafloq.com/read/7-innovative-uses-of-clustering-algorithms`, 2019. [Online; accessed April 27, 2022].
13. Scikit learn 2.3.6. Hierarchical clustering. `https://scikit-learn.org/stable/modules/clustering.html#hierarchical-clustering`, 2022. [Online; accessed April 27, 2022].
14. Jason Brownlee. 10 clustering algorithms with python. `https://machinelearningmastery.com/clustering-algorithms-with-python`, 2020. [Online; accessed April 28, 2022].
15. Wikipedia. Silhouette (clustering). `https://en.wikipedia.org/wiki/Silhouette_(clustering)`, 2022. [Online; accessed June 14, 2022].
16. Indraneel Dutta Baruah. Cheat sheet for implementing 7 methods for selecting the optimal number of clusters in python. `https://towardsdatascience.com/cheat-sheet-to-implementing-7-methods-for-selecting-optimal-number-of-clusters-in-python-898241e1d6ad`, 2020. [Online; accessed June 14, 2022].
17. Wikipedia. Davies–bouldin index. `https://en.wikipedia.org/wiki/DaviesBouldin_index`, 2022. [Online; accessed June 14, 2022].

Chapter 15
DBScan

Lina Yao

The Density Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm was first proposed in 1996 by [1]. It has been proven to have promising performance in various applications, and it received the SIGKDD test-of-time award in 2014. DBSCAN is a unsupervised machine learning and density-based clustering method. It aims for identifying high-density regions in data space and treating them as clusters, separated by regions of low point density. Let's starting by an illustrative example for better understanding the density. Taking a look at Figure 15.1, there 3 sets of sample points (the figures from left to right). Intuitively, we can identify these clusters of points and unclustered outliers straightforwardly, no matter how those sample points are distributed or in what shapes. By instinct, we naturally observe and recognize the density differences within those areas and consider each cluster of points have higher denseness. While the rest points of considerably lower densities are regarded as noise and belong to no cluster [1]. Based on a set of points in this example, DBSCAN groups together points that are close to each other based on an arbitrary distance measurement in different application domains (e.g., Euclidean distance) and a minimum number of points that is a threshold to distinguish the outliers in the lower density regions.

15.1 The Algorithm

Suppose we have a set of data points

$$\mathcal{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots, \mathbf{x}^{(N)}\}$$

where $\mathbf{x}^{(i)} \in \mathcal{X}$ are drawn i.i.d. (independently and identically distributed). The core idea of DBSCAN is to estimate the density for each data point $x^{(i)}$ as

School of Computer Science and Engineering, University of New South Wales, Australia

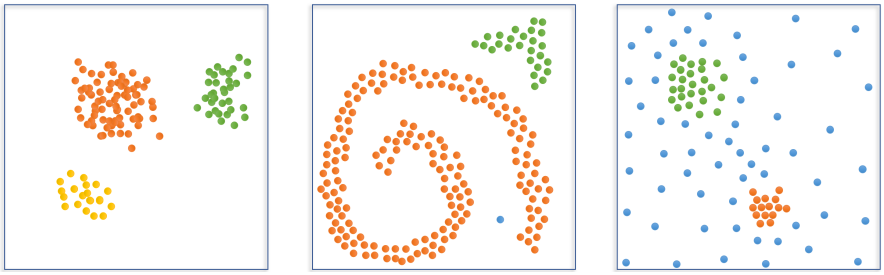


Fig. 15.1: Samples for density illustration. Different colors represent different clusters of points

a minimal number of $MinPts$ of points that lie inside a radius ϵ around $x^{(i)}$. Points with distances to each other no greater than a pre-defined threshold ϵ can be deemed connected and within the same cluster. As such, DBSCAN has two key parameters.

- ϵ is a positive real value. It is defined using a distance function and specifies the neighborhoods. Two points are considered to be neighbors if the distance between them are less than or equal to ϵ, i.e., $dist(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}) \leq \epsilon$. The data points within a radius of ϵ from a given point $\mathbf{x}^{(i)}$, form its ϵ-Neighborhood denoted as $N_\epsilon(\mathbf{x}^{(i)}) : \{\mathbf{x}^{(i)} | dist(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \leq \epsilon\}$.
- $MinPts$ is a small positive constant integer, that specifies the minimum number of data points to define a cluster.

If the neighborhood $N_\epsilon(\mathbf{x}^{(i)})$ contains a number of points equal to or more than $MinPts$, it's called high density, otherwise it is low density. For example, if we set $MinPts = 4$ in Figure 15.2, ϵ-Neighborhood of point A is a high density area since it has 5 points including A itself in its surrounding neighbourhood, which is more than $MinPts = 4$, while ϵ-Neighborhood of point C is low density area since its only has 2 points including C itself. Up to this point, we have defined two key parameters, $MinPts$ indicates the density and ϵ indicates the radius of ϵ-Neighbourhood, and gained the basic understandings of measuring the density. The points in a given dataset are then classified into the following categories.

- *Core point*. A core point has a ϵ-neighbourhood with more than $MinPts$ data points (including the given point itself);
- *Border point*. A border point locates in the neighborhood of certain core point, but itself has a number of points less than $MinPts$ within its ϵ-Neighbourhood;
- *Outlier*. A point that is neither a core point nor a border point.

As recalled, the intuition of DBSCAN is to find the regions in a data space which have higher density, separated by regions of lower density. In other

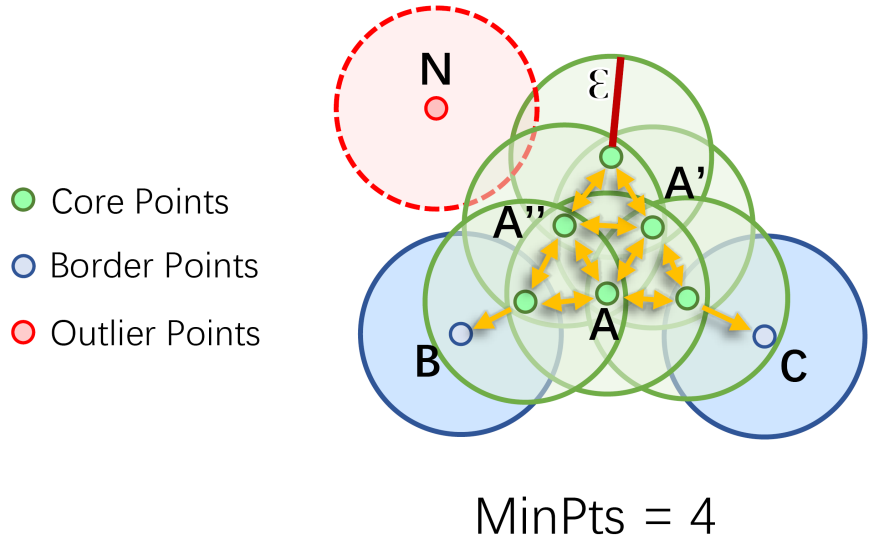


Fig. 15.2: In this example, the minPts parameter is 4, and the ϵ radius is indicated by the circles. Red point is a outlier point, which is not density reachable. Green points are a core point, and pale blue points are border points. Double arrows indicate direct density reachability, which is symmetric. Arrows connecting the border points B and C indicate density connected, while both are density reachable from the A. The density connectivity is asymetric. N is not connected indicating not density reachable, and thus considered to be a outlier point/noise. Figure adapted from Schubert et al. [2]

words, to group each data point in a cluster which contains at least $MinPts$ number of points in the ϵ-neighbourhood of this point. If this point is a core point, and then it forms a cluster along with all data points including core points and border points that are reachable from it. The ϵ-neighbourhood of a certain border point usually has fewer points than the ϵ-neighbourhood of points located within its cluster. A very small $MinPts$ is necessary to be set to group all the points belonging to a same cluster. However, this could also cause a problem in distinguishing points from the data noise/outlier. To tackle this issue, DBSCAN requires that every point $\mathbf{x}^{(i)}$ is in a cluster where there is a point $\mathbf{x}^{(j)}$ so that $\mathbf{x}^{(i)}$ is within ϵ-neighbourhood of $\mathbf{x}^{(j)}$ and its neighbourhood has more than $MinPts$ number of points. To achieve this, the different levels of density reachability are defined [1]

- *Directly density-reachable.* Point $\mathbf{x}^{(i)}$ is directly density-reachable from a point $\mathbf{x}^{(j)}$ wrt. ϵ and $MinPts$ if $\mathbf{x}^{(j)} \in N_\epsilon(\mathbf{x}^{(i)}) : \{\mathbf{x}^{(i)} | dist(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \leq \epsilon\}$ and $|N_\epsilon(\mathbf{x}^{(j)})| \geq MinPts$. Obviously, directly density-reachable is symmetric for core point pairs. For instance, core points A, A' and A" are

directly reachable in Figure 15.2. Such symmetric relationship cannot be established for a core point and a border point.

- *Density-reachable.* A point $\mathbf{x}^{(i)}$ is density reachable from a point $\mathbf{x}^{(j)}$ wrt. ϵ and $MinPts$ if there is a chain of points $\mathbf{x}^{(i)}, \mathbf{x}^{(i+1)}, ..., \mathbf{x}^{(j)}$ where from i to j, all points satisfy that $\mathbf{x}^{(i+1)}$ is directly density-reachable from $\mathbf{x}^{(i)}$. For instance, point B is density-reachable from A, while A is not density-reachable from B in Figure 15.2. Density-reachability is a canonical extension of direct density-reachability. This relation is transitive, yet not symmetric. Two border points within the same cluster may be not density reachable from each other because the core point condition might not hold for both of them. However, there must be a core point in a cluster from which both border points of this cluster are density-reachable. Therefore, the following notion of density-connectivity is introduced to cover this relation of border points.
- *Density-connectivity.* A point $\mathbf{x}^{(i)}$ is density connected to a point $\mathbf{x}^{(j)}$ wrt. ϵ and $MinPts$ if there is a point $\mathbf{x}^{(k)}$ such that both, $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ are density-reachable from $\mathbf{x}^{(k)}$ wrt. ϵ and $MinPts$. Density-connectivity is a symmetric relation. For instance, border points B and C become density-connected in Figure 15.2 via core points A, A' or other green core points.

Algorithm 17 DBSCAN

Parameters:
$\mathcal{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots, \mathbf{x}^{(N)}\} \in \mathbb{R}^d$
ϵ: radius distance of neighbour
$MinPts$: minimal size of a cluster

$C \leftarrow 0$
for all unvisited point $\mathbf{x}$ in $\mathcal{X}$ **do**
  mark $\mathbf{x}$ as $Visited$
  $N_\epsilon(\mathbf{x}) \leftarrow$ regionQuery($\mathbf{x}$, ϵ) {finding initial neighbourpoints of $\mathbf{x}$}
  if $|N_\epsilon(\mathbf{x})|$ <MinPts {size of neighbourpoints of $\mathbf{x}$} **then**
    mark $\mathbf{x}$ as $NOISE$
  else
    $C \leftarrow C + 1$ {start a new cluster}
    expandCluster($\mathbf{x}$, NeighborPoints, C, ϵ, $MinPts$) {See Algorithm 18.}
  end if
end for

Now, we have a formal notion of density. A cluster C is defined to be a set of density-connected points wrt ϵ and $MinPts$, which is non-empty and satisfying the following two properties (i) Maximality: for any data point $\mathbf{x}^{(i)} \in C$ and $\mathbf{x}^{(j)}$ that is density-reachable from it wrt ϵ and $MinPts$, then $\mathbf{x}^{(j)} \in C$; (ii) any of two points in this cluster are density-connected. A toy example in Figure 15.2 explains the how it progresses. All neighbors within the ϵ-neighbourhood of a core point A are considered to be part of

Algorithm 18 expandCluster

Parameters:
$\mathbf{x}$: current points
$\mathbf{x}' \in N_\epsilon(\mathbf{x})$: neighbour points of current points $\mathbf{x}$
C: current cluster
ϵ
$MinPts$: minimal size of a cluster

```
add x to cluster C
for all point x' in N_ε(x) do
    if x' is not visited then
        mark x' as visited
        N_ε(x') ← regionQuery(x' , ε) {See Algorithm 19.}
    end if
    if |N_ε(x')| ≥ MinPts then
        N_ε(x) = N_ε(x) ∪ N_ε(x')
    end if
end for
```

Algorithm 19 regionQuery

Parameters:
$\mathbf{x}$: current point
ϵ: represent min distance of neighbour

```
Return all points within ε distance of x
```

the same cluster due to they are direct density reachable to A. If any of these points in its ϵ-neighbourhood is again a core point like A' and A'', and their neighborhoods are transitively included (called density reachable) into the same cluster. The border points (like point B) in the same set are density connected. As the progress continues, point B will be maximally density connected with another border point C via a chain. Relatively, points which are not density reachable from any core point are considered noise and do not belong to any cluster like the red point N. The cluster is complete as it becomes surrounded by border points and there are no more points within ϵ-neighbourhoods. A new random point will be selected and repeat the same process to identify the next cluster.

The pseudocode of DBSCAN is shown in Algorithm 17. It is noted that the RegionQuery$(\cdot)$ function is called for each point, which most significantly contributes to the runtime complexity of DBSCAN. The overall runtime complexity is $\mathcal{O}(n \cdot RegionQuery(\cdot))$. If this function is implemented in the most naive way, the runtime complexity of DBSCAN would be $\mathcal{O}(n^2)$, which is the worst case. Many advanced techniques are proposed to speed up the region query, such as R-tree, KD-tree etc [2]. It is commonly recognized that the average runtime complexity of DBSCAN is $\mathcal{O}(n \cdot \log n)$.

15.2 An Example of DBSCAN

Clustering has been widely used in a variety of real-world applications like market analysis, social network analysis, segmentation and object detection. In this chapter, we demonstrate how DBSCAN can be used for object segmentation task on a RGB+Depth scene dataset, SUN RGB-D, which is a RGB-D scene understanding benchmark suite. Object segmentation is concerned with partitioning an image into different objects. It can be formulated as an clustering problem whose main objective is to separate pixels into homogeneous clusters (i.e., objects) that hold on maximum similarity within the clusters while reaching the minimum dissimilarity cross the clusters. Each cluster then corresponds to a different object.

The SUN RGB-D dataset contains 10,000 RGB-D images. It is densely annotated and includes 146,617 2D polygons and 58,657 3D bounding boxes with accurate object orientations. A 3D room layout where the images are captured and scenario categories for the images are provided is also available. The detailed description and download information can be found from here [1]. We use the scikit-learn [2] to implement. Figure 15.3 shows the results. We can clearly observe that DBSCAN can accurately segment the different shapes of furniture.

15.3 Determine ϵ and $MinPts$

DBSCAN has 2 key parameters ϵ and $MinPts$, which can be critical to its performance. It requires joint tuning these two parameters to find the optimal combination to ensure the clustering performance. In general, there is no common practice to determine $MinPts$. It depends on domain knowledge and understanding of the given datasets. A low $MinPts$ means it will build more clusters from noise. In many cases, the domain knowledge is generally unknown especially considering that data is usually normalized before further processing. Ester et. al [1] propose to use the sorted k-distance graph to assist with deciding $MinPts$. The k-dist graph plots the average distance between each point and its k-nearest neighbours in ascending order, where $k = MinPts$. In their case, experiments indicate that with a $k > 4$, k-dist graphs introduce substantially more computational costs while contributing no significantly better results than the 4-dist graph. Therefore, $MinPts = 4$ was recommended for their 2-dimensional data. It is noted that this setting may not be applicable to other datasets.

[1] `http://rgbd.cs.princeton.edu/`

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html`

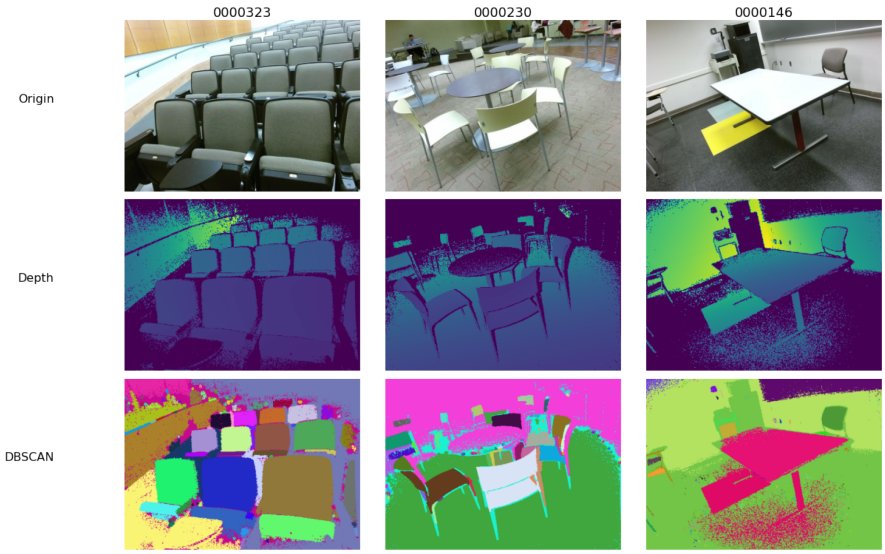


Fig. 15.3: Object detection using DBSCAN. The first row shows the original images, the second row shows the corresponding depth images, and the last row shows the detection results using DBSCAN, where different colors represent different clusters.

More heuristics for $MinPts$ are proposed, for example, $MinPts$ can also be set as $MinPts = \frac{1}{n}\sum_{i=1}^{n} |N_\epsilon(\mathbf{x}^{(i)})|$, where $|N_\epsilon(\mathbf{x}^{(i)})|$ is the number of points in ϵ-neighbourhood of given data point $\mathbf{x}^{(i)}$ and n is the total number of data points in the dataset [3]. Another way for choosing $MinPts$ is to derive it from the number of dimensions d of data set by taking $MinPts = 2 * d - 1$ [4].

In general, ϵ should be chosen as small as possible. However, if ϵ is too small, many points may be considered as outliers because a considerable amount of points may be discarded as outliers due to the very small neighborhoods, which results in many points not becoming core points or border points. While, a large value for ϵ may produce rather huge clusters with too many points inside. In the original DBSCAN paper [1], an interactive approach is offered to detemine the optimal ϵ. The basic process is to first calculate the average of the distances of every point to its k nearest neighbors, where $k = MinPts$, and the computed k-distances are plotted in a descending order. The knee point formed by the plot indicates a threshold where a sharp change occurs along this k-distance curve. The value of knee point is treated as the optimal ϵ. The Figure 15.4 shows an example of determining the ϵ using k-distance plot.

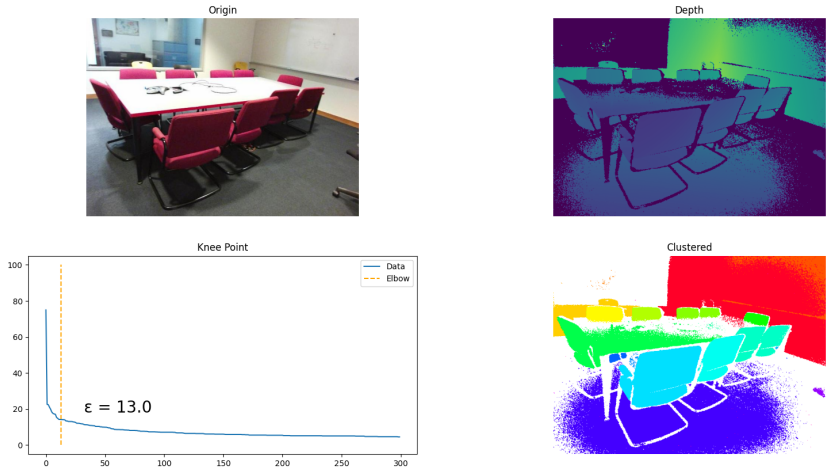


Fig. 15.4: An illustrative example is finding optimal ϵ given $k = 4$; the knee point corresponds to $\epsilon = 13.0$. The clustering result shows the good performance with these two parameters.

15.4 Summary and Discussion

DBSCAN can work on discovery of clusters with arbitrary shape of datasets and the number of clusters that does not need to be predefined. It has good noise resistance and can handle noisy dataset well. The desirable range of data noise ratio for DBSCAN to tolerate well is between 1% to 30% [2].

However, DBSCAN may not be scalable to large datasets or high-dimensional dataset as it requires more memory and computing power. Recently, there is some debate on the runtime complexity of orginal DBSCAN in the research community. The running time of DBSCAN depends on how many times the function $RangeQuery(\cdot)$ is called. The original DBSCAN paper claims the average runtime complexity is $\mathcal{O}(n \log n)$. Gan et al. [5] argued that the algorithm actually requires $\mathcal{O}(n^2)$. However, their conclusion was questioned by other researchers. For example, Schubert et al. [2] pointed out some inaccuracies in the way DBSCAN was represented by Gan et al. [5] instead of the algorithm itself, especially on the assumption about the performance of spatial index structures used to support the RegionQuery() method such as R-trees. Therefore, $\mathcal{O}(n^2)$ is the worst case. Interested readers can dig more details by referrring to these research papers.

Furthermore, for efficiency reasons, DBSCAN does not perform density estimation between all points. All neighbors within the ϵ radius of a core point are considered to be part of the same cluster as per defined direct density reachable. If any of these neighbors is again a core point, transitively

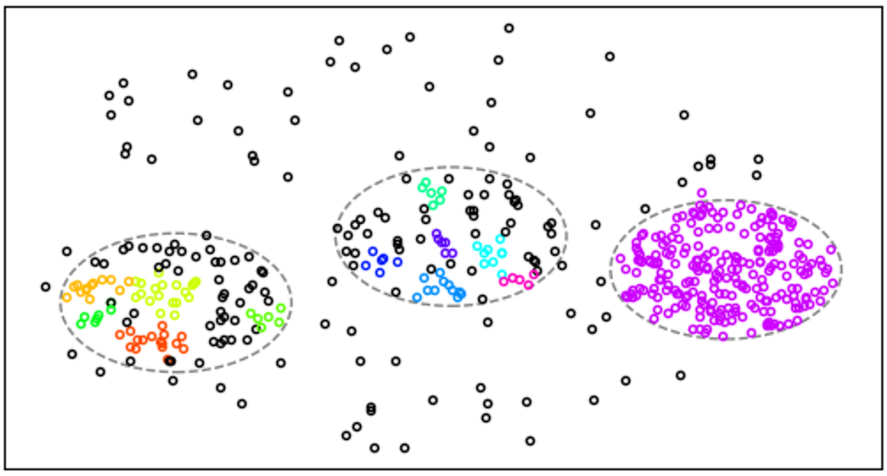


Fig. 15.5: An illustrative example is DBSCAN dealing with varying density dataset. The colorful dots indicate the clusters identified by DBSCAN, while the dash lines indicate the groundtruth clusters.

their density reachable neighbors are included. Non-core points in this set, or, border points, are included as well as per density connectivity. Points which are not density reachable from any core point are considered noise and do not belong to any cluster. However, if the density of the sample set is not uniform, DBSCAN performs poorly as illustrated in Figure 15.5.

15.5 Excercises

1. Compared with K-means, give the advantages and disadvantages of DBSCAN.
2. Run the code provided by varying $MinPts$ and ϵ of the example given in this chapter to observe the impact of performance.
3. Apply k-distance plot to find optimal ϵ. You can use the SUN RGB-D dataset listed in the example given in this chapter.

15.6 Answers

1. The advantages of DBSCAN are (1) it doesn't need hyperparameter like predefining the number of clusters k as k-means; (2) it can work well to form the artitary shapes of clusters; (3) it is robust to outliers. The disadvantages of DBSCAN are (1) it may work properly on the varying

density datasets; (2) it requires the prior knowledge for distance function, minPts and ϵ

2. Keeping one parameter fixed and varying another one to observe how changing parameters can impact on the performance. As per extensive research studies, with increased $MinPts$, the performance might be observed improved espeically over large, high dimensional datasets or datasets containing lots of duplicates. The ϵ is a bit tricky. Its value may heavily rely on domain knowledge (e.g., distance function in different application domains in terms of clustering location clusters measured in km, or clustering in medical imaging measure in cm). You may observe significantly different situations over different datasets.
3. The code of solution can be found at
`https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1/blob/main/Part%203%20-%20Learning%20Systems/Unsupervised%20Learning/DBScan/code/find_eps.py`
or
`https://gist.github.com/LinaYao/00f3b2cf65600f7b03df020c81cc5de9`.

References

1. Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, pages 226–231, 1996.
2. Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.
3. Kedar Sawant. Adaptive methods for determining dbscan parameters. *International Journal of Innovative Science, Engineering & Technology*, 1(4):329–334, 2014.
4. Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data mining and knowledge discovery*, 2(2):169–194, 1998.
5. Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 519–530, 2015.

Chapter 16
Expectation Maximization

Bruno Almeida Pimentel

Expectation Maximization (EM) is one of the most frequently used algorithms for clustering in Machine Learning [1, 2]. Its popularity comes from the underlying simplicity of the algorithm. EM can be used in a number of applications, such as Data Mining, Clustering, Natural Language Processing, Signal Processing, Medical Image Reconstruction amongst others [3, 4, 5, 6, 7]. This chapter is aimed at providing the reader with knowledge for understanding and using the EM method. Furthermore, readers can exercise their knowledge by implementing the method based on the provided definitions.

16.1 Intuition Behind the EM clustering algorithm

This section is aimed at providing the reader with the basic underlying idea of the EM method operation. Two practical experiments are discussed in order to expose the main steps of the method.

16.1.1 A coin-flipping experiment

With the aim of explaining the mathematical intuition as well as the Expectation Maximization algorithm operation, this section focuses on a simple coin-flipping experiment. So, suppose that we want to estimate the bias of two coins. Also, consider that these coins can be fair or not (i.e., they can be more heavily weighted on their heads face, for example). This experiment is based on steps where each one consists of two main actions:

Instituto de Computação - Universidade Federal de Alagoas

- Randomly choose a coin;
- Flip the chosen coin 10 times.

After carrying out these steps 6 times, we may achieve the result presented in the table below (where H means heads and T means tails):

Steps	**Coin**	**Trial**	**# coin A heads**	**# coin B heads**
1	A	HTTTHHTHTH	5	0
2	B	HHHHTHHHHH	0	9
3	B	HTHHHHHTHH	0	8
4	B	HTHTTTHHTT	0	4
5	A	THHHTHHHTH	7	0
6	B	HTTHTHHHHH	0	7

To independently estimate each coin's bias, we must consider the total of heads and the total of flips. For coin A, the total of flips is 20 (2 flips trials of 10 flips each one). Thus, the bias of coin A would be 12/20. Similarly, for coin B the bias is 28/40. Therefore, when we know which coin was flipped, the problem of estimating the coin's bias can be solved as above. However, what about if we do not know which coin was flipped?

When we do not know which coin was flipped, our table can be rewritten as follows:

Steps	**Coin**	**Trial**	**# coin A heads**	**# coin B heads**
1	?	HTTTHHTHTH	?	?
2	?	HHHHTHHHHH	?	?
3	?	HTHHHHHTHH	?	?
4	?	HTHTTTHHTT	?	?
5	?	THHHTHHHTH	?	?
6	?	HTTHTHHHHH	?	?

This means that we do not know about the target variable (coin A or B). In this context, the EM algorithm can be used to estimate the bias of each coin and, therefore, to identify the coins based on their biased trials. For the expectation step (E-step), assume that the initial biases are $s_A = 0.3$ and $s_B = 0.8$, when the trial HTTTHHTHTH (or event E) is presented. Considering the initial biases, the probability of these flips to come from coin A given the event E can be computed by:

$$P(A|E) = P(A|HTTTHHTHTH) = \frac{10!}{5!5!}0.3^5(1-0.3)^5 \tag{16.1}$$

The probability of these flips to come from coin B can be found by:

$$P(B|E) = P(B|HTTTHHTHTH) = \frac{10!}{5!5!}0.8^5(1-0.8)^5 \tag{16.2}$$

It is also possible to apply the Bayes theorem:

$$P(A|E) = \frac{P(E|A)P(A)}{P(E|A)P(A) + P(E|B)P(B)} \tag{16.3}$$

and

$$P(B|E) = \frac{P(E|B)P(B)}{P(E|B)P(B) + P(E|A)P(A)} \tag{16.4}$$

More generally, for an event E where the number of heads is h and the number of tails is $t = 10 - h$:

$$P(A|E) = \frac{s_A(1-s_A)^t}{s_A(1-s_A)^t + s_B(1-s_B)^t} \tag{16.5}$$

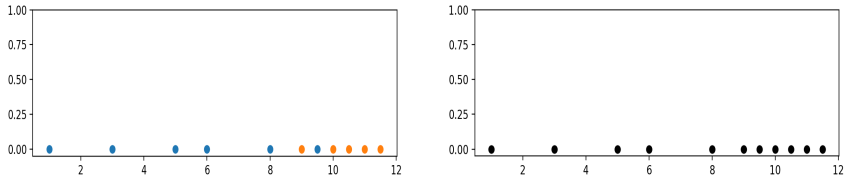
$$P(B|E) = \frac{s_B(1-s_B)^t}{s_B(1-s_B)^t + s_A(1-s_A)^t} \tag{16.6}$$

We can then obtain estimates (i.e., the probability to belong to a specific coin) for each trial (event) on the table above and check whether it is more likely being a result of repeatedly tossing the coin A or B. Thus we assign a coin to each event (E-step) and update the parameters s_A and s_B (M-step) so that s_A will now consist in the mean of the proportions of tails given all trials assigned to coin A. s_B can be similarly computed. These steps are repeated until the algorithm convergence is achieved.

16.1.2 A 1-dimensional numeric dataset experiment

The previous subsection presented a discrete problem where the Bayes theory fits very well the characteristics of the problem. Figure 16.1 shows a 1-dimensional numeric dataset consisting of two classes. In the case of Fig. 16.1 (a), the labels (or the target variable $\mathbf{z}$) are present whereas in 16.1 (b) they are not. Given the scenario of Fig. 16.1 (a), it would be easy to create a model to represent the problem by building two Gaussian models (one for each class) based on the mean (μ) and variance (σ^2) of the examples of each class. Nevertheless, the scenario of Fig. 16.1 (b), where the classes' labels are missing, is a typical unsupervised problem where the EM algorithm can be applied.

Taking Fig. 16.1 (b) as example, initially the EM algorithm instantiates two Gaussian functions based on random parameters (μ_o, σ_o^2) and (μ_b, σ_b^2) for clusters *orange* and *blue*, respectively. After that, the probabilities of each example to belong to each cluster are computed. The first step is to compute the probability of each example to come from each Gaussian model (cluster) according to Equation 16.7 (where the probability of the example $\mathbf{x}^{(i)}$ belong to the cluster *blue* is computed):



(a) Labeled dataset. (b) Unlabeled dataset

Fig. 16.1: 1-dimensional dataset.

$$P(\mathbf{x}^{(i)}|b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} exp\left(-\frac{(\mathbf{x}^{(i)} - \mu_b)^2}{2\sigma_b^2}\right) \tag{16.7}$$

Then, the Bayesian posterior probability of the same example being from the class *blue* can be computed according to Equation 16.8:

$$b_i = P(b|x^{(i)}) = \frac{P(\mathbf{x}^{(i)}|b)P(b)}{P(\mathbf{x}^{(i)}|b)P(b) + P(\mathbf{x}^{(i)}|o)P(o)} \tag{16.8}$$

notice that in the first step of the EM, the priors $P(b)$ and $P(o)$ are equal and both can be 0.5, for example. For computing the posterior probability for the orange class we can use the already computed posteriors of the blue class as depicted in Equation 16.9:

$$o_i = P(o|\mathbf{x}^{(i)}) = 1 - b_i \tag{16.9}$$

After that, the posterior probabilities for each class and examples will be computed and the Gaussian models will be shifted towards the examples that are better represented by them. For example, the blue cluster will have its parameters (mean and variance) adjusted based on the following equations, respectively:

$$\mu_b = \frac{b_1\mathbf{x}^{(1)} + b_2\mathbf{x}^{(2)} + \ldots + b_n\mathbf{x}^{(n)}}{b_1 + b_2 + \ldots + b_n} \tag{16.10}$$

$$\sigma_b^2 = \frac{b_1(\mathbf{x}^{(1)} - \mu_b)^2 + \ldots + b_n(\mathbf{x}^{(n)} - \mu_b)^2}{b_1 + b_2 + \ldots + b_n} \tag{16.11}$$

After that, the priors $P(o)$ and $P(b)$ can already be more precisely estimated and the process continues until a stop criterion is reached.

Therefore, considering the example of Figure 16.1b where $\mathbf{X} = \{0.1, 0.3, 0.5, 0.6, 0.8, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15\}$, the two Gaussian functions, blue and orange, have parameters ($\mu_b = 0.1$, $\sigma_b^2 = 0.01$) and ($\mu_o = 0.3$, $\sigma_o^2 = 0.01$). In order to obtain the outcomes from the pdf functions between 0 and 1, the highest *peak* between both Gaussian functions was found and each $P(\mathbf{x}^{(i)}|c)$ (where c stands for the class) was normalized/divided by this *peak*. The re-

sult of the first step of the EM is depicted in Figure 16.2. The left plot shows the initial state of the Gaussian models while the right plot shows the state of the Gaussian functions after the first M-step.

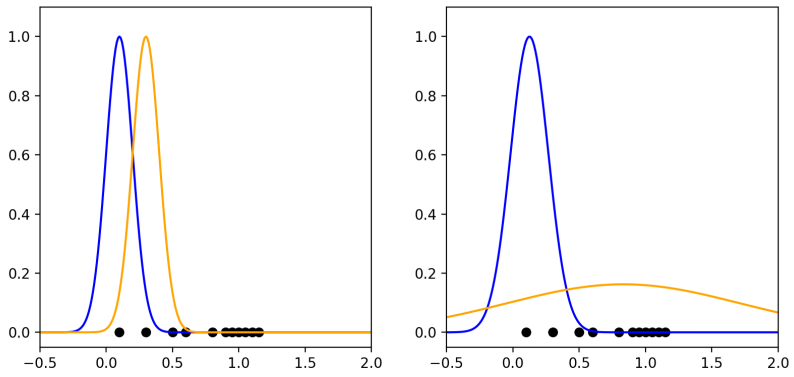


Fig. 16.2: First step of the EM for an 1-dimensional artificial dataset.

It is noticeable that the EM algorithm works similarly to the k-means algorithm, but instead of creating hard margins (i.e., no overlap among clusters) the EM creates soft margins (i.e., the probabilities of an example ($\mathbf{x}^{(i)}$) to belong to different clusters are not 0). In addition, the example exposed in this section considered a 1-dimensional dataset. For problems in the n-dimensional hyperspace (i.e., $n \geq 2$) it is necessary to use a covariance matrix instead of the simple variance.

16.2 A Gentle Introduction to Expectation Maximization

As seen in the examples in the previous sections, the EM algorithm represents the examples from each class by using a probabilistic model for each class distribution. To represent these distributions in the same hyperspace is a problem known as **mixture models** that employs a systematic way (i.e., Maximum Likelihood Estimation - MLE) to find the parameters for each probabilistic model that better fit its respective group of examples (cluster). A mixture model can be roughly represented by the following two equations:

$$p(\mathbf{x}) = \sum_{i=1}^{k} \pi_i p_i(\mathbf{x}) \tag{16.12}$$

$$\sum_{i=1}^{k} \pi_i = 1 \tag{16.13}$$

In Equations 16.12 and 16.13 k represents the number of statistical models in the mixture and $p(\mathbf{x})$ stands for the probabilistic function regarding any statistical distribution (e.g., Bernoulli, Gaussian (Normal), etc.) and π_i represents the weight of the ith probabilistic model.

The Gaussian statistical models stand as the most adopted models for the EM algorithm. Therefore, Gaussian Mixture Models (GMMs) [8] can be seen as a special case of the EM algorithm. The GMMs use a combination of Gaussian (or Normal $N(\mu, \Sigma)$) models such that the parameters to be optimized are the mean ($\boldsymbol{\mu}$) and covariance matrix ($\boldsymbol{\Sigma}$) from each Gaussian model. Thus, the posterior probability p can be rewritten as follows:

$$p(\mathbf{x}|\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \ldots, \boldsymbol{\Sigma}_k) = \sum_{i=1}^{k} \pi_i N(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k). \tag{16.14}$$

For simplicity, the set of parameters $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \ldots, \boldsymbol{\Sigma}_k\}$ will now be denoted by θ.

The aim of a GMM, ruled by the set of parameters θ, is to precisely fit a dataset $\mathbf{X}$. For this it is necessary to estimate θ by maximizing the following likelihood function:

$$log f(\mathbf{X}|\theta) = \sum_{i=1}^{N} log \sum_{j=1}^{k} \pi_j N(\mathbf{x}^{(i)}|\mu_j, \Sigma_j) \tag{16.15}$$

At this point, our challenge is to optimize the parameters from Equation 16.15. However, given the complex relationship among the parameters of the different Gaussian models it is not possible to use a mathematical framework such as gradient descent, for example. The aim of the EM algorithm is, then, to iteratively solve this problem.

As expected, each Gaussian model must represent a subset of examples in $\mathbf{X}$. The individual influence of a particular Gaussian model (indexed by k) over an example $\mathbf{x}$ can be computed as follows:

$$z_k(\mathbf{x}) = \frac{\pi_k N(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{i=1}^{K} N(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)}. \tag{16.16}$$

So, $z_k(\mathbf{x})$ can be seen as the probability of the example $\mathbf{x}$ being represented by the kth Gaussian model. As aforementioned, there is not a formal method for estimating the set of parameters θ, however, these parameters can be updated as follows:

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N_k} z_k(\mathbf{x}^{(i)}) \mathbf{x}^{(i)} \tag{16.17}$$

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N_k} z_k(\mathbf{x}^{(i)}) (\mathbf{x}^{(i)} - \boldsymbol{\mu}_k^{new})(\mathbf{x}^{(i)} - \boldsymbol{\mu}_k^{new})^T \tag{16.18}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{16.19}$$

In Equations 16.17, 16.18 and 16.19, $N_k = \sum_{i=1}^{N} z_k(\mathbf{x}^{(i)})$. In addition, they are not supposed to directly maximize Equation 16.15. In fact, they seek for the optimal parameters of the following similar Equation:

$$\hat{f}(\mathbf{X}|\theta) = \sum_{i=1}^{N} \sum_{j=1}^{k} z_j(\mathbf{x^{(i)}}) log \frac{\pi_j N(\mathbf{x}^{(i)}|\mu_j, \Sigma_j)}{z_j(\mathbf{x^{(i)}})} \tag{16.20}$$

Figure 16.3 shows an example of a Gaussian Mixture Model (GMM) using three Gaussian models. The mean parameter of the Gaussian models are, respectively, 5, 10 and 17; they share the same variance, i.e., 2. Dotted lines indicate the Gaussian models and the continuous line shows the resulting GMM. Each Gaussian model has $\pi_k = \frac{1}{3}$.

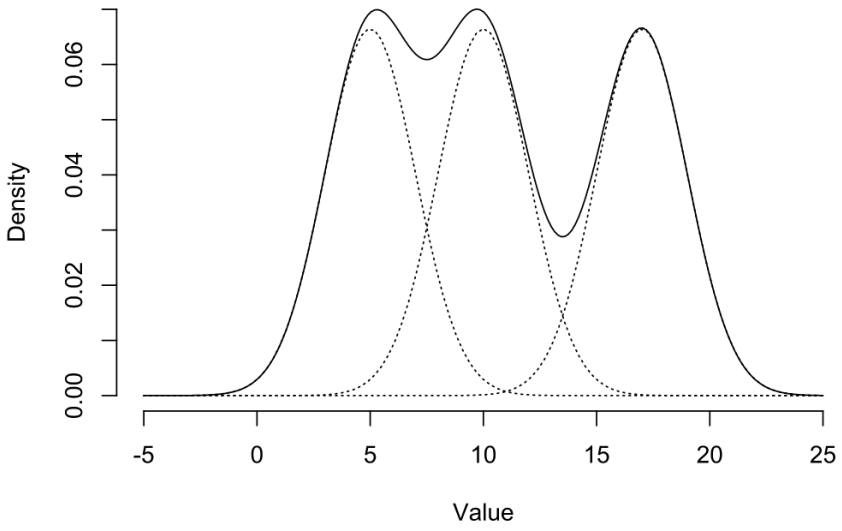


Fig. 16.3: Example of Gaussian mixture. Adapted from [9].

16.2.1 The role of the Covariance Matrix ($\boldsymbol{\Sigma}$)

The covariance matrix $\boldsymbol{\Sigma}$ provides the algorithm with the ability to identify clusters of different shapes and sizes. The diagonal values of the matrix contains the variances of the respective features of the problem: the bigger these values, the more spread out the cluster is. Values that outsize the matrix diagonal quantifies the covariance between pairs of problem features. From covariance values the correlation between problem features can be computed.

In order to illustrate the importance of covariance matrix $\boldsymbol{\Sigma}$, Figure 16.4 shows an example of a simple data set and resulting partitions after applying the well known K-Means algorithm and the EM algorithm for the spacial case when $\boldsymbol{\Sigma}$ is a diagonal matrix.

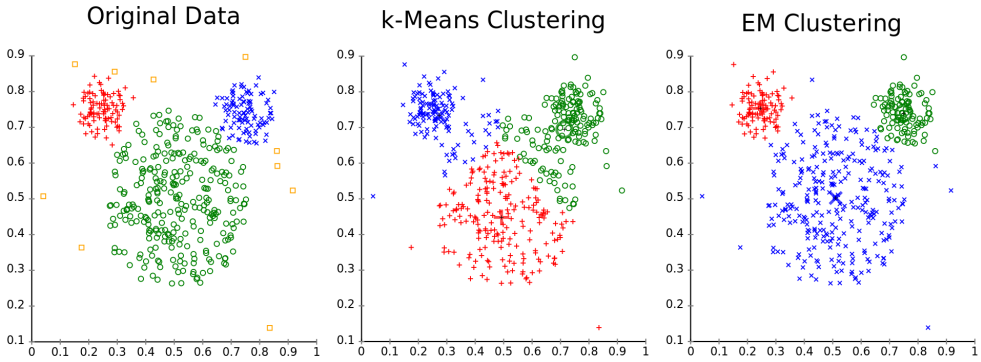


Fig. 16.4: Example of partitions found by K-Means and EM clustering algorithms. Adapted from [10].

The original data set contains three classes with different spherical sizes. After applying K-Means clustering algorithm, the resulting partition is different from original one since this algorithm divides the space into clusters of equal dispersion. In other words, K-Means finds spherical clusters of equal sizes. On the other hand, after applying EM algorithm, the resulting partition is similar to original one. This is because EM algorithm is able to create a model that identifies different distributions, which allows to find a resulting partition with clusters of different sizes and shapes.

16.3 The EM Algorithm

The EM algorithm can be characterized by two main steps: Expectation (E-step) and Maximization (M-step). In the E-step, the algorithm computes the expected value of a dataset example using the current estimate of the

parameter. In the M-step, the algorithm uses information from previous step to define a maximum-likelihood estimate of the parameters. Figure 16.5 shows an overall flow chart for the EM algorithm.

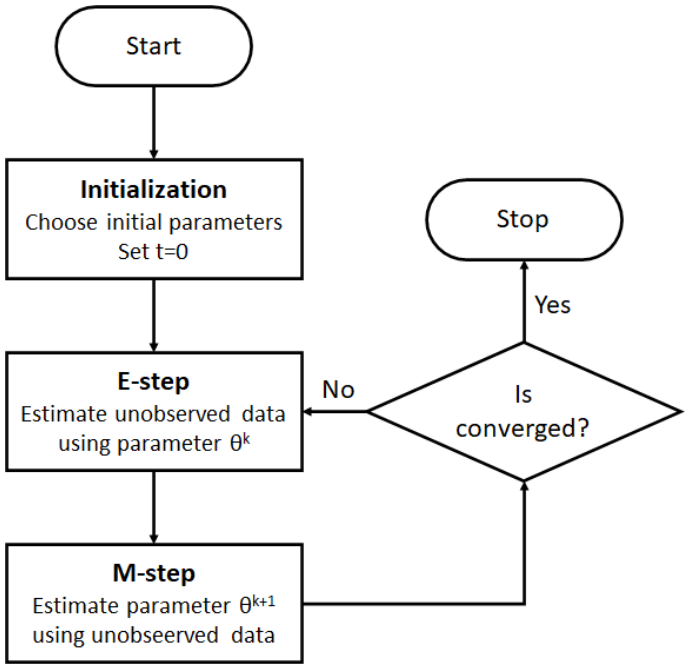


Fig. 16.5: Flow chart for EM algorithm.

Algorithm 20 presents the pseudo-code for the EM method. The E-step and M-step are executed until one or more convergence criterion are satisfied. Examples of convergence criteria are presented in Algorithm 20.

16.4 Applications

Due to the importance of the EM algorithm for the Machine Learning field, several researchers have applied it to a wide variety of real problems. These problems are concerned about Natural Language Processing, Signal Processing, medical image reconstruction and so on. This section briefly present some studies that applied EM algorithm as tool for solving real problems.

Ramme et al. (2009) [11] used the EM algorithm to segment the hand phalanx bones. The goal of this work is to analyze whether a semi-automated

Algorithm 20 EM algorithm

Parameters: Dataset $\mathbf{X}$, number of clusters k.
Output: Final partition.

1: Choose initial parameters. Set time $t = 0$. Set ϵ.
2: **repeat**
3: **E-step**: find the values π_k (for $k = 1, 2, \ldots, k$) using the Equation 16.19
4: **M-step**: update the parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ (for $k = 1, 2, \ldots, k$) using equations 16.17 and 16.18, respectively
5: **until** one of the following criteria is satisfied:

- the maximum number of iterations is reached;
- there is no more difference between the current and previous partitions; or
- the difference between the likelihood function values at iteration t and $t - 1$ is smaller than a predefined threshold ϵ.

technique will improve the efficiency while providing similar definitions as compared to a human rater.

Kujawinska et al. (2016) [12] applied the EM algorithm to support purchasing decisions in the welding industry. Authors analyzed the EM for the selection of material (212 combinations of flux and wire melt) for the SAW (Submerged Arc Welding) method process. The work showed that each of the 212 records can be assigned with a probability of affiliation to all the four clusters used in the experiments. These probabilities allow the user to make a better decision, since cases with ambiguous assignment can be better analyzed.

Subudhi et al. (2020) [13] used the Expectation Maximization method and the Random Forest classifier for automated segmentation and classification of brain stroke. The part of the brain affected by the stroke was segmented using the EM algorithm and Magnetic Resonance Imaging (MRI) of brains.

Lakshminarayanan et al. (2020) [14] recovered the high-resolution image of a corresponding low-resolution image. For this, the authors proposed a new integrated approach based on the iterative super-resolution algorithm and the EM method for face hallucination (the process of improving a low resolution – LR – image to a high resolution – HR – image without altering the originality of the image).

16.5 Exercises

1. Based on the experiment presented in Section 16.1.1, use EM algorithm to estimate the bias of each coin after 100 iterations.
2. Use 300 samples of Gaussian Mixture with mixing probability equal to 1/3 as follow:

$$X_1 \sim N\left(\begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}1 & 4\\4 & 1\end{bmatrix}\right)$$

$$X_2 \sim N\left(\begin{bmatrix}5\\5\end{bmatrix}, \begin{bmatrix}4 & 2\\2 & 6\end{bmatrix}\right)$$

and implements the EM algorithm to estimate its parameters.

3. Show a scatter plot of the above dataset for different values of mixing probability: 1/2, 1/3, 1/4 and 1/5. What can you observe from the results?
4. Changing the dataset to:

$$X_1 \sim N\left(\begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}1 & 4\\4 & 1\end{bmatrix}\right)$$

$$X_2 \sim N\left(\begin{bmatrix}5\\5\end{bmatrix}, \begin{bmatrix}4 & 2\\2 & 6\end{bmatrix}\right)$$

$$X_3 \sim N\left(\begin{bmatrix}3\\3\end{bmatrix}, \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\right)$$

repeat the question 3. What can you observe from the new results?

References

1. Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.
2. Chuong B Do and Serafim Batzoglou. What is the expectation maximization algorithm? *Nature biotechnology*, 26(8):897–899, 2008.
3. Frank Dellaert. The expectation maximization algorithm. Technical report, Georgia Institute of Technology, 2002.
4. Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.
5. By R CEPPELLINI, M Siniscalco, and CAB Smith. The estimation of gene frequencies in a random-mating population. *Annals of Human Genetics*, 20(2):97–115, 1955.
6. Elad Tzoreff and Anthony J Weiss. Expectation-maximization algorithm for direct position determination. *Signal processing*, 133:32–39, 2017.
7. Xia Li, Zhisheng Zhong, Jianlong Wu, Yibo Yang, Zhouchen Lin, and Hong Liu. Expectation-maximization attention networks for semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9167–9176, 2019.
8. Sylvia Richardson and Peter J Green. On bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: series B (statistical methodology)*, 59(4):731–792, 1997.
9. Smason79. Gaussian mixture example. `https://commons.wikimedia.org/wiki/File:Gaussian-mixture-example.svg`, 2010. [Online; accessed 09-September-2021].
10. Chire. Cluster analysis performed on an artificial dataset ("Mouse", similar to a well-known comic figure) comparing k means and EM clustering results. `https://commons.wikimedia.org/wiki/File:ClusterAnalysis_Mouse.svg`, 2010. [Online; accessed 08-September-2021].

11. Austin J Ramme, Nicole DeVries, Nicole A Kallemyn, Vincent A Magnotta, and Nicole M Grosland. Semi-automated phalanx bone segmentation using the expectation maximization algorithm. *Journal of digital imaging*, 22(5):483–491, 2009.
12. Agnieszka Kujawińska, Michał Rogalewicz, and Magdalena Diering. Application of expectation maximization method for purchase decision-making support in welding branch. *Management and Production Engineering Review*, 7, 2016.
13. Asit Subudhi, Manasa Dash, and Sukanta Sabut. Automated segmentation and classification of brain stroke using expectation-maximization and random forest classifier. *Biocybernetics and Biomedical Engineering*, 40(1):277–289, 2020.
14. K Lakshminarayanan, R Santhana Krishnan, E Golden Julie, Y Harold Robinson, Raghvendra Kumar, Le Hoang Son, Trinh Xuan Hung, Pijush Samui, Phuong Thao Thi Ngo, Dieu Tien Bui, et al. A new integrated approach based on the iterative super-resolution algorithm and expectation maximization for face hallucination. *Applied Sciences*, 10(2):718, 2020.

Chapter 17
Self Organizing Maps (SOM)

George G. Cabral

Self Organizing Maps (SOM) were initially introduced by Teuvo Kohonen [1, 2] in the early 1980s inspired by the way the cerebral cortex handles sensory information, such as visual, olfactory and auditory information. The auditory cortex in the auditory system, for example, can be roughly represented as a surface (or map) so that different regions process different sound frequencies. In addition, nearby regions are responsible for processing similar frequencies. These concepts were successfully adapted from neuroscience to computer Artificial Neural Networks (ANNs) through the SOM networks. Among all types of ANNs, perhaps, the ANN architecture is the one which better resembles the human brain functioning. This chapter provides an introduction to SOM.

17.1 An Overview of SOM's Mechanisms

For practical purposes, a SOM network model uses as topological maps 2d or 3d grids such that each grid cell represents a neuron. Nevertheless, these maps can have different topological organizations. Figure 17.1 shows maps using two different topologies, squares and hexagons. In these maps, each unit (square or hexagon) represents an output neuron. Most of real world problems lie in a high dimensional space. The map can be seen as a lower dimensional representation of the space. Finding a SOM model which maps the original space to a low dimensional space may result in, among the benefits, less data storage requirements, for example.

Given a set of training examples $\mathcal{T} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \cdots, \mathbf{x}^{(N)}\}$, where each training example $\mathbf{x}^{(n)}$ is a d-dimensional vector, a training **epoch** consists in the presentation of the n training examples to the classifier for learning

Department of Computing, Federal Rural University of Pernambuco, BR

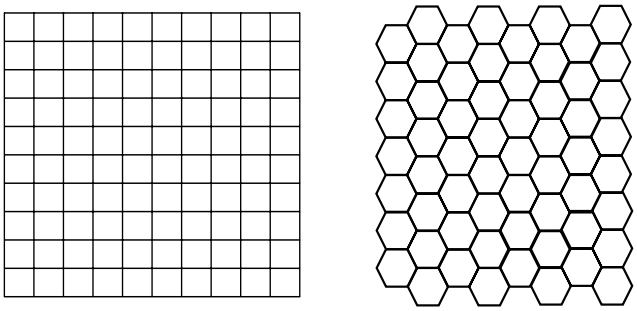


Fig. 17.1: Examples of 2d maps. (left) squared map and (right) hexagonal map.

purposes. Nevertheless, the whole training procedure takes several epochs, often, hundreds or more, i.e., the n training examples are presented multiple times. The learning phase of a self organization involves 3 major concepts from neuroscience.

Competition As a training example $\mathbf{x}^{(n)}$ is presented to the network, each neuron competes against the others in such a manner that the winner neuron is the neuron with its weights closer to $\mathbf{x}^{(n)}$'s features. Notice that each neuron, therefore, has a set of d weights associated to it (provided that d is the number of problem features). The measure of how close a neuron is to a training example can be computed by the Euclidean distance between two examples (Equation 17.1).

$$EuclidDist(\mathbf{x}^{(i)}, \mathbf{z}) = \sqrt{(x_1^{(i)} - z_1)^2 + (x_2^{(i)} - z_2)^2 + \cdots + (x_j^{(i)} - z_j)^2} \tag{17.1}$$

Cooperation In the cerebral cortex, each received sensory stimulation results in a specific region which achieves a higher activation value - the winner neuron in the competition phase. Nevertheless, locations close to the winner region (the neighbourhood) are also activated by a smaller magnitude. This magnitude decreases and tends to zero as the regions get far from the winner neuron. In a SOM ANN, this concept is adapted by computing a neighbourhood to each neuron. Figure 17.2 shows an example where the winner neuron (red) has its weights reinforced by a factor of 1, its closest neighbours (orange) have their weights reinforced by a factor of 0.5 and its indirect neighbours (yellow) have their weights reinforced by a factor of 0.25.

A popular way to find the topological factor for weights update of the neighbour neurons is given by the use of a Gaussian function [3]. In order to rule the winner neuron neighbourhood, a function must possess two important properties: (1) to provide symmetric values from the winner neuron and (2) decrease monotonically as the distance from the winner neuron gets

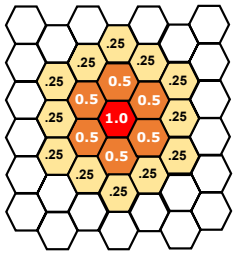


Fig. 17.2: Example of neighbourhood and factors values of weight update according to the distance a of the neurons to a winner neuron.

larger. The Gaussian function, presented in Equation 17.2 has both properties. In Eq. 17.2, (i) $d_{a,b}$ consists in the distance between the winner neuron a and its neighbour neuron b and (ii) σ_t is a decay factor at epoch t that rules the coverage of the neighbourhood.

$$F_{a,b}(t) = exp(-d_{a,b}^2/2\sigma_t^2) \tag{17.2}$$

Ideally, as the time passes, σ decreases, eventually, resulting in a neighbourhood comprising only the winner neuron. This happens because, initially, the SOM is coarsely learning the problem. As the time passes, the algorithm starts to refine the learning process to learn the problem's details. A common method to decrease σ is given in Equation 17.3. In Eq. 17.3, τ_1 is a constant and σ_0 is the first value assigned to σ.

$$\sigma_t = \sigma_0 exp(-t/\tau_1) \quad t = 0, 1, 2, 3, ... \tag{17.3}$$

For the multivariate case, a positive-definite covariance matrix must be defined. Similarly to the univariate case, shrinking the values in this matrix exponentially decreases the width of the neighbourhood function.

Notice that the map topology may slightly influence the clusters' organizations. For example, in a square topology (Figure 17.1 (left)), a neuron may have up to eight direct neighbors that supposedly should be at the same distance to the winner neuron, however, the four neurons in the corner are more distant than the other four. In this case, choosing a small map (a map with few neurons) may lead to unsmooth separations between clusters. On the other hand, in the hexagon map (Figure 17.1 (right) and Figure 17.2), a winner neuron may have up to six closest neighbors all of them with the same distance to it. In this case, the topology allows a smother interaction among the winner neuron and its neighbors.

Adaptation The adaptation (or learning) process takes place as the outputs neurons learn the problem making the topological map organized such that the clusters can be identified as the algorithm converges to a solution. As seen in the cooperation phase, not only the winner neuron moves toward a

training example. Instead, the entire neighbourhood moves toward it. Once the winner neuron is found, its weights are updated according Equation 17.4. The learning rate $\eta(t)$, as well as the neighbourhood size, also exponentially shrinks as the time passes. Therefore, one plausible way to shrink the learning rate is given by $\eta(t) = \eta_0(-t/\tau_2)$, where τ_2 is a constant and η_0 is the first value assigned to the learning rate.

$$\Delta w_{j,i}(t) = \eta(t) \times F_{j,i}(t) \times (x_i - w_{j,i}) \tag{17.4}$$

In short and recapping, in practice, (i) during the **competition phase** a winner neuron (**c**) is found, (ii) in the **cooperation phase** a set of **c**'s neighbors in the topological map is defined and (iii) in the **adaptation phase** the weights of the winner neuron and its neighbors are adjusted.

17.2 The Self Organizing Map Algorithm

The vast majority of the works define the SOM topology as containing only the input and output layers. The input layer, as most of the ANNs, acts as sensors so that the number of neurons is the same as the number of features of the problem. Therefore, the domain of the input layer is $\mathbb{R}^d$, where d is the number of features of the problem. The output layer (or the Kohonen layer) is usually a 2-dimensional map (Figure 17.3) where each cell represents a different neuron. The input layer is connected to each output neuron such that the number of connections is given by $d \times j$, where j is the number of output neurons. Figure 17.3 depicts an example of SOM architecture where $j = 88$. The input layer, is formed by d neurons fully connected to each output neuron and each connection between input neuron i and output neuron j has an associated weight $w_i^{(j)}$. In Figure 17.3, the connections between the input layer and the output neuron 1 (one) are highlighted in red.

Algorithm 21 depicts the operation sequence of a SOM neural network for a problem containing d features and j output neurons. Initially (line 1), all the weights for the connections among the input neurons and output neurons are randomly set. In line 4, a training example $\mathbf{x}^{(i)}$ is presented and the closest neuron to it in the map (c) is chosen as winner. The weights of c then "move toward" $\mathbf{x}^{(i)}$ considering the learning rate $\eta(t)$ (lines 6 and 7). The same adjustment happens to the weights of the neighbors of c in the map, however, in this case the adjustment is ruled by an additional distance factor computed by the neighborhood function $F_{x,y}(t)$ so that neighbors closer to c have their weights strongly updated (lines 11 and 12). Lines 16 and 17 are responsible for the update of parameters that affect the fine convergence of the algorithm in later epochs.

At the end of the training phase, it is expected that output neurons in a same topological region represent a cluster (or class). In particular, examples

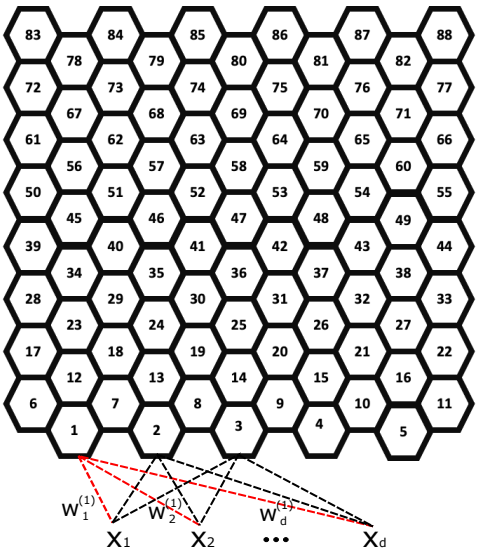


Fig. 17.3: Partial example of SOM architecture. The connections between the input layer and the output neuron 1 are highlighted in red.

that are similar to each other are mapped to the same or nearby regions of the topological map.

17.2.1 Example of the Algorithm Execution

In order to provide a comprehensive understanding of the canonical SOM algorithm, Figure 17.4 presents a 3-dimensional dataset containing 16 colors in the RGB format which will be learned so that the main stpdf of the algorithm can be shown. Notice that the clustering algorithms assume that there is redundant information in the training set, therefore, we inserted variations of the dominant colors blue, green, black and brown in the dataset.

Initially, the topological map must be defined. For this example a (20, 20) matrix where each neuron is 1 unit distant from its nearest neighbor in the same row (or column) was used. In the sequel, for each neuron b, each connection's weight $w_k^{(b)}$ (where k varies from 0 to 2) is randomly assigned between 0 and 1. In other words, for this problem we can say that, initially, each output neuron will be associated to a random RGB color (Figure 17.5 a)).

The algorithm parameters were as follows:

- $\eta_0 = 0.5$ (initial learning rate)

Algorithm 21 Self Organizing Map

Parameters: output map containing j neurons and $d \times j$ connections with respective weights ($w_d^{(j)}$) between input and output layers, neighborhood function F, τ_1, τ_2, η_0, n_epochs, training set $\mathcal{T}$, θ

Output: SOM model with proper weights between input and output layers

Set a random value for each of the $j \times d$ weights at each connection between the input and output layers. These values must be in the same interval of the feature values in $\mathcal{T}$.
for $t \in \{1 \cdots n\_epochs\}$ **do**
  for $\mathbf{x}^{(i)} \in \mathcal{T}$ **do**
    Given $\mathbf{x}^{(i)}$, find the winner neuron c
    for $k \in \{1 \cdots d\}$ **do**
      $\Delta w_k^{(c)}(t) = \eta(t) \times EuclidDist(x_k^{(i)}, w_k^{(c)})$
      $w_k^{(c)} = w_k^{(c)} + \Delta w_k^{(c)}(t)$
    end for
    for each neighbor $neigh$ of c **do**
      for $k \in \{1 \cdots d\}$ **do**
        $\Delta w_k^{(neigh)}(t) = \eta(t) \times F_{c,neigh}(t) \times EuclidDist(x_k^{(i)}, w_k^{(neigh)})$
        $w_k^{(neigh)} = w_k^{(neigh)} + \Delta w_k^{(neigh)}(t)$
      end for
    end for
  end for
  $\{\sigma_t$ is used in the neighborhood function $F_{x,y}(t)\}$
  $\sigma_{t+1} = \sigma_t exp(-t/\tau_1)$
  $\eta(t+1) = \eta_t(-t/\tau_2)$
end for

- $\theta = 0.05$ (threshold for which the neighborhood function is considered)
- $\sum = [1.5, 0.0; 0.0, 1.5]$ (covariance matrix for defining the Gaussian neighborhood function)
- For this example, τ_1 and τ_2 weren't used since the learning rate and neighbourhood size weren't decreased.

Notice that some of these parameters values are unlikely to be used in practice (as the initial learning rate value of 0.5, for example), but in this case, they were chosen for illustration purposes.

The very first step of the algorithm searches for the nearest neighbor (lets call it b) in the map of the training example of index [0] in Figure 17.4 (lets call it a), i.e., (0.49, 0.10, 0.54). The nearest neighbor output neuron in the map is the neuron with closest weights, in this case (0.46 0.31 0.59). Therefore, the new weights of b are computed as follows:

$$W_{b,0}(t) = W_{b,0}(t-1) + \eta(t) \times F_{a,b}(t) \times (x_0^{(a)} - w_0^{(b)}) = 0.46 + 0.5 * 1 * (0.49 - 0.46)$$

$$W_{b,1}(t) = W_{b,1}(t-1) + \eta(t) \times F_{a,b}(t) \times (x_1^{(a)} - w_1^{(b)}) = 0.46 + 0.5 * 1 * (0.10 - 0.31)$$

$$W_{b,2}(t) = W_{b,2}(t-1) + \eta(t) \times F_{a,b}(t) \times (x_2^{(a)} - w_2^{(b)}) = 0.46 + 0.5 * 1 * (0.54 - 0.59)$$

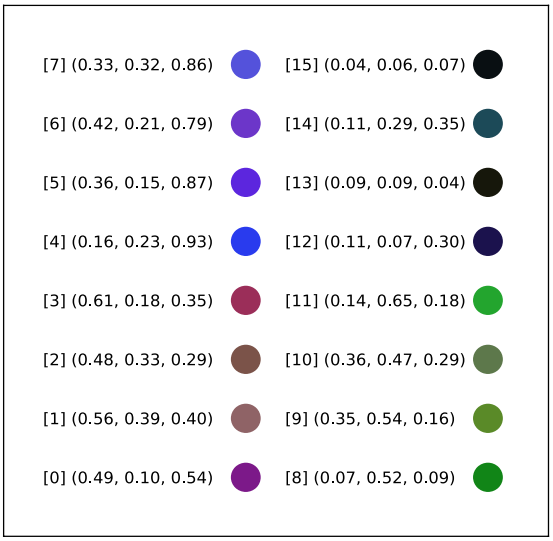


Fig. 17.4: Training set containing 16 RGB colors where each color component is normalized between 0 and 1 and the presentation order of the examples is indexed in square brackets.

In the sequel, all neighbours neurons, i.e., the neurons whose the scaled neighbourhood function yields a larger value than θ have theirs weights adjusted accordingly this value.

This procedure is repeated for each training example in order to complete a training epoch. The total number of epochs is defined by the practitioner.

Figure 17.5 depicts part of the evolution of the algorithm computation above presented in terms of the topological map. Figure 17.5 a) shows the initial state of the topological map where each neuron represents a random RGB color. In Figure 17.5 b), the winner neuron of the 1st epoch and 1st training iteration (b) with weights (0.46 0.31 0.59), has its neighbourhood defined (crosses points) so that for each neighbour z $F_{b,z}(t) > \theta$. Notice that the winner neuron is computed according the neuron weights and the neighbourhood is defined based on the topological distance in the map to the winner neuron. Figure 17.5 c) and d) also refer to the first epoch shows the computation for the winner neurons for the training set examples positioned at indexes 4 and 11. Figure 17.5 e) presents an example of computation for the 6th epoch. In this stage is already possible to notice some clusters. Figure 17.5 f) represents the topological map at 20th epoch. At this stage, the map is close to the convergence due to the small number of examples in the training set.

Implementation Issues

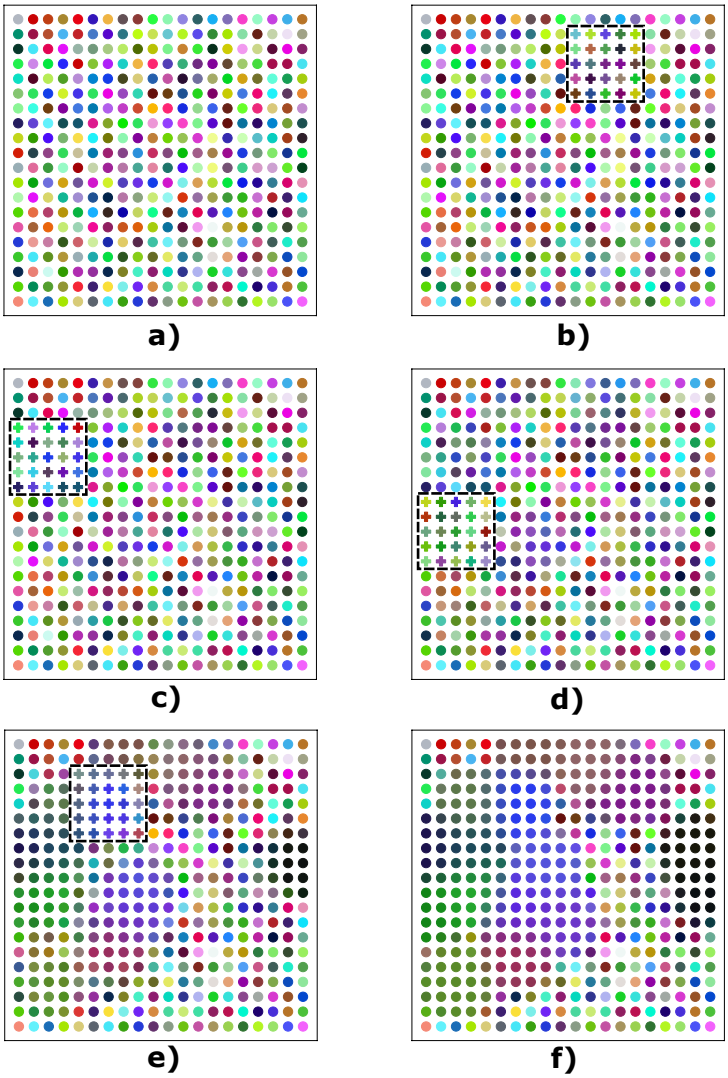


Fig. 17.5: a) Original random topological map; b) Epoch 0 - training example [0] neighbourhood after their weights' updates; c) Epoch 0 - training example at position 4's neighbourhood after their weights' updates; d) Epoch 0 - training example at position 11's neighbourhood after their weights' updates; e) Epoch 5 - training example at position 4's neighbourhood after their weights' updates; f) Map status at epoch 20.

The PDF defined by the vector b and $\sum$ is comprised in the interval (0, r), where 0 is located at the infinite and r is the largest value located at the mean b. For the current implementation, r was scaled between 0 and 1. This explains the value 1 for the neighbourhood function in the previous weights adjustments.

Particularly for this illustration example, the small number of examples in the training set combined to the small variation in these examples caused, from a specific epoch, the winner neurons converge to the same neurons again and again. This explains some untouched areas in the map, such as the last row.

17.3 Growing Self Organizing Map (GSOM) - A Dynamic version of SOM

The GSOM was initially developed by Alahakoon and Srinivasan [4] who claimed that their method had significant advantages for knowledge discovery over the original one. The main difference between both methods consists in a *growing* topological map introduced in the GSOM. Furthermore, given the growing feature of the GSOM, its spreading factor enables the practitioner to track hierarchical aspect of the clusters, i.e., one can find the path from a finer cluster to a higher level cluster.

The following high level stpdf depicts the GSOM algorithm:

1. Initialization Phase
    a. Randomly assign weights to, usually, 4 neurons connected in a 2d grid.
    b. Compute the Growth Threshold (GT) based on pre-defined parameters.
2. Growing Phase
    a. Present a training example to the network.
    b. Find the winner neuron, similar to the original SOM.
    c. Update weights values of the winner neuron and its neighbors as well as in the original SOM method
    d. The difference between the weights and the training example will be used as an error to be accumulated in the winner neuron as its total error (TE)
    e. If TE > GT and the winner neuron is a boundary neuron, grow it (i.e., place new neurons in free positions in the neighbourhood of the current neuron. For example, above or at right.). Otherwise, distribute its weights to the neighbors of the current neuron.
    f. Initialize the new nodes weights to match the current neighbourhood
    g. Reset the learning rate to its starting value

h. Repeat stpdf a. to g.

3. Smoothing phase

   a. Decrease the learning rate and fix a small neighbourhood.
   b. Repeat weights adaptation as in the growing phase.

Amongst the advantages of the GSOM over the SOM are its capability to represent more complex topological maps and the possibility to discover hierarchical relationships, as previously mentioned.

This is a high level description of the GSOM. Many details, such as new nodes growth situations, new node weights assignment and neighborhood definition, were uncovered since this is not the aim of this text. For more details, the introductory paper [4] as well as available implementations on the web can be helpful.

17.4 Applications

Virtually, SOM networks can be applied to any knowledge area. Nevertheless, some of them have been particularly benefited from its potential.

17.4.1 Dimensionality Reduction

Dimensionality reduction consists in transforming a problem from a high dimensional space to a low dimensional space. Among other, this enable us to: (i) better visualize relationships among the categories of the problem; (ii) reduce the noise effect on the data; (iii) identify clusters; and (iv) create models which serve as filters to reduce the data storage.

Dimensionality reduction is a natural application of Self Organizing Maps given that transforming a n-dimensional problem to a 2d or 3d map explicitly reduces the original problem data dimension. The popularity of SOMs networks amongst different science fields caused them to be subject of studies for the feature reduction for a number of different problems [5][6][7][8].

17.4.2 Surface Reconstruction

Reconstructing the surface of an object based on a set of examples structured or not is a common problem in many fields such as medicine [9], Arts [10] and manufacturing [11]. The idea is to reproduce the original shape based on a partial set of examples, i.e., these examples may not be able to represent missing parts of the shape. SOMs can generate a

map that is close to the original shape given a training set provided with the spatial coordinates of the points [12]. Commonly, reconstruction process of methods based on self organizing maps use randomly sampled 3d points as training data for the learning algorithm where the mesh vertices correspond to the output nodes of the map [12]. Some examples of images reconstructed from 3d points can be found at: `https://github.com/alecjacobson/common-3d-test-models`. Some examples of works that use SOMs or its variations for surface reconstruction are: [13], [14] and [12]

17.4.3 Image Segmentation

Image segmentation consists in identifying the boundaries (lines, curves, etc.) of the objects present in an image. Some goals behind this are to reduce the amount of information in order to better track video objects, identify areas of interest, etc.. As an example, in a image containing a group of persons, imagine that you want to identify faces so that from these faces you can eventually find some specific person. With this aim, we may need to separate the image in many regions such that pixels in the same region share similar characteristics, as color values, and there are no abrupt changes in these characteristics inside the same region. For this task, SOMs can act as an intermediary step [15][16][17][18]. Figure 17.6 depicts an example that can be used to segment images captured by an autonomous car sensor, for example.

Fig. 17.6: Segmentation for object identification in a traffic image. Source: Cityscapes Dataset [19] https://www.cityscapes-dataset.com/.

17.4.4 Text Mining

Clustering documents is one of the most important text mining research areas [20][21][22]. Even though some details are lost in this simplification, other information arise and, thus, we can observe the data from a macro point of view [23]. Liu et al. [23] also pointed out that text clustering can also act as

a pre-processing step for some natural language processing applications, e.g., automatic summarization, user preference mining, or be used to improve text classification results.

17.4.5 Speech Recognition

Speech recognition is an area that can vastly be benefited by self organizing maps, particularly by dynamic SOMs [24][25][26]. For example, customer services of big companies rely on automatic speech recognition to actuate on call centers in order to understand *yes* or *no* questions and diminish the need of human interaction. In addition, a number of electronic devices also use voice recognition tasks to translate voice commands to machine entries. These type of applications can employ Self Organizing Maps for the end of the application, e.g., for classify a voice signal as *yes* or *no*, or as an intermediary procedure, e.g., for pre-processing voice signals in categories.

17.5 Strengths and Weaknesses of SOM networks

One the main advantages of SOM networks it's that it can easily represent similarities in the data without the need of a complex fine tuning of the classifier's parameters (i.e., finding the output neurons' topology and the learning rate is not a complex task). The simplicity of the algorithm may be also seen as an advantage since knowing how the method behaves provides the practitioner with ability to customize it to domain specific problems.

Among the disadvantages of SOM networks it is the necessity of a large amount of data to operate. Sufficient data is a necessary condition for most of the unsupervised learning algorithms to perform well, nevertheless, for SOM networks, depending on the training parameters and as the number of training epochs increase a small cluster may be forgotten in the case of a class imbalanced problem, for example. Another issue is the fact that it may converge to a map containing two similar clusters distant from each other according to the number of trained epochs and learning rate values. In addition, it may not perform well for categorical data as well. Still, a large number of SOM variations are available to address the original SOM's issues.

17.6 Summary

The Self Organizing Map ANN, also called Kohonen map, is a simple, yet, effective clustering algorithm. Different from other neural networks SOM net-

works use competitive learning instead of error correction for training purposes. In comparison to other clustering algorithms, it does not require much *apriori* knowledge of the data, such as the number of clusters or the cluster's density, to operate.

Given the arrangement presented in the topological map it is intuitive the use of SOM networks, in addition to clustering, also for dimensionality reduction in the data. Therefore, many of the applications of SOM networks involve data representation. This chapter has shown some popular SOM applications, however, as the number of SOM variants grow, the number of problems tackled by this family of algorithms multiplies.

17.7 Exercises

All resources necessary for the exact reproduction of the experiments in the exercises below are provided in a python notebook available at: `https://colab.research.google.com/drive/1_1ukwihWCfgK7288sX2FH9zLAot_s-ft?usp=sharing`. Alternatively, you can also access them at the code folder within the folder of this chapter of the book in its git repo: `https://github.com/ieee-cis/IEEE-CIS-Open-Access-Book-Volume-1`.

1. Given the dataset of Figure 17.4, instantiate a (2,2) topological map and execute one training epoch without considering the neighborhood function.
2. For the training set of Figure 17.4 and a (10,10) topological map compute:
   a. The neighborhood function for each neighbor of the winner neuron of the first training example given the covariance matrix provided in Section 17.2.1.
   b. Change the values of the covariance matrix and compute again the neighborhood function for each neighbor of the winner neuron of the first training example.
   c. Plot the bidimensional Gaussian function generated by each examined covariance matrix.
3. Change the provided code for the example in Section 17.2.1 to handle a dataset containing 1000 examples of different shades of colors red, green and blue. Change also the map to a 50 x 50 map. Introduce the use of parameters τ_1 and τ_2 by changing the parameter of the neighborhood function and the number of epochs such that the algorithm perform a refinement at later eopchs.
4. Explain the main differences between the clustering algorithms k-means and SOM.

References

1. Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.
2. T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.
3. Simon S. Haykin. *Neural networks and learning machines*. Pearson Education, 2009.
4. D. Alahakoon, S. K. Halgamuge, and B. Srinivasan. Dynamic self-organizing maps with controlled growth for knowledge discovery. *IEEE Transactions on Neural Networks*, 11(3):601–614, 2000.
5. Bálint Hegymegi-Barakonyi, László Orfi, Gyorgy Kéri, and István Kovesdi. [application of kohonen self-organizing feature maps in qsar of human admet and kinase data sets]. *Acta pharmaceutica Hungarica*, 83(4):143–8, 2013.
6. Dian Pratiwi. The use of self organizing map method and feature selection in image database classification system. *ArXiv*, abs/1206.0104, 2012.
7. Yakup Kutlu and D. Kuntalp. Feature reduction method using self organizing maps. In *ELECO 2009 - 6th International Conference on Electrical and Electronics Engineering*, pages II–129, 12 2009.
8. Yonggang Liu, Robert Weisberg, and Christopher Mooers. Performance evaluation of the self-organizing map for feature extraction. *Journal of Geophysical Research*, 111, 05 2006.
9. R.M. Satava and S.B. Jones. Current and future applications of virtual reality for medicine. *Proceedings of the IEEE*, 86(3):484–489, 1998.
10. F. Bernardini, H. Rushmeier, I.M. Martin, J. Mittleman, and G. Taubin. Building a digital model of michelangelo's florentine pieta. *IEEE Computer Graphics and Applications*, 22(1):59–67, 2002.
11. Fausto Bernardini, Chandrajit Bajaj, Jindong Chen, and Daniel Schikore. Automatic reconstruction of 3d cad models from digital scans. *International Journal of Computational Geometry & Applications*, 09, 1998.
12. Renata L. M. E. do Rego, Aluizio F. R. Araujo, and Fernando B. de Lima Neto. Growing self-organizing maps for surface reconstruction from unstructured point clouds. In *2007 International Joint Conference on Neural Networks*, pages 1900–1905, 2007.
13. Yizhou Yu. Surface reconstruction from unorganized points using self-organizing neural networks. In *IN IEEE VISUALIZATION 99, CONFERENCE PROCEEDINGS*, pages 61–64, 1999.
14. Ad.M.B. Junior, A.D.D. Neto, and J.D. de Melo. Surface reconstruction using neural networks and adaptive geometry meshes. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, volume 1, pages 803–807, 2004.
15. Dongxiang Chi. Self-organizing map-based color image segmentation with k-means clustering and saliency map. *International Scholarly Research Notices*, 2011:1–18, 2011.
16. N. C. Yeo, K. H. Lee, Y. V. Venkatesh, and Sim Heng Ong. Colour image segmentation using the self-organizing map and adaptive resonance theory. *Image Vis. Comput.*, 23:1060–1079, 2005.
17. Pawel Sanocki, Michal Kawulok, Bogdan Smolka, and Jakub Nalepa. Segmentation of hyperspectral images using self-organizing maps. In *Real-Time Image Processing and Deep Learning 2021*, volume 11736, pages 147 – 153. International Society for Optics and Photonics, SPIE, 2021.
18. A. Ortiz, J. M. Gorriz, J. Ramirez, and D. Salas-Gonzalez. Improving mr brain image segmentation using self-organising maps and entropy-gradient clustering. *Inf. Sci.*, 262:117–136, 2014.
19. Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes

dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

20. Sumith Matharage, Hiran Ganegedara, and Damminda Alahakoon. A scalable and dynamic self-organizing map for clustering large volumes of text data. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1–8, 2013.
21. Emil St. Chifu, Tiberiu St. Letia, and Viorica R. Chifu. Unsupervised aspect level sentiment analysis using self-organizing maps. In *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 468–475, 2015.
22. Muthu Raj, ROSNA HAROON, and N SOBHANA. A novel extractive text summarization system with self-organizing map clustering and entity recognition. *Sādhanā*, 45, 2020.
23. Yuan-Chao Liu, Ming Liu, and Xiao-Long Wang. Application of self-organizing maps in text clustering: A review. In Magnus Johnsson, editor, *Applications of Self-Organizing Maps*, chapter 10. IntechOpen, 2012.
24. R. L. K. Venkateswarlu and R. Vasantha Kumari. Novel approach for speech recognition by using self — organized maps. In *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, pages 215–222, 2011.
25. Yujun Wang and Hugo Van hamme. Gaussian selection using self-organizing map for automatic speech recognition. In *Proceedings of the 8th International Conference on Advances in Self-Organizing Maps*, page 218–227. Springer-Verlag, 2011.
26. S. Lokesh, Priyan Malarvizhi Kumar, M. Ramya Devi, P. Parthasarathy, and C. Gokulnath. An automatic tamil speech recognition system by using bidirectional recurrent neural network with self-organizing map. *Neural Computing and Applications*, 31(5):1521–1531, 2019.

Chapter 18
Clustering Evaluation

Valmir Macario

The evaluation of unsupervised algorithms, such as clustering methods, is aimed at validating the data structures (in the feature space) found by these algorithms. If the purpose is to validate newly developed algorithms, i.e., comparing them with others in the state of the art, it can be useful to compare the obtained results against previously known structures, and, then, assessing the capacity of these algorithms to find a structure close to the one already known [1]. However, in some cases a previously known structure may not be available. So it is necessary to validate whether the found structure met the expected criteria for optimal clustering structures. Some aspects that are inherent of an optimal scheme are the two proposed criteria for clustering evaluation and selection [2]:

- Compactness: the members of the same cluster should be as close to each other as possible. A common measure of compactness is the variance, which must be minimized.
- Separation: different clusters must be be widely spaced. There are three common approaches measuring the distance between two different clusters [3]:
  - Single linkage: it measures the distance between the closest members of the clusters.
  - Complete linkage: it measures the distance between the most distant members.
  - Comparison of centroids: it measures the distance among the centers of the clusters.

These peculiarities make the clustering validation a complex task, once there is no a known solution to compare the results obtained by the algorithms, and often there is no a single solution.

The validation of a clustering performance is, in general, based on statistical measures, which evaluate in a quantitative and objective perspectives the found structures [4]. The index that is used to validate structures is called

validation criterion. Cluster evaluation criteria strategies are usually divided in three types:

- Relative criteria: evaluate clusters based on a criterion that assesses whether the performance yields good clustering partitions according to some assumptions. They can be used to compare multiple clustering algorithms or to determine the most appropriate value for one or more parameters of an algorithm, such as the number of clusters.
- Internal criteria: only use descriptive characteristics of the original data to validate the quality of a cluster.
- External criteria: evaluate the quality of a clustering performance according to previously known structures. These structures are usually labeled datasets, which are used to assess the juxtaposition among the clusters generated by the algorithm and the previously labeled structure.

These validation criteria can be used to evaluate several types of clustering structures such as hierarchies, partitions (hard or fuzzy) and individual clusters.

Usually a relative criterion is used to compare multiple clustering performances. The best option is determined by the maximum or minimum values for a specific index, because for some validation criteria a higher value denotes a best clustering performance, while for other validation criteria the minimum value is related to the best clustering performance.

The external and internal validation criteria are based on statistical tests and have a high computational cost. [3]. The basic idea is to test whether the examples of a dataset are randomly structured or not. This analysis is based on the Null Hypothesis, $H0$, expressed as a statement of random structure of a dataset. To test this hypothesis, the Monte Carlo techniques are often used as a solution [5]. However, it is very difficult to set thresholds for deciding whether the index value is large or small enough to consider the resulting clusters potentially useful or valid [1]. So, a number of validity indices have been defined and proposed in the literature for each of above approaches [4, 6, 7, 5]. In the next section some classic relative, external and internal criteria index are presented.

18.1 Relative Criteria

The relative criteria evaluate the partitions π according to a scheme of a set of defined schemes according to a pre-specified criterion. The best clustering is selected based on the capability of a relative measure, like compactness and separation of the partitions, to mirror the behavior of the external index and properly distinguish among better and worse partitions [8].

18.1.1 Mathematical Notation

Let $\mathcal{T} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(i)}, \ldots, \mathbf{x}^{(n)}\}$ be the dataset which consists of n observed examples of the form $\mathbf{x}^{(i)} = \{x_1^{(i)}, x_2^{(i)}, \ldots, x_p^{(i)}\}$ where p stands as the number of features of the problem. Be $\pi^e = \{\mathbf{c}^{(1)}, \mathbf{c}^{(2)}, \ldots, \mathbf{c}^{(k)}\}$ (e.g., the set of resulting clusters' representations) the clustering algorithm partition result, and $\pi^r = \{C_1, C_2, \ldots, C_k\}$ the ground truth on the clustering structure (i.e., C_i comprises all examples in the cluster i). The cluster representation, that can be a centroid (i.e., the mean of the examples in a cluster) or a medoid (i.e., the example in $\mathcal{T}$ nearest to the centroid) of cluster k is $\mathbf{c}^{(k)}$. In addition, it is also necessary to define a similarity or dissimilarity measure between two examples $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$, i.e., $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. Typically the similarity measure used for most methods is the Euclidean distance (Equation 18.1):

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sqrt{(x_1^{(i)} - x_1^{(j)})^2 + (x_2^{(i)} - x_2^{(j)})^2 + \cdots + (x_p^{(i)} - x_p^{(j)})^2} \quad (18.1)$$

18.1.2 Intracluster Variance

The intracluster variance is given by Equation (18.2), where $\mathbf{c}^{(k)}$ is the centroid of the cluster C_k [9]. This measure assesses the quality of the clusters based on the compression among them. The output values are in the range $[0, \infty]$ so that the smaller the value of var, the better the partition.

$$var(\pi) = \sqrt{\frac{1}{n} \sum_{\mathbf{c}^{(k)} \in \pi^e} \sum_{\mathbf{x}^{(i)} \in C_k} d(\mathbf{x}^{(i)}, \mathbf{c}^{(k)})} \quad (18.2)$$

18.1.3 Connectivity

Connectivity reflects the degree to which neighboring examples are placed in the same cluster. Connectivity is given by Equation 18.3, where v is the number of closest neighbors that contribute to connectivity and nn_{ij} is the $\mathbf{j}^{(th)}$ closest neighbor to the example $\mathbf{x}^{(i)}$. The smaller the *con* index value, the better the partition. The values of this index vary from $[0, \infty]$.

$$con(\pi) = \sum_{\mathbf{x}^{(i)} \in \mathcal{T}} \sum_{j=1}^{v} f(\mathbf{x}^{(i)}, nn_{ij}) \quad (18.3)$$

$$f(\mathbf{x}^{(i)}, nn_{ij}) = \begin{cases} \frac{1}{j} & if\ \mathbf{x}^{(i)} \in \mathbf{c}^{(k)},\ nn_{ij} \notin \mathbf{c}^{(k)} \\ 0 & if\ not \end{cases} \quad (18.4)$$

18.1.4 Dunn Index

The Dunn's index [7] goal is to identify compact and separate clusters. Thus, in the best scenario, the distance between two clusters is large and their diameters are small. It is a function of the distance between the clusters C_a and C_b so that $d(C_a)$ measures the internal distance dispersion of cluster C_a.

$$D(\pi) = \min_{a=1,\ldots,k} \left\{ \min_{b=a+1,\ldots,k} \left\{ \frac{d(C_a, C_b)}{\max_{l=1,\ldots,k} d(C_l)} \right\} \right\} \tag{18.5}$$

$$d(C_a, C_b) = \min_{\mathbf{x}^{(i)} \in Ca, \mathbf{x}^{(j)} \in C_b} d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{18.6}$$

$$d(C_a) = \max_{\mathbf{x}^{(i)}, \mathbf{x}^{(j)} \in C_a} d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{18.7}$$

The Dunn's index is not influenced by the number of clusters. By evaluating the number of clusters from $x \geq 2$, the highest value of Dunn's index can indicate the optimal number of clusters, i.e., the number of clusters that better represents the data.

The problem with the Dunn's index is twofold: (i) its complexity and (ii) its sensitivity to noise examples which can cause an increase in the diameter of the clusters. To tackle its sensitivity to noise there are some variations of the index that mainly change the distances calculated in Equations 18.6 and 18.7 [10].

18.1.5 Silhouette

The silhouette value [11] was pointed out in [12] as the most suitable relative index according to their experiments. Higher values of the silhouette indicates compact and separated clusters.

$$sil(\mathbf{x}^{(i)}) = \begin{cases} 1 - \frac{a(\mathbf{x}^{(i)}, C_i)}{b(\mathbf{x}^{(i)})}, \\ 0, \\ \frac{b(\mathbf{x}^{(i)})}{a(\mathbf{x}^{(i)}, C_i) - 1} \end{cases} \tag{18.8}$$

$$a(\mathbf{x}^{(i)}, C_i) = \frac{1}{|C_k|} \sum_{\mathbf{x}^{(i)}, \mathbf{x}^{(j)} \in C_k, \mathbf{x}^{(i)} \neq \mathbf{x}^{(j)}} d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{18.9}$$

$$b(\mathbf{x}^{(i)}) = \min_{\mathbf{x}^{(i)} \in C_i, C_i \neq C_j} a(\mathbf{x}^{(i)}, C_j) \tag{18.10}$$

in which $a(\mathbf{x}^{(i)}, C_k)$ is the average distance between $\mathbf{x}^{(i)} \in C_k$ and the remaining examples in C_k and $b(\mathbf{x}^{(i)})$ is the minimum of the average distances between $\mathbf{x}^{(i)}$ and examples belonging to different clusters. The value of $sil(\mathbf{x}^{(i)})$

ranges from -1 to $+1$. If the value is close to -1, the example $\mathbf{x}^{(i)}$ is closer, on average, to a cluster C_j such that $\mathbf{x}^{(i)} \notin C_j$. If the value is close to $+1$, then it means that average distance of example $\mathbf{x}^{(i)}$ to the cluster it belongs to is smaller than to any different cluster.

Besides the silhouette of each example, the silhouette of each cluster can be computed according to Equation 18.11. One way to choose the best value of k is to select the value that yields the greatest value of $sil(\pi)$.

$$sil(C_k) = \frac{1}{|C_k|} \sum_{\mathbf{x}^{(i)} \in C_k} sil(\mathbf{x}^{(i)}) \tag{18.11}$$

$$sil(\pi) = \frac{1}{n} \sum_{i=1}^{n} sil(\mathbf{x}^{(i)}) \tag{18.12}$$

The Silhouette Coefficient (SC), is the maximum $sil(\pi)$ for π generated with $k = 2, 3, \ldots, (n-1)$. SC is a measure that quantifies the structure discovered by a clustering algorithm. A value close to 0 means that no substantial structure was found. Less than 0.5 indicates that the structure is weak, between 0.5 and 0.7 indicates a reasonable structure and greater than 0.7 indicates a strong structure [1].

It is worth to notice the existence of another type of clustering algorithms that create fuzzy partitions of the dataset. The fuzzy clustering algorithms yields a pertinence degree u_{ik} of each example to each cluster, rather than being associated with just one cluster as in a hard partition. There are several indexes used for fuzzy clustering algorithms: partition coefficient (PC) [13] , partition entropy (PE) [13], Xie-Beni index (XB) [13] extended Xie-Beni index [13], Fukuyama-Sugeno (FS) [13] and others [13, 14].

18.2 Internal Criteria

Internal criteria indexes measure the match of a cluster obtained by the algorithm to the clustering structure without having access to external information about the ground truth on the clustering structure (see next Subsection 18.3). In general, these criteria are used to select the best number of clusters k. Normally, two types of internal validation metrics can be combined: cohesion and separation measures. Cohesion is an intra-cluster measure that evaluates how closely the examples of the same cluster are to each other, while separation is an inter-cluster measure that quantifies the level of separation between clusters [15]. Cohesion can be computed by Equation 18.13 and separation by Equation 18.14.

$$Cohesion(C_k) = \sum_{\mathbf{x}^{(i)} \in C_k} \sum_{\mathbf{x}^{(j)} \in C_k} d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{18.13}$$

$$Separation(C_k, C_l) = \sum_{\mathbf{c}^{(i)} \in C_k} \sum_{\mathbf{c}^{(j)} \in C_l} d(\mathbf{c}^{(i)}, \mathbf{c}^{(j)}) \tag{18.14}$$

These metrics can also be defined for prototype-based clustering techniques, where the similarities from data examples to cluster centroids or medoids are measured. When the similarity function is the squared Euclidean distance, the cohesion metric defined above is equivalent to the cluster SSE (Sum of Squared Errors); also known as SSW (Sum of Squared Errors Within Cluster) [15].

$$SSE = \sum_{\mathbf{x}^{(i)} \in C_k} d(\mathbf{x}^{(i)}, \mathbf{c}^{(k)})^2 = \frac{1}{2m_i} \sum_{\mathbf{x}^{(i)} \in C_k} \sum_{\mathbf{x}^{(j)} \in C_k} d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})^2 \tag{18.15}$$

There are some difficulties when using these measures, like the SSE that presents lower values for data that actually have clusters than for random, unstructured data. Another difficulty is the dependence of these indexes on the values used for data characteristics, such as number of objects, number of dimensions, number of clusters and scattering [4]. To alleviate some of these limitations, the relative indexes presented in Section 18.1 also can be used as internal criteria. Besides the relative indexes, you can use some others indexes, like the Gap Statitical [16] and Clest proceeding [17] to compute internal criteria.

18.3 External Index

External validation metrics compare the resulting partitions (π^e) achieved by a clustering method and independent partitions created based on prior knowledge about the real data structure (π^r), usually performed on labeled data. Since unsupervised learning techniques are primarily used when class labels are not available, external validation methods can't be applied on most of the clustering problems. However, they can still be applied when external information is available (e.g., for synthetic data)[15].

An external index for partitions suitability evaluates the degree to which two partitions of n examples match. In order to carry out this analysis, a contingency matrix must be built to evaluate the clusters found by the algorithm. Given a pair of examples $(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, the contingency matrix can depict four possible situations:

- True Positive (TP): they belong to the same cluster, both in π^e and in π^r.
- False Positive (FP): they belong to the same cluster in π^e but belong to different clusters in π^r.

- False Negative (FN): they belong to different clusters in π^e but belong to the same cluster in π^r.
- True Negative (TN): they belong to different clusters, both in π^e and π^r.

From these four situations, we can easily obtain:

- The number of pairs belonging to the same cluster in π^e: $m1 = TP+FP$.
- The number of pairs belonging to the same cluster in π^r: $m2 = TP+FN$.
- The total number of pairs of examples: $M = TP + FP + FN + TN = \frac{n(n-1)}{2}$

These values can then be used to compute a number of external indexes, such as Hubbert, Jaccard, Rand and Corrected Rand [4] that will be described in next subsections.

18.3.1 Rand Index

The Rand index, computed by Equation 18.16, measures the probability of two examples to belong to same cluster or to belong to different clusters on both partitions π^e and π^r.

$$Rand = \frac{TP + TN}{M} \tag{18.16}$$

18.3.2 Jaccard Index

The Jaccard index computes the probability of two examples that belong to same cluster on one of the partitions also belong to the same cluster on the other partition. The Jaccard index is defined by Equation 18.3.2.

$$J = \frac{TP}{(TP + FP + FN)} \tag{18.17}$$

18.3.3 Folkes and Mallows Index

The Folkes and Mallows index computes the similarity among the clusters found by the algorithm with respect to the a ground-truth. This index ranges between 0 and 1 and larger values indicate more similarity between two partitions. This index is computed by Equation 18.18.

$$FM = \sqrt{\frac{TP}{(TP + FP)} \times \frac{TP}{(TP + FN)}} \tag{18.18}$$

18.3.4 Normalized Hubert Index

The Normalized Hubert Index measures the linear correspondence between two partitions. The values are in the range [-1,1] with bigger values indicating that the two partitions have bigger correspondence. Equation 18.3.4 defines the Normalized Hubert Index.

$$\Gamma = \frac{(M \times TP) - (m_1 m_2)}{\sqrt{m_1 m_2 (M - m_1)(M - m_2)}} \tag{18.19}$$

18.3.5 Corrected Rand Index

A common characteristic for most of external indexes is their high sensitivity to the number of classes in partitions or to the spatial distribution of examples in clusters. For example, some indexes tend to output higher values for partitions with more classes (Hubbert and Rand), others for partitions with fewer classes (Jaccard) [18]. The Corrected Rand Index (CR) has its outputs corrected according to correct answers in the comparisons of the partitions, so it does not have any of these undesirable characteristics [19]. This index can be obtained from the contingency table and built from the two partitions shown in Table 18.1. The CR can be defined by Equation 18.20:

Table 18.1: Contingency table of two partitions

	C_1	C_2	$\ldots$	C_K	
Q_1	n_{11}	n_{12}	...	n_{1K}	$n_{1.}$
Q_2	n_{21}	n_{22}	...	n_{2K}	$n_{2.}$
...	...	...		...	
Q_H	n_{H1}	n_{H2}	...	n_{HK}	$n_{R.}$
	$n_{.1}$	$n_{.2}$		$n_{.K}$	$n_{..} = n$

$$CR = \frac{\sum_h^H \sum_k^K \binom{n_{hk}}{2} - \binom{n}{2}^{-1} \sum_h^H \binom{n_{h\cdot}}{2} \sum_k^K \binom{n_{\cdot k}}{2}}{\frac{1}{2}\left[\sum_h^H \binom{n_{h\cdot}}{2} + \sum_k^K \binom{n_{\cdot k}}{2}\right] - \binom{n}{2}^{-1} \sum_h^H \binom{n_{h\cdot}}{2} \sum_k^K \binom{n_{\cdot k}}{2}} \tag{18.20}$$

where $\binom{n}{2} = \frac{n(n-1)}{2}$ and n_{hk} is the number of examples belonging to the same cluster for both partitions π_k^e and π_h^r. The term $n_{\cdot k}$ depicts the number of examples in cluster C_k and $n_{h\cdot}$ is the number of examples at class Q_h, and n is the total number of examples.

This index varies in the range [-1, 1], where 1 indicates perfect cohesion between the partitions and -1 indicates that there is no cohesion between the

partitions. Milligan and Cooper's work [19] indicates that values below 0.05 are randomly generated partitions.

18.3.6 Information Variation Index

The information variation (VI) [1] is computed according to Equation 18.23. It measures the amount of information lost or gained to move from the π^e partition to the π^r partition. In this equation, $EN(\pi^a)$ computes the entropy of the partition π^a using Equation 18.22 and the mutual information shared between two partitions is given by the Equation 18.21. The probability of an object share both partitions π^e and π^r is given by $p(k,h) = \frac{n_{hk}}{n}$ and the probability of an object belong to cluster C_k is $p(k) = \frac{n_{\cdot k}}{n}$. This index has zero as the best value, indicating that the two partitions are identical and have no upper limit.

$$I(\pi^a, \pi^b) = \sum_{k=1}^{K}\sum_{h=1}^{H} p(k,h)\log\left(\frac{p(k,h)}{p(k)p(h)}\right) \tag{18.21}$$

$$EN(\pi^a) = -\sum_{k=1}^{K} p(k)\log p(k) \tag{18.22}$$

$$VI(\pi^e, \pi^r) = EN(\pi^e) + EN(\pi^r) - 2I(\pi^r, \pi^r) \tag{18.23}$$

18.3.7 Normalized Mutual Information Index

The external mutual information index (I) originates from the principles of information theory and the notion of entropy [20]. When entropy is applied to the clustering process, it is a metric that measures the uncertainty that an element $\mathbf{x}^{(i)}$ will be associated with a certain cluster C_k. For the case that an element is associated with a cluster by chance, the entropy will be 0. The notion of entropy can be extended to mutual information, which measures how much we can, on average, reduce the uncertainty about the random choice that an element $\mathbf{x}^{(i)}$ is associated with a C_k cluster, taking into account that its cluster is known from another previous clustering. The mutual information is calculated using the Equation (18.21).

The mutual information I is a metric about the space of all clusters. However, it is not limited by a constant value, which makes it difficult to interpret. In the work by Strehl and Ghosh [21], a normalization by geometric mean was proposed. To this end, they determine the cluster that has the maximum average of the normalized mutual information of all the clusters under consideration, where the normalized mutual information (NMI) between two

clusters is defined as follows by the Equation 18.24.

$$NMI(\pi^e, \pi^r) = \frac{I(\pi^r, \pi^e)}{\sqrt{EN(\pi^r)EN(\pi^e)}} \tag{18.24}$$

where $I(\pi^r, \pi^e)$ is the mutual information between π^r and π^e. $EN(\pi^r)$ and $EN(\pi^e)$ are entropies of π^r e π^r respectively computer with Equation 18.22. NMI has its values between $[0, 1]$, being 1 the best clustering match.

18.3.8 Accuracy Rate

The accuracy (ACC) looks for a decision rule that minimizes the probability of error [22]. The maximum value of a ACC indicates the best performance of the clustering algorithm.

$$ACC = \left(\frac{\sum_{k=1}^{K} \max_{1 \leq h \leq H} \ n_{hk}}{n} \right) \tag{18.25}$$

where n_{hk} is as described.

18.4 Summary and Discussion

In this chapter, strategies and evaluation criteria for clustering algorithms analysis were disclosed. Briefly, the three types of existing criteria for evaluation were detailed: relative, internal and external criteria. The relative criterion is useful to compare several clustering strategies with respect to some aspect and to decide the ideal number of clusters. The internal criteria, evaluate the quality of a cluster according to some property existing in the original data. The relative and internal criteria evaluate cluster quality when there is no previous information about the original data. Yet, the external criteria allows the confrontation of the clustering obtained by an algorithm with a previously known data structure. In addition, several indices that can be applied with each criterion were discussed, different ways in which these indices can be used, as well as several recent approaches that consider aspects such as stability of generated partitions.

18.5 Exercises

1. What is the difference between an external and relative criteria index?
2. Compare Dunn's and Silhouette's indices. Which clustering criteria do they benefit?
3. An expert is analyzing the result of a clustering algorithm on a dataset A. He knows that this set of data has a known structure π^r. In the experiments were obtained the following values: $TP = 92$, $FP = 8$, $FN = 12$ and $TN = 88$. Compute the external indices:

   a. Rand Index
   b. Jaccard Index
   c. Folkes and Mallows Index
   d. Normalized Hubert Index.

4. A comparison with 2 clustering algorithms has obtained the best partitions of each algorithm (π^1, π^2), respectively. Computing the corrected rand index for each partition were obtained the following values: $CR(\pi^1, \pi^r) = 0,9$ and $CR(\pi^2, \pi^r) = 0,2$. What you can say about each of partitions?

References

1. Katti Faceli, Ana C. Lorena, , J Gama, Tiago A. de Almeida, and André C. P. L. F. de Carvalho. *Inteligência Artificial–uma abordagem de aprendizado de máquina*. 2021.
2. M.J.A. Berry and G. Linoff. Data mining techniques for marketing, sales and customer support. *John Wiley and Sons*, 1996.
3. Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of intelligent information systems*, 17(2):107–145, 2001.
4. Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.
5. Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Fourth Edition*. Academic Press, 2009.
6. Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. Cluster validity methods: part i. *ACM Sigmod Record*, 31(2):40–45, 2002.
7. Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. Clustering validity checking methods: Part ii. *ACM Sigmod Record*, 31(3):19–27, 2002.
8. Lucas Vendramin, Ricardo JGB Campello, and Eduardo R Hruschka. Relative clustering validity criteria: A comparative overview. *Statistical analysis and data mining: the ASA data science journal*, 3(4):209–235, 2010.
9. Julia Handl, Joshua Knowles, and Douglas B Kell. Computational cluster validation in post-genomic data analysis. *Bioinformatics*, 21(15):3201–3212, 2005.
10. James C Bezdek and Nikhil R Pal. Some new indexes of cluster validity. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 28(3):301–315, 1998.
11. Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.
12. Marcel Brun, Chao Sima, Jianping Hua, James Lowey, Brent Carroll, Edward Suh, and Edward R. Dougherty. Model-based evaluation of clustering validation measures. *Pattern Recognition*, 40(2007):807–824, 2007.

13. Nikhil R Pal and James C Bezdek. On cluster validity for the fuzzy c-means model. *IEEE Transactions on Fuzzy systems*, 3(3):370–379, 1995.
14. Malay K Pakhira, Sanghamitra Bandyopadhyay, and Ujjwal Maulik. Validity index for crisp and fuzzy clusters. *Pattern recognition*, 37(3):487–501, 2004.
15. Julio-Omar Palacio-Niño and Fernando Berzal. Evaluation metrics for unsupervised learning algorithms. *arXiv preprint arXiv:1905.05667*, 2019.
16. Robert Tibshirani, Guenther Walther, and Trevor Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, 2001.
17. Sandrine Dudoit and Jane Fridlyand. A prediction-based resampling method for estimating the number of clusters in a dataset. *Genome biology*, 3(7):1–21, 2002.
18. R. Dubes. How many clusters are best? an experiment. *Pattern Recognition*, 20(6):645–663, 1987.
19. G. W. Milligan and M. C. Cooper. A study of the comparability of external criteria for hierarchical cluster analysis. *Multivariate Behavorial Research*, 21:441–458, 1986.
20. Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2 edition, 2006.
21. Alexander Strehl and Joydeep Ghosh. Cluster ensembles - a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, (3):583–617, 2002.
22. L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC, Boca Raton, 1984.