

Natural Computing Series

José Raúl Romero
Inmaculada Medina-Bulo
Francisco Chicano *Editors*



Optimising the Software Development Process with Artificial Intelligence

 Springer

Natural Computing Series

Founding Editor

Grzegorz Rozenberg

Series Editors

Thomas Bäck , Natural Computing Group–LIACS, Leiden University, Leiden, The Netherlands

Lila Kari, School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Susan Stepney, Department of Computer Science, University of York, York, UK

Scope

The Natural Computing book series covers theory, experiment, and implementations at the intersection of computation and natural systems. This includes:

- **Computation inspired by Nature:** Paradigms, algorithms, and theories inspired by natural phenomena. Examples include cellular automata, simulated annealing, neural computation, evolutionary computation, swarm intelligence, and membrane computing.
- **Computing using Nature-inspired novel substrates:** Examples include biomolecular (DNA) computing, quantum computing, chemical computing, synthetic biology, soft robotics, and artificial life.
- **Computational analysis of Nature:** Understanding nature through a computational lens. Examples include systems biology, computational neuroscience, quantum information processing.

José Raúl Romero · Inmaculada Medina-Bulo ·
Francisco Chicano
Editors

Optimising the Software Development Process with Artificial Intelligence



Springer

Editors

José Raúl Romero  Department of Computer Science and Numerical Analysis University of Córdoba Córdoba, Córdoba, Spain

Inmaculada Medina-Bulo  Department of Computer Science and Engineering University of Cádiz Puerto Real, Cádiz, Spain

Francisco Chicano  ITIS Software Department of Languages and Computing Sciences University of Málaga Málaga, Málaga, Spain

ISSN 1619-7127

Natural Computing Series

ISBN 978-981-19-9947-5

ISBN 978-981-19-9948-2 (eBook)

<https://doi.org/10.1007/978-981-19-9948-2>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd. The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Foreword

Optimisation has really taken off as an area of research, in general, not just in the software engineering community, since the 1940s. However, we can trace the origins of software optimisation right back to the very birth of software itself, with Ada Lovelace's prescient comments on the nature of software and the need for its optimisation. That is, 180 years ago, she wrote:

In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

This is an extract from “Note D” of “Sketch of the Analytical Engine Invented by Charles Babbage” by Ada Augusta Lovelace, 1842. It is surely the first instance of any author writing about the need to improve software execution performance by optimising the expression of the computation itself: optimising the program.

The application of meta heuristic search to software optimisation dates back to the 1970s, but has really taken off in the last two decades with the advent of the field that has come to be known as “Search-Based Software Engineering” (SBSE). The interest in optimisation techniques and evolutionary computation as a vehicle for softer optimisation rendered the software engineering community highly receptive to artificial intelligence techniques more generally. Recent breakthroughs in machine learning have further stimulated software engineers’ interest in artificial intelligence in all of its forms and many applications.

Artificial intelligence techniques have been used to optimise almost every kind of engineering artefact and the processes by which they are made, right across the spectrum of engineering disciplines, including mechanical, biological, chemical, and even social engineering disciplines. Nevertheless, it is in software engineering that these techniques find a uniquely well-fitted potential in their application: Not only is the engineering artefact optimised using AI techniques but also the AI techniques are, themselves, implemented in the same engineering material—software. This opens up possibilities for self-application, and, with that, the potential for continuous, dynamic adaptive optimisation in deployed software environments.

I firmly believe that a deeper understanding of the nature of optimisation and its applications to software engineering will yield many more insights into the nature of optimisation itself and also, in so doing, will shed further light on the specific properties of software as an exciting (arguably the most exciting, and certainly the most peculiar) engineering material with which our species has yet worked.

The foundational nature of optimisation has led not only to an explosion of research on Search-Based Software Engineering but also to widespread uptake of computational search as a practical and flexible tool for software engineering optimisation. Most notably, techniques associated with software testing and repair have found their way into widespread industrial practice in many companies, both large and small. This impact has been felt by almost every user of the Internet, through applications at companies such as Meta, Google, and Microsoft. Currently, approximately 2.9 billion people are running software (consisting of tens of millions of lines of code) on smartphones. They are running software that has been automatically fixed by computational search techniques that automatically found fixes for bugs that were automatically detected by test cases that were, themselves, automatically designed by computational search.

This is but one recent example of the application of search-based software engineering to testing and bug fixing, and we can be sure that this is just the beginning. We are witnessing the early development of a fundamentally new approach to software engineering in which artificial intelligence and human intelligence combine to optimise software systems and the processes by which they are produced. Our grandchildren will surely consider the idea that the word “programmer” refers exclusively to a human as being just as quaint and anachronistic as we, today, would consider the view of our nineteenth-century engineering forebearers, who thought that the word “computer” would naturally and solely only ever refer to a human.

Much of the overall body of work on software optimisation has focused on software products, such as optimisation of the code itself, and also documentation, design diagrams, and test suites. However, optimisation has also been applied to software engineering processes. It is this set of related application areas that forms the concern of this excellent compendium of work on optimising software development processes with artificial intelligence.

In this single volume, the reader will find many aspects of the overall software development life cycle tackled using optimisation, from the elicitation of requirements and systems modelling through to development and on to deployment (e.g. cloud-based deployment) and on-going maintenance. These are the processes of most interest and importance to practising software engineers, and form the foundations for multiple research disciplines within the overall body of work on software engineering. The reader will also find work on optimising the overall management of processes and on developer productivity assistance (such as code completion) as well as work on software testing.

The chapters in this book have been written by leading researchers in the field of software process optimisation using AI techniques. The collection into a single volume provides an overview and introduction to an exciting field of engineering optimisation, with many practical applications and challenges for further research.

To further support practitioners and researchers from the software engineering community, seeking to deploy these techniques, the final chapters in this book provide foundational tutorial work on meta heuristics and machine learning. This ensures that the book is useful for researchers and practitioners alike. The interested reader, who wants to follow up further having read the book, will also find many other surveys and tutorials available online.

London, UK
July 2022

Mark Harman

Preface

A software development project is complex, and poses constant challenges to the professionals who supervise, plan, design, analyse, develop, and maintain it. When we think of a software project, we must think of all its phases and activities: both in those phases referring to the planning and control of the engineering project, and in the phases of the development process itself, from the specification of the requirements and the architectural design to the testing, refactoring, and maintenance of the software.

In recent years, there has been a clear interest in automating and providing support to the professionals, with the goal of reducing the effort, time, cost, and risk of the different phases of a software project. To this end, initiatives related to the use of artificial intelligence (AI) for optimising these tasks have been tested for decades, from expert systems to search and optimisation techniques to machine learning. The editors of this volume have been working for more than 20 years in software engineering (SE), including the application of AI techniques for the optimisation of the different phases of the software life cycle. And we must recognise that during this time great advances have been made, but the most important comes from the industry itself, which acknowledges the need to apply techniques that improve its software engineering process and demands solutions for it. AI-enhanced software engineering or AI4SE (artificial intelligence for software engineering) was born.

An important factor is the popularisation of general-purpose tools among the developer community based on these techniques. Examples of these applications are the GitHub Copilot intelligent assistant, which had a great impact on the community by making visible and tangible the pros and cons of using AI for software development assistance. Other development assistant tools such as DeepMind AlphaCode or OpenAI Codex have also attracted the attention of professionals in this field—a field that had already been generating assistant tools in academia for some time, such as EvoSuite for the automatic generation of test suites.

The discussions generated about these tools, their internal foundations, their scope, and practicability, as well as the near future of AI4SE research and development have motivated the need for a volume like this. With this book, we aim to

provide Information Technology (IT) professionals (practitioners, developers, software engineers, or managers) as well as advanced graduate and Ph.D. students with a practical introduction to the use of AI techniques to improve and/or optimise the different phases of the software development process, from the initial project planning to the latest deployment. Notice that AI is a broad term, and the book is intended to cover it from different perspectives: optimisation and search algorithms applied to software engineering (also known as Search-Based Software Engineering, SBSE), machine learning, and pattern mining in software analytics, mining software repositories (MSR), natural language processing (NLP), etc. The AI solutions for SE used throughout the book are explained in a didactic way to provide the reader with a sufficient basis for a complete understanding of its content.

For producing this contributed volume, we have relied on renowned authors, highly experienced in each of the areas of the book. We first divided the project according to the classic phases of software project development and studied how AI had been used for the optimisation and improvement of each of them. The authors were invited not only to write descriptively about the state of AI4SE in their specific domain but also to describe real use cases and to develop practical and reproducible examples to enable the reader to understand, execute, and/or modify practical case studies. Most of the chapters come with a reproducibility package composed of source code and datasets for the reader to experience the techniques described in the chapters. A snapshot of the reproducibility packages has been uploaded to Zenodo as complementary material for this volume. But the chapters also provide links to a live version of the packages in GitHub, where the interested reader could find new updates of the software tools presented in this book. Finally, all chapters were reviewed by experts from industry and academia and were double-checked by the editors. The result is a homogeneous book that, despite dealing with complex problems in each topic, allows the reader to go deeper into the areas of interest from the ground up. The two final chapters of the volume cover the necessary background on artificial intelligence techniques for understanding the rest of the text.

Córdoba, Spain
Cádiz, Spain
Málaga, Spain
October 2022

José Raúl Romero
Inmaculada Medina-Bulo
Francisco Chicano

Acknowledgment

The editors would like to thank the qualified experts in AI4SE who have contributed to this volume. Their view opens a unique window to the readers through which they can observe future trends in the application of AI to the different problems that arise during the software development process.

Special mention should be made to Dr. Mark Harman, Professor at University College London and Research Scientist in a renowned IT company. Dr. Harman is considered one of the fathers of search-based software engineering, and we thank him for his willingness to participate in this project by writing the foreword to this volume.

We would also like to thank the reviewers for their selfless work, who have volunteered their time to provide an even better version of each of the chapters:

Dr. Shaukat Ali

Chief Research Scientist, Simula Research Laboratory, Norway

Dr. Jaume Bacardit

Reader, Newcastle University, United Kingdom

Dr. Juan J. Durillo

Scientific Staff, Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Germany

Mr. Aitor Facio Valero

Project Manager, Navantia, Spain

Dr. Arnaud Liefooghe

Associate Professor, University of Lille, France

Mrs. María Mora

CIO at Municipal Information Technology Center, Málaga, Spain

Dr. Gregorio Robles

Full Professor, Universidad Rey Juan Carlos, Spain

Mr. Rubén Salado
Head of Backend, Genially, Spain

Dr. Christopher L. Simons
Lecturer, University of the West of England, United Kingdom.

The development of this volume has been financially supported by Grant PID2020-115832GB-I00 funded by MICIN/AEI/10.13039/501100011033; and the Spanish Research Network SEBASENet¹ funded by the Spanish Ministry of Science and Innovation (SMSI), and co-funded by the European Regional Development Fund (ERDF), project RED2018-102472-T. This work has also been supported by SMSI and ERDF through the project AwESOMe (PID2021-122215NB-C33), and by the University of Malaga (project B4-2019-05).

¹ SEBASENet Research Network: <https://www.uco.es/SEBASENet/>

Contents

1	Introduction	1
	José Raúl Romero, Inmaculada Medina-Bulo, and Francisco Chicano	
Part I Planning and Analysis		
2	Artificial Intelligence in Software Project Management	19
	Liyian Song and Leandro L. Minku	
3	Requirements Engineering	67
	Fitsum Kifetew, Anna Perini, and Angelo Susi	
4	Leveraging Artificial Intelligence for Model-based Software Analysis and Design	93
	Antonio Garmendia, Dominik Bork, Martin Eisenberg, Thiago Ferreira, Marouane Kessentini, and Manuel Wimmer	
Part II Development and Deployment		
5	Statistical Models and Machine Learning to Advance Code Completion: Are We There Yet?	121
	Tien N. Nguyen	
6	Cloud Development and Deployment	155
	José Antonio Parejo and Ana Belén Sánchez	
Part III Testing and Maintenance		
7	Automated Support for Unit Test Generation	179
	Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveira Neto, and Robert Feldt	
8	Artificial Intelligence Techniques in System Testing	221
	Michael Felderer, Eduard Paul Enoiu, and Sahar Tahvili	

9	Intelligent Software Maintenance	241
	Foutse Khomh, Mohammad Masudur Rahman, and Antoine Barbez	

Part IV AI Techniques from Scratch

10	Metaheuristics in a Nutshell	279
	Javier Ferrer and Pedro Delgado-Pérez	
11	Foundations of Machine Learning for Software Engineering	309
	Aurora Ramírez and Breno Miranda	

Contributors

José Antonio Parejo Smart Computer Systems Research and Engineering Lab (SCORE), Research Institute of Computer Engineering (I3US), Universidad de Sevilla, Seville, Spain

Antoine Barbez Polytechnique Montreal, Montreal, Canada;
Dalhousie University, Halifax, Canada

Dominik Bork TU Wien, Business Informatics Group, Vienna, Austria

Francisco Chicano University of Málaga, Málaga, Spain

Francisco Gomes de Oliveira Neto Chalmers and the University of Gothenburg, Gothenburg, Sweden

Pedro Delgado-Pérez Department of Computer Science and Engineering, University of Cádiz, Cádiz, Spain

Martin Eisenberg JKU Linz, CDL-MINT, Institute of Business Informatics - Software Engineering, Linz, Austria

Eduard Paul Enoiu Mälardalen University, Västerås, Sweden

Michael Felderer University of Innsbruck, Innsbruck, Austria;
Blekinge Institute of Technology, Karlskrona, Sweden

Robert Feldt Chalmers and the University of Gothenburg, Gothenburg, Sweden

Thiago Ferreira University of Michigan-Flint, College of Innovation & Technology, Michigan, USA

Javier Ferrer ITIS Software, University of Malaga, Malaga, Spain

Afonso Fontes Chalmers and the University of Gothenburg, Gothenburg, Sweden

Antonio Garmendia JKU Linz, Institute of Business Informatics - Software Engineering, Linz, Austria

Gregory Gay Chalmers and the University of Gothenburg, Gothenburg, Sweden

Marouane Kessentini Oakland University, School of Engineering and Computer Science, Michigan, USA

Foutse Khomh Polytechnique Montreal, Montreal, Canada

Fitsum Kifetew Fondazione Bruno Kessler, Trento, Povo, Italy

Mohammad Masudur Rahman Dalhousie University, Halifax, Canada

Inmaculada Medina-Bulo University of Cádiz, Cádiz, Spain

Leandro L. Minku School of Computer Science, University of Birmingham, Birmingham, UK

Breno Miranda Informatics Center, Federal University of Pernambuco, Recife, Brazil

Tien N. Nguyen University of Texas at Dallas, Dallas, USA

Anna Perini Fondazione Bruno Kessler, Trento, Povo, Italy

Aurora Ramírez Dept. Computer Science and Numerical Analysis, University of Córdoba, Córdoba, Spain

José Raúl Romero University of Córdoba, Córdoba, Spain

Ana Belén Sánchez Smart Computer Systems Research and Engineering Lab (SCORE), Research Institute of Computer Engineering (I3US), Universidad de Sevilla, Seville, Spain

Liyan Song Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China

Angelo Susi Fondazione Bruno Kessler, Trento, Povo, Italy

Sahar Tahvili Mälardalen University, Västerås, Sweden;
Ericsson AB, Stockholm, Sweden

Manuel Wimmer JKU Linz, CDL-MINT, Institute of Business Informatics - Software Engineering, Linz, Austria

Chapter 1

Introduction



José Raúl Romero, Inmaculada Medina-Bulo, and Francisco Chicano

Abstract Software engineering focuses on the creation of high-quality software products according to basic design principles and best practices. Since its inception in the 1950s, the complexity of software systems, their environment and infrastructure, the associated requirements, and the methods and methodologies used have increased dramatically. This greater complexity of project management brings a significant increase in the associated risk, which is reflected in the search for techniques to mitigate it through automation and optimisation of activities of the software development process, as well as their improvement by learning from previous experiences. In this context, the synergy between software engineering and artificial intelligence has been a revulsive for companies. In this chapter, we introduce the field of AI4SE and explore the problems and application domains where these techniques have proven to work successfully. This volume focuses on these solutions. Therefore, at the end of the chapter, we provide a brief introduction to the organisation of the book and explain very briefly each of the chapters that comprise it.

1.1 Introduction

Software engineering was first coined as a term at an uncertain point in time between the 1950s and 1960s. Some prominent software developers at that time, such as Margaret Hamilton (lead software engineer of the Apollo Project), recognised that the application of techniques, methodologies and processes was essential for their work,

J. R. Romero (✉)
University of Córdoba, Córdoba, Spain
e-mail: jrromero@uco.es

I. Medina-Bulo
University of Cádiz, Cádiz, Spain
e-mail: inmaculada.medina@uca.es

F. Chicano
University of Málaga, Málaga, Spain
e-mail: chicano@uma.es

just as it was done in other engineering disciplines. Since then, software engineering has encompassed a set of skills that enable a professional to design, develop, test, deploy and maintain increasingly complex software products. It was in 1968–1969 when NATO conferences gave significant weight to the term software engineering, and eventually popularised it. Since then, the ultimate goal of software engineering is the creation of high-quality software products. In 1975, Ross et al. [7] first set out the principles and objectives that affect the practice of software engineering. These were a series of basic principles (modularity, abstraction, localisation, hiding, uniformity, completeness, and confirmability), on which they discussed how they should be applied in the practice to reach understandability, reliability, efficiency and modifiability. To achieve this, various software development activities were modulated, such as determining requirements, designing software, specifying implementation, debugging, and tuning (dedicated to achieving performance objectives in these early software systems).

However, we have reached the present day without a consensus on a body of principles for this discipline, unlike in other engineering disciplines. SWEBOK (Software Engineering Body of Knowledge)¹ emerged in 2004 as a guide for the description of the generally accepted knowledge about software engineering. SWEBOK is divided into knowledge areas (KA). In version 4 (draft version at the time of writing this volume) it has been extended to 17, incorporating relevant areas such as software architecture, software operations, and security. From previous versions, there were already KAs more specific for the common activities of any software development cycle, such as those referring to software requirements, design, construction, testing, or maintenance. Also, other KAs refer to activities closer to process and project management, methodological or theoretical aspects that need to be covered to complete any industrial software engineering project: software configuration management, software engineering management, software engineering process, models and methods, software quality, professional practice, economics, computing foundations and mathematical and engineering foundations. This collection of knowledge about the software engineering practice became the ISO/IEC TR 19759:2015 standard, published with the objective to “characterise the boundaries of the software engineering discipline and provide topical access to the literature supporting that discipline”.

Initiatives such as SWEBOK highlight the similarities between software engineering and other engineering disciplines. However, in the absence of fundamental principles, i.e., those that constitute the basis from which all these principles originate, a large number of proposals have emerged that establish different frameworks, methods, techniques and tools. Recently, Al-Sarayreh et al. [1] have published a study in which they compiled up to 592 software engineering principles, published between 1969 and 2020. Nevertheless, despite this impressive number of principles collected, the authors conclude that software engineering is an immature and still evolving discipline. Certainly, progress has been made in the tools used, methods applied and the technological environments that support them. In fact, the evolution

¹ SWEBOK. <https://www.computer.org/education/bodies-of-knowledge/software-engineering> (last accessed: July 27, 2022).

in the complexity of the development process has led to the definition of new principles specifically focused on aspects like, for example, replicating the success of past software projects. It could have a high future impact on the industry but requires the improvement of software engineering processes by enhancing planning, management, and development of software at different phases of the software development life cycle.

Despite the aforementioned increase in complexity, we must not overlook the fact that, while we focus on the development and enforcement of complex software engineering processes, today's marketplace demands the need for speed and quality in daily deliveries of all software artefacts, meeting increasingly demanding project and software requirements. This increasing managing project complexity entails a significant increase of risk in terms of shaping both processes and product outcomes. In fact, many costs of software products are attributed to those current techniques that are ineffective in managing that complexity according to the most up-to-date practices of software engineering. IT project managers know the need to properly manage the complexity associated with the most demanding software projects, applying measures for project planning and control, without underestimating associated risks, learning from past experiences, and trying to optimise the different activities undertaken by the development team.

It is a changing and challenging context, where the availability of optimisation, automated support and improvement mechanisms for software development process activities can provide a competitive advantage to companies. In particular, artificial intelligence (AI) has been a significant advance that the software engineering community has begun to take advantage of in the last decade.

1.2 The Rise of Artificial Intelligence

According to [8], AI “is concerned with not just understanding but also building intelligent entities—machines that can compute how to act effectively and safely in a wide variety of novel situations.” These intelligent agents are founded on the idea that they will be able to act rationally, i.e., in a way that achieves the best outcomes, or at least the best *expected* outcomes. To this end, AI focuses on the study and development of agents that are able to “do the right thing”, that is, to meet the goals we impose on the agent in terms of models. Unfortunately, models are usually limited, as perfect rationality can only be achieved in theoretical analyses. This is evident when the model tries to solve a complex real-world problem. In this case, it is not possible to assume that the agent machine will be able to meet fully specified objectives. For example, in the case of a flight between two cities, it seems obvious that the objective would be for the plane to land at its destination, for which it must perform a series of correct actions: taxi, take-off, fly, and land. However, this is not always the case. There may be weather problems, mechanical breakdowns, incidents with passengers or malicious acts that cause the best solution to be different from

achieving the objective. Moreover, sometimes trying to meet the specified objective may have negative consequences, even if they have been defined as rational.

AI is a discipline that, in one form or another, has been discussed for a long time in intellectual, scientific and, recently, technological forums. Aristotle already conceived of a system for the mechanical simulation of human reasoning that ideally would be able to generate conclusions mechanically. Since then, human interest in simulating their own rational functioning has been present throughout history. The theory of probability developed by mathematicians such as Gerolamo Cardano, Blaise Pascal or Pierre Fermat was an important step in the treatment of uncertainty and incomplete theories. In fact, Bayes' rule for updating probabilities in light of new evidence has been an essential mechanism for the development of AI. The same applies to formal logic, which has been worked on since ancient Greece but whose development by the mathematician George Boole propelled its use as a core element in a long early period of AI research. Meanwhile, advances in computing have facilitated the development of the agent machines needed in AI. Algorithm, compatibility, or tractability have been computational terms that have been developed fundamentally during the twentieth century and that have motivated the need to foster research in intelligent systems, capable of adapting to the resources and speed of computers, despite the growing complexity of the problems to be solved.

In light of the background, if we had to recognise an initial milestone that marks the birth of AI as it is perceived today, it would be the work by Warren McCulloch and Walter Pitts in 1943. For the first time, they were able to propose an artificial neural network model inspired by the functioning of neurons in the brain, Turing's theory of computation and propositional logic. Since then, AI has evolved in parallel with the computational capabilities of machines and has been used in applications of all kinds. Noteworthy are neural networks, which have recently taken a qualitative leap forward with the emergence of deep learning; models inspired by the evolution of species, such as evolutionary computation and genetic programming; and expert systems, which for a long time offered an interesting alternative for solving complex problems, since they allow larger reasoning steps and incorporate domain-specific knowledge that facilitates the resolution of more limited situations.

More recently in history, organisations have been observing the need to value their own data, which is becoming more and more accessible and in greater quantity. Precisely this availability of data has had a significant impact on efforts to focus on statistical analysis, optimisation and machine learning. In addition, fields such as artificial vision, natural language processing or robotics, which were originally developed in silos, have been integrated under the umbrella of AI with great success. All this wide diversity of techniques has led to the current AI research landscape being composed of a broad set of topics and techniques. In fact, notice that new AI methods are constantly appearing and barriers between them are becoming blurry, but the majority of methods still have roots in the long-established areas presented in Russell and Norvig's handbook [8].

Problem-Solving

Problem-solving agents try to find the sequence of actions that form the path to the goal state in an unknown environment. For this purpose, computational search and optimisation processes are carried out, which in general have four phases: the objective formulation, the problem formulation (states and actions to reach the objective), the search (simulation of sequences until the sequence called solution is found), and the execution (execution of the actions on the solution). Optimisation is also about finding the best state according to the objective function. For both search and optimisation, an infinite number of strategies have been proposed, judged on the basis of their completeness, cost optimality, time complexity and space complexity. In this context, metaheuristics are extensively discussed in the chapter “Metaheuristics in a Nutshell” of this volume. These higher-level procedures seek to apply a heuristic that is capable of providing a sufficiently good solution to an optimisation problem, especially under conditions restricted in both information and computational resources.

Machine Learning

This category covers those techniques that analyse vast collections of data to find hidden relationships and make predictions. In general, a computational agent that is able to improve its performance after making observations about the world is called machine learning. In this case, the agent observes data, builds a model according to that data, and finally uses the model to generate hypotheses about the world. The specific techniques applied to reach such an improvement will depend on the component that needs to be improved, the prior data from which the agent can learn the model, and the feedback generated on that data. It is precisely this feedback that determines the way in which the types of learning are classified [8]:

- *Supervised learning.* Based on input/output pairs, the agent observes a mapping function between the two. This requires that the data on which the model is trained to serve as examples, i.e., the output is known for the inputs it receives. Such an output is called a label. Classification and regression are the best-known tasks.
- *Unsupervised learning.* The agent is able to find patterns of behaviour in the data without the need for explicit feedback. These patterns can be extracted from hidden relationships in the data, or they can be used to detect potential clusters of input examples. Clustering is the best-known unsupervised task.
- *Reinforcement learning.* The agent learns from a mechanism of rewards and punishments so that it will end up deciding which actions are the most profitable prior to the reinforcement.

Depending on the categorisation, we could find other categories, such as semi-supervised learning, transfer learning or deep learning, among others. In the chapter “Foundations of Machine Learning for Software Engineering”, a review of machine learning is made, going deeper into the most used techniques throughout the following chapters.

Knowledge, Reasoning and Planning

This category refers to forms of embodying real-world concepts, and how to infer new knowledge or explain causal facts. In this case, knowledge-based agents apply reasoning over their internal knowledge conceptualisation to decide the actions to be taken. Seemingly, it is like these agents knew things, but in a restricted environment where they still apply limited representations of knowledge taken from their knowledge base. For these systems to reach good decisions there must be a way to incorporate new pieces of knowledge, i.e., sentences duly expressed in their own knowledge representation language, as well as a way to query what the agents know. Both operations allow new sentences to be inferred from the existing ones using algorithms based on the inference rule, whereby patterns are found that the algorithms can use to find proof. In this category, it is common to find methods based on logic, model checking, local search, SAT solving, etc. Also, ontologies are considered within this category, as they enable the representation of large-scale knowledge. Ideally, general-purpose ontologies are expected to cover a wide variety of knowledge of any domain, but they are difficult to build and handle.

How to represent the knowledge, as well as selecting the right algorithms, affect the actions and states considered for planning the sequence of actions for an intelligent agent to accomplish a goal. In general, notice that planning systems are problem-solving algorithms that operate on factored representations of states and actions. In the real world, agents should be able to handle uncertainty due to partial observability, nondeterminism or adversaries, if they want to improve their decisions. Bayesian networks are commonly applied in this context, as they provide a concise way to represent conditional independence relationships in the domain. An extension of Bayesian networks known as decision networks also serve as a formalism to express and solve decision problems. In the specific case of stochastic environments, Markov decision processes allow specifying the probabilistic outcomes of actions and a reward function to calculate the reward value for each state. In this way, the utility of a sequence can be calculated as the sum of all rewards over such a sequence.

Communicating, Perceiving and Acting

These agents pretend to understand and simulate human abilities such as language processing, vision and movements. Regarding the communicative capacity of machines, Alan Turing already proposed his intelligence test based on the communicative capacity of the machine through language. The goal of natural language processing (NLP) is to communicate some knowledge to the interlocutor, as well as to receive messages from a speaker and infer significant meaning. In this way, these agents aim to communicate with humans in situations where interaction is required, e.g., through bots. We must also consider that most human knowledge is hidden in the text, e.g., Wikipedia could be an information source example. Therefore, if we want the agent to learn, it is better if it can understand its meaning.

In addition to language, humans also have the ability to observe their environment with their eyes. Agents must therefore be able to interpret and infer knowledge from the flood of data coming from cameras or sensors. Most agents using vision

employ passive detection, i.e., they apply some computation on the received signals. Active detection, on the other hand, consists of sending a signal, such as radar, and detecting a reflection. The agent must perform feature extraction by performing direct calculations applied to the sensor responses. A feature is a number that is obtained by applying simple calculations and contains very useful information.

Finally, robots are physical agents that perform tasks by manipulating the physical world. This world is observed through sensors. In addition, they are equipped with a set of legs, wheels, joints, and grippers designed to assert physical forces on the environment. Some robots have the ability to move through paths, i.e., a sequence of points in the geometric space of the agent's movement. Different types of algorithms are applied to find the best path to move, including problem-solving techniques. This process is called motion planning.

1.3 AI4SE: Artificial Intelligence for Software Engineering

The idea of AI-enhanced software engineering (SE) is not new. Automatic programming and knowledge-based assistants like expert systems were prominent in the 1980s. Forty years later, both software development practices and artificial intelligence have developed substantially. Thus, the landscape of AI-enhanced SE was reviewed in a study in 2012 [5], identifying three areas: probabilistic software engineering; classification, learning and prediction for software engineering; and search-based software engineering. If we compare with the classification seen in Sect. 1.2, we can observe that this study does not consider communication, AI methods with great potential such as natural language processing for current proposals like bots or programming assistants.

Note that software engineers do not need to be familiar with AI techniques, which makes it impossible for them to be able to identify which predictive models or metaheuristics would be most appropriate in each situation [2]. In fact, there are often misconceptions about what AI can do for software engineers. As a recent paper [3] points out, it is common to think that AI can solve any problem. But the reality is that today, the average AI-based solution rarely provides the quality that meets its actual practicality. Making the right choice requires clear objectives and a good understanding of each technique. Given the potential impact and risks involved, it is necessary to consider the point at which AI should be applied and its actual responsibility within the process and outcomes [4].

The evolution in the way software is constructed, its increasing complexity, as well as easier access to software data and AI technologies [4], has led to a growing interest in the synergy between SE and AI. The scientific community is a clear reflection of this interest. Since 2017, AI-enhanced SE studies have accounted for more than 20% of the total papers published internationally in the field of software engineering [6]. In the pre-pandemic year of 2019, a new milestone was reached with a total of 28 scientific workshops, where emerging AI topics such as bots or deep learning gained special relevance for the SE community. Figure 1.1 shows the evolution in terms of

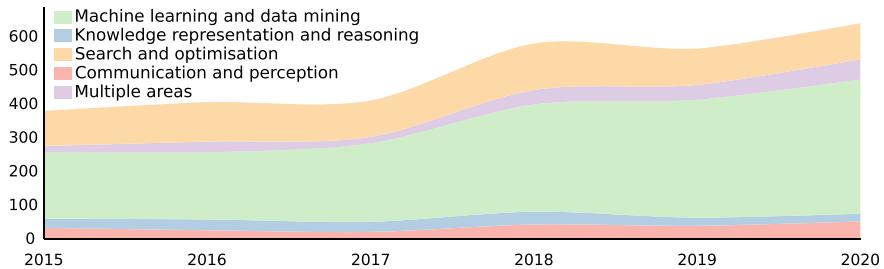


Fig. 1.1 Evolution of AI-enhanced SE (taken from [6])

the number of publications per year in AI-enhanced SE, divided according to the classification of Russell and Norvig. In this study [6], we observe that the two areas of AI that are mostly applied to SE problems are machine learning and problem-solving, i.e., search and optimisation.

According to [6], the analysis of development aspects—often involving mining repository data—and building models for maintenance tasks—e.g., defect prediction and bug detection—are particularly popular. Also, some SE tasks frequently treated as optimisation problems are those related to maintenance and evolution, and programming. In testing, search has been intensively applied to the extent that search-based software testing has become a consolidated area itself. Knowledge and reasoning present a quite regular distribution among topics like modelling, services and testing. Finally, communication-oriented technologies fit more often with those topics where textual information abounds, such as requirements, development assistance and programming.² Table 1.1 lists the topics where AI-based solutions have already been applied.

1.4 Organisation of This Book

The aim of the book is to provide a practical introduction to the use of AI techniques to improve and/or optimise the different phases of the software development process, from initial project planning to final deployment. To this end, the different phases of a software project will be addressed but from a differential perspective: an approach based on AI techniques that have already been successfully applied to specific but complex and tedious problems within each of these phases. As described in Sect. 1.2, AI is a broad term and encompasses multiple areas, with applied search and optimisation algorithms (problem-solving) and machine learning being predominant in addressing problems in this field. This volume focuses on them.

² More information can be found at <https://www.uco.es/kdis/sbse/aise/> (last accessed: July 30, 2022).

Table 1.1 Main topics appearing in AI publications for each SE area (extracted from [6]). The order of appearance of the topics indicates the highest recurrence of these solutions

SE area	Topics
Methodological aspects and general SE topics	Software engineering, business process, software, agile, task analysis, open source, software quality
Project management	Effort estimation, release planning, human factors, software, project management, task analysis, feature extraction, release engineering
Requirements	Requirements, non-functional requirements, requirements elicitation, use cases, ambiguity, next release problem, requirements analysis, user stories
Architecture and product lines	Software product lines, software architecture, variability, feature models, reverse engineering, architecture recovery, software maintenance, software testing
Modelling and design	Model-driven engineering, model checking, UML, OCL, reverse engineering, model inference, metamodel, modelling, program understanding, software design
Programming	Source code, code search, API, program comprehension, software evolution, code reuse, open source, libraries, Java, code review
Development support	Stack overflow, Github, continuous integration, open source, software repositories, git, devops, software analytics, software evolution, pull request
Maintenance and evolution	Defect prediction, software evolution, software maintenance, refactoring, bug, code smells, program repair, technical debt, reverse engineering, clone detection
Quality and metrics	Software quality, software metrics, measurement, code metrics, reliability, security, code quality, quality assurance, refactoring, robustness
Testing, validation and verification	Software testing, automated test generation, test case prioritisation, mutation testing, test case generation , regression testing, test data generation, debugging, unit testing, random testing
Services and cloud computing	Cloud computing, service composition, quality of service, web services, service recommendation, service discovery, microservices, cloud, cloud services
Mobile, IoT and cyber-physical systems	Android, mobile apps, software testing, app/user reviews, app market analysis, stack overflow, feature extraction, mobile development, program analysis, software evolution, energy consumption

This book is designed for computer science professionals (practitioners, developers, software engineers or managers), as well as advanced graduate and PhD students, and researchers interested in AI4SE. Therefore, all chapters are explained in a didactic way and encourage the practical nature of the discipline. To this end, most of the chapters will contain examples and source code that can be downloaded by the reader to try their operation. The repository is hosted by Zenodo at <https://dx.doi.org/10.5281/zenodo.6965479>.

The volume is organised in two parts. After this introduction, the following chapters cover the classic phases of the software development process: project management, requirements engineering, analysis and design, coding, deployment, unit and system testing, and maintenance. The second part of the book will include two state-of-the-art chapters, which will review the concepts related to the AI techniques used in these chapters: metaheuristics and machine learning.

1.4.1 AI for the Software Development Process

Chapters “Artificial Intelligence in Software Project Management”—“Intelligent Software Maintenance” will address the different AI4SE techniques applied in each of the above phases.

Chapter Artificial Intelligence in Software Project Management

This chapter discusses how artificial intelligence techniques can support software project managers with two key software project management activities: software project scheduling and software effort estimation.

The authors of this chapter mention that software project management is a very challenging task that involves multiple businesses and human factors, and conflicting objectives and that these challenges are exacerbated when dealing with medium to large-size software projects. The authors claim that, under such circumstances, artificial intelligence has the potential to play a significant role in supporting software project managers, enabling them to make more informed management decisions.

First, the requirements engineering problem is introduced and two specific use cases supporting decision-making related to the requirements elicitation from user feedback, and to requirements prioritisation, are presented.

Then, some solutions to the two problems that are based on metaheuristics and AI techniques, specifically machine learning, natural language processing, and genetic algorithms, are illustrated. The solutions have been evaluated in the field in industrial settings highlighting the importance of applying artificial intelligence solutions to existing organisational problems.

Based on the two use cases, the authors of this chapter have also glimpsed at recent trends to support requirements management activities by exploiting artificial intelligence techniques based on optimisation, machine learning and deep learning, with the objective of improving the overall efficiency of the process and to facilitate the work of the analysts and decision-makers performing those activities.

Finally, a discussion of the use cases in the context of the requirements engineering management process is included, highlighting opportunities and limits of the artificial intelligence approaches and future possibilities to apply artificial intelligence in many other activities of the requirements engineering process.

Chapter Requirements Engineering

This chapter focuses on requirements engineering, in particular, on two specific use cases, namely requirements elicitation from textual user feedback, and requirements prioritisation.

The authors present solutions to the two problems based on artificial intelligence techniques, specifically machine learning, natural language processing, and genetic algorithms. In addition, the application of the proposed methods in industrial contexts allowed the authors to validate their usefulness in terms of increased efficiency of the organisations during their decision-making processes.

Requirements engineering aims at supporting the understanding of the purpose of a software system to be built, and at keeping the whole design and development process aligned with it. Research in requirements engineering provides methods and techniques to support various activities in the requirements life cycle, from requirements elicitation to requirements verification and validation. This chapter focuses on those techniques that exploit artificial intelligence.

In this chapter, first, the requirements engineering and key concepts are introduced. Then, the requirements elicitation use case and the requirements prioritisation use case are described.

Next, the authors address the two use cases in the broader context of the requirements engineering management process.

Finally, a discussion is offered highlighting opportunities and limits of the artificial intelligence approaches and current trends in the use of artificial intelligence in requirements engineering.

Chapter Leveraging Artificial Intelligence for Model-Based Software Analysis and Design

In this chapter, the authors review existing applications of artificial intelligence methods to model-based software engineering problems. So a representative use case of how a model-based software analysis and design problem can be solved using genetic algorithms is presented. In particular, the chapter focuses on the well-known and challenging modularisation problem: splitting an overarching, monolithic model into smaller modules.

Several approaches have been proposed that integrate modelling with artificial intelligence methods such as genetic algorithms, among others. Two encodings are presented, the model-based and the transformation-based encoding, which are both applied for the modularisation of entity-relationship diagrams.

First, the authors of this chapter provide the background regarding model-driven engineering. Then, they introduce the running example of this chapter (the modularisation of entity-relationship diagrams) and summarise selected applications of artificial intelligence for model-based engineering.

Next, the chapter focuses on describing the model-based and transformation-based encoding strategies. In addition, their usefulness in implementing solutions for modularising entity-relationship diagrams is described.

Finally, the authors discuss how these encodings may be adapted to other structural models and conclude with an outlook on future research lines related to software modelling intelligence.

Chapter Statistical Models and Machine Learning to Advance Code Completion

Code completion is an important feature in any Integrated Development Environment that is daily used by software developers. It helps to speed up the process of coding by suggesting the next code to write, thus reducing common mistakes. While early code completion tools are based on program analysis, artificial intelligence is used in the most advanced code completion, providing a better-sorted list of suggestions.

This chapter makes review of the current state-of-the-art approaches using artificial intelligence for code completion. The three main approaches used for AI-assisted code completion are based on software mining, statistical language models and deep learning. In the approaches based on software mining, the idea is to suggest code that matches code patterns previously identified in other pieces of code. Statistical language models, like the N-gram language model, capture the probability distributions of sequences of tokens and use these distributions to suggest the most probable next token. The deep learning approaches transform the problem of predicting the next token into a supervised learning problem and use deep neural networks to solve it.

The chapter describes the three approaches in different sections, provides examples of them, and points to references for the interested reader. In the final conclusions, the authors highlight the drawbacks of software mining and suggest topics that require further research in the field, like the combination of program analysis and data-driven approaches and the use of human developers in the research studies.

Chapter Cloud Development and Deployment

Two challenges in software deployment and configuration are the binding of web services and the selection of an optimal configuration in highly-configurable systems. In the web services binding, the goal is to select the web service to use from a family of functionally equivalent and compatible web services. This is done to optimise various non-functional properties, such as cost, availability, and others that are integrated into what is known as Quality of Service. Regarding the highly-configurable systems, the goal is to choose one of the many possible configurations of the system that better fits user preferences taking into account conflicting objectives, like cost, functionality or complexity.

The chapter illustrates how to apply artificial intelligence to solve these two problems. In particular, the authors use GRASP with Path Relinking to solve the web services binding problem and Multi-Objective Evolutionary Algorithms to find the optimal configuration in highly-configurable systems.

After an introduction to the topic, the chapter presents the challenges that can be addressed using AI techniques in the context of software deployment and configuration. Then, it illustrates how to solve the web services binding and the optimal configuration search in highly-configurable systems using running examples and metaheuristic algorithms. Finally, the authors conclude that AI-based techniques like the ones used in the chapter will be more necessary in the near future, where microservices architectures and domain-driven development are leading the later phases of the software development life cycle.

Chapter Automated Support for Unit Test Generation

To illustrate how artificial intelligence can support unit testing, this chapter introduces the concept of search-based unit test generation. This technique frames the selection of test input as an optimization problem (seeking a set of test cases that meet some measurable goal of a tester) and unleashes powerful metaheuristic search algorithms to identify the best possible test cases within a restricted timeframe.

Creating unit tests is a time and effort-intensive process with many repetitive, manual elements. Automation of elements of unit test creation can lead to cost savings and can complement manually-written test cases.

The authors first introduce a running example, give an overview of unit testing and test design principles, and present the terminology used.

Then, they describe and explain the elements of search-based test generation, including solution representation, fitness (scoring) functions, search algorithms, and the resulting test suites.

Finally, advanced concepts that build on the foundation laid in this chapter are presented. So the chapter concludes by discussing these advanced concepts and gives pointers to further reading for how artificial intelligence can support developers and testers when unit testing software.

Chapter Artificial Intelligence Techniques in System Testing

This chapter presents where and how artificial intelligence techniques can be applied to automate and optimise system testing activities.

System testing is essential for developing high-quality systems, but the degree of automation in system testing is still low. Therefore, there is high potential for artificial intelligence techniques like machine learning, natural language processing, or search-based optimization to improve the effectiveness and efficiency of system testing.

In this chapter, first, the authors provide an overview of the system testing state-of-the art, where testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

Then, they identify different system testing activities (test planning and analysis, test design, test execution, and test evaluation) and indicate how artificial intelligence techniques like optimization algorithms, natural language processing and machine learning could be applied to automate and optimise these activities.

Furthermore, the chapter includes an industrial case study on test case analysis, where artificial intelligence techniques are applied to encode and group natural language into clusters of similar test cases for cluster-based test optimization.

Finally, the chapter discusses the levels of autonomy of artificial intelligence in system testing.

Chapter Intelligent Software Maintenance

Software products are not perfect, they contain defects or depend on software (e.g., libraries) that contain defects. Once discovered, these defects have to be fixed, which requires a change in the software product. The software also ages: new functionality is required, there are changes in the requirements, old unused functionality should be removed, etc. All this requires software to be maintained during its lifetime. Software maintenance is a high-cost activity of the software development process: more than 70% of the total cost of the software could be used in maintenance.

This chapter illustrates with two sample applications how artificial intelligence can be applied in the context of software maintenance to alleviate the costs. The first application consists of code smell detection using deep learning. The idea is to automatically find those parts of the code that needs to be refactored in order to fit with good programming practices and to be easy to understand and maintain. The second application proposes the use of machine learning to reformulate questions for reusable code search in order to get more effective answers that help developers during their maintenance tasks.

The chapter starts with an introduction to software maintenance activities. Next, the application of deep learning to detect design anti-patterns is presented. The authors propose a convolutional neural network that uses historical source code metrics as inputs. The results show that the proposal outperforms state-of-the-art techniques for the same purpose.

Then, in a new section, the application of machine learning to query reformulation for reusable code search is presented. The proposal uses information from Stack Overflow to find candidate API classes based on pseudo-relevant feedback and two-term weighting algorithms. It then ranks the candidates using Borda count and semantic proximity. The top 10 candidates in this rank are added to the query. The results show that the proposed method is more effective than the state-of-the-art techniques.

1.4.2 *Background on Metaheuristics and Machine Learning*

Metaheuristics and machine learning algorithms are the two families of artificial intelligence techniques most commonly applied in software engineering. The reader can find many application cases in chapters “Artificial Intelligence in Software Project Management”–“Intelligent Software Maintenance”. For those unfamiliar with these techniques, chapters “Metaheuristics in a Nutshell” and “Foundations of Machine

Learning for Software Engineering” provide some background where the reader can find further references to deepen the working principles of these techniques.

Chapter Metaheuristics in a Nutshell

Metaheuristics are the most popular algorithms used to solve the optimisation problems that arise in the context of search-based software engineering. Many of the previous chapters use metaheuristic algorithms in their proposals. In order to help readers who are not familiar with the working principles of metaheuristics, this chapter provides a description of the main metaheuristic algorithms appearing in the previous chapters.

After a short introduction, the chapter presents some background definitions, including the concepts of optimisation problem, multi-objective optimisation and metaheuristics. Next, it describes two popular taxonomies to classify ad hoc heuristics and metaheuristics. Metaheuristics are usually divided into two categories: trajectory-based and population-based metaheuristics. Two separate sections provide a more detailed description of these two categories including details of some of the most popular examples of these metaheuristics.

The chapter also includes a section presenting methodological tools and good practices to evaluate the performance of different metaheuristic algorithms. The section describes quality indicators used in the context of single- and multi-objective optimisation as well as a procedure to apply statistical tests and check if the observed differences are significant or not.

Chapter Foundations of Machine Learning for Software Engineering

Machine learning gives computers the ability to solve problems based on the previous experience. There are many examples of applications of machine learning to software engineering tasks in chapters “Artificial Intelligence in Software Project Management”–“Intelligent Software Maintenance” of this book. In the context of software engineering the experience comes in the form of software repositories, software project features, textual descriptions of requirements, textual test specifications, historical information of software code metrics, and textual questions and answers in developers’ sites like stack overflow. The goal of this chapter is to introduce the background on machine learning to understand the techniques used in chapters “Artificial Intelligence in Software Project Management”–“Intelligent Software Maintenance”.

After an introduction, the chapter presents the pipeline of tasks used in any machine learning application. Then, it presents three of the main learning tasks in machine learning: classification, regression and clustering. For each of them, there is a section where some of the most popular techniques are presented together with metrics used for their evaluation and the methodology used to evaluate the techniques. Deep learning is also covered in an additional section where convolutional and recurrent neural networks are presented together with autoencoders and other techniques.

References

1. K.T. Al-Sarayreh, K. Meridji, A. Abran, Software engineering principles: a systematic mapping study and a quantitative literature review. *Eng. Sci. Technol.* **24**(3), 768–781 (2021)
2. L. Briand, AI in SE: a 25-year Journey (Keynote), in *1st International Workshop on Software Engineering Intelligence* (2019)
3. A.D. Carleton, E. Harper, M.R. Lyu, S. Eldh, T. Xie, T. Menzies, Expert perspectives on AI. *IEEE Softw.* **37**(4), 87–94 (2020)
4. R. Feldt, F. Gomes de Oliveira Neto, R. Torkar, Ways of applying artificial intelligence in software engineering, in *Proceedings of 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)* (2018), pp. 35–41
5. M. Harman, The role of artificial intelligence in software engineering, in *Proceedings of 1st International Workshop on Realizing AI Synergies in Software Engineering (RAISE)* (2012), pp. 1–6
6. A. Ramírez, J.R. Romero, Synergies between artificial intelligence and software engineering: evolution and trends, in *Handbook on Artificial Intelligence-enhanced Software Engineering*, ed. by G.T. et al. (Springer, 2022)
7. D. Ross, J. Goodenough, C. Irvine, Software engineering: process, principles, and goals. *Computer* **8**, 17–27 (1975)
8. S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th edn. (Pearson, 2020)

Part I

Planning and Analysis

Chapter 2

Artificial Intelligence in Software Project Management



Liyan Song and Leandro L. Minku

Abstract The success of a software project highly depends on how well the project is managed. This includes crucial activities such as estimating the effort required to develop the software project, creating a software project schedule including allocation of human resources, managing project risks, monitoring progress, etc. Inadequate handling of such activities can thus lead to serious consequences to software companies. However, software project management is also a very challenging task that involves multiple business and human factors, and conflicting objectives. These challenges are exacerbated when dealing with medium- to large-size software projects. Under such circumstances, artificial intelligence has the potential to play a significant role in supporting software project managers, enabling them to make more informed management decisions. This chapter discusses how artificial intelligence techniques can support software project managers with two key software project management activities: software project scheduling and software effort estimation.

2.1 Introduction

Software project management is a software engineering task concerned with ensuring that the software is delivered on time and on budget, and in accordance with the requirements of the stakeholder organisations [1]. It includes crucial activities such as estimating the effort required to develop the software project, creating a software project schedule including allocation of human resources, managing project risks, monitoring progress, etc.

L. Song

Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China

e-mail: songly@sustech.edu.cn

L. L. Minku (✉)

School of Computer Science, University of Birmingham, Birmingham, UK

e-mail: l.l.minku@bham.ac.uk

Inadequate handling of such activities can lead to serious consequences to software development companies. For instance, if the software project goes over budget, the company will lose profit and may even get a negative profit margin. Depending on how large the monetary loss is, the software company could even go bankrupt. If the software project overruns, the software development company may lose a client organisation or even be in breach of contract.

However, software project management is also a very challenging task that involves multiple business and human factors, and conflicting objectives. These challenges are exacerbated when dealing with medium- to large-size software projects. Under such circumstances, Artificial Intelligence (AI) has the potential to play a significant role in supporting software project managers, enabling them to make more informed management decisions. This chapter discusses how AI techniques can support software project managers with two key software project management activities: software project scheduling and software effort estimation.

Software project scheduling is the process of organising the work to be done in a software project into different tasks and deciding who will work on which task and when [1]. AI techniques can be used to help software project managers with finding good allocations of employees to tasks with certain objectives in mind, such as minimising the cost and duration of the project. Section 2.2 introduces software project scheduling and AI approaches that can be used to support it, focusing on the work of Alba and Chicano [2], Minku et al. [3] and Shen et al. [4] as examples.

Software effort estimation is the process of predicting the effort required to develop a software project. AI can be used for software effort estimation as a decision support tool, based on which project managers can justify, criticise or adjust the estimation derived by the experts. Such AI-based estimations are repeatable, objective, efficient and can often provide better understanding of the estimation process. There have been many AI approaches proposed for software effort estimation, and we will take the effort estimator proposed by Song et al.'s [5] as an example to explain typical procedures of adopting AI for effort estimation. Section 2.3 introduces software effort estimation and AI techniques to support it.

2.2 Software Project Scheduling (SPS)

Software Project Scheduling (SPS) consists in organising the work to be done in a software project into different tasks and deciding who will work on which task and when [1]. It requires identifying the tasks to be performed in a software project; the dependencies among the tasks; the employees available to perform tasks, their skills and salaries; estimating the effort required to develop each task; allocating employees to tasks; and ultimately producing charts to communicate the project tasks, their duration and employee allocation. In more traditional software development processes, a detailed schedule is produced at the beginning of the project, and this schedule is then adjusted as the project is developed. In agile software development, the initial schedule is typically more coarse, identifying the different phases of the

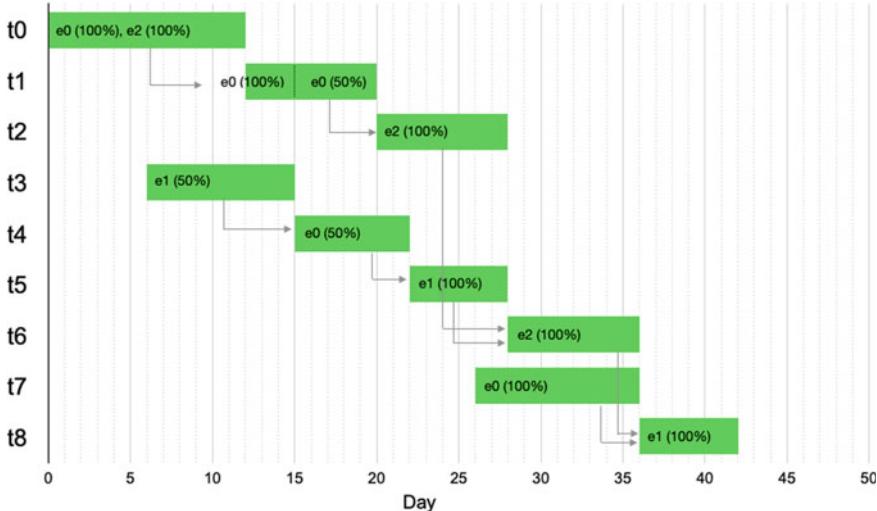


Fig. 2.1 Illustrative example of project Gantt chart with nine tasks t_0 – t_8 and three employees e_0 to e_2 . Arrows represent dependencies between tasks. Percentages in brackets represent the percentage of the employee's full-time work dedicated to the corresponding task. Each vertical gray line represents the end of a given working day

project. More detailed schedules are then produced for each phase or iteration of the project during its development [1].

An illustrative example of project Gantt chart that could have been produced as part of the SPS process at the beginning of a project is given in Fig. 2.1. In this project, there are nine tasks t_0 to t_8 and three employees e_0 to e_2 . Employees can be allocated with different amounts of dedication to tasks (shown in parentheses in the Gantt chart). This dedication corresponds to the percentage of the employee's full-time work that is dedicated to each task. For example, if an employee's full-time work is 8 h per day, then a dedication of 50% would correspond to 4 h per day. Tasks can be allocated to more than one employee at the same time, e.g. e_0 and e_2 are both allocated to t_0 . A given employee can also be allocated to more than one task that occur concurrently. For instance, employee e_0 is allocated to both t_1 and t_4 , which co-occur from Day 16 to Day 20. At the beginning of t_1 , e_0 works with 100% dedication to this task, but, at the end of this task, e_0 shares their time between t_1 and t_4 , dedicating 50% of their time to each of these tasks. Employee e_0 remains working with 50% dedication to t_4 until the end of this task, even though this employee was not allocated to any other concurrent task in this project. It could be, for example, that this employee was dedicating the other 50% of their time to another project.

Typically, schedules are created with certain objectives in mind. For instance, one may be interested in producing a schedule that minimises the cost and duration of the project. For example, the schedule illustrated in Fig. 2.1 is estimated to take 42 days to complete. Possibly, this project could have taken less time to complete if

more employees were allocated to it. However, allocating more employees might include some employees with higher salaries, meaning that the cost of the project could potentially increase. The cost of a software project can be calculated based on the salaries of the employees allocated to it plus the cost of any other resources that may need to be used to develop the project.

Creating a software project schedule can be a daunting task, especially when the project is large and the company has a large number of employees. The space of possible allocations of employees to tasks can be enormous [6]. The complexity of the allocations can also be high, given that employees can work in parallel on different tasks and each task can be allocated to more than one employee at the same time. Constraints such as employees being unable to work on tasks for which they do not have the required skills, employees being unable to work more than a maximum number of hours per day, and task dependencies add to the challenge [3]. Moreover, the objectives that one may be interested in optimising when creating a schedule may frequently be conflicting. For instance, a schedule that allocates expensive staff (those with higher salaries) may lead to a more expensive project, but may result in a shorter duration. Producing an optimal schedule manually is thus a difficult problem.

An alternative to producing a schedule completely manually is to adopt AI approaches. These approaches would still require project managers to decide how to split the project into different tasks and provide information such as task dependencies, employees' salaries and skills. However, based on such information, AI could support software managers in creating project schedules with the aim of optimising certain objectives such as project cost and duration. Section 2.2.1 discusses some of the existing AI approaches for SPS.

2.2.1 *AI Approaches for SPS*

Despite SPS being typically a human activity, the objectives that the software manager may be interested in optimising when creating a schedule (e.g. project cost and duration) can be mathematically formulated. Therefore, SPS can be seen as an optimisation problem, i.e. a problem where we are interested in finding a solution that minimises or maximises one or more objective functions. Such formulation enables the SPS problem to be solved by AI approaches called approximate search algorithms, such as metaheuristics [7] (see chapter 'Metaheuristics in a Nutshell'). Solutions generated automatically by AI can then be used to support software managers in making more informed decisions during the SPS process.

Several different formulations of SPS as an optimisation problem have been investigated in the literature. For example, some researchers formulate SPS as the problem of finding the order with which tasks should be undertaken and the assignment of employees to tasks that minimises the duration of the software project [8], or the cost (salaries paid) and the amount of penalties incurred from missing milestone deadlines in the project [9]. Others formulate it as the problem of finding the allocation of employees to tasks that minimises the duration and cost (salaries paid) of the

Table 2.1 Example of employees available for a given project

Employee name	Hourly salary	Skills	Normal hours	Maximum dedication
$e_0 = \text{John}$	$s_0 = \$10$	$sk_0 = \{ \text{Python, SQL} \}$	$h_0 = 8\text{ h}$	$md_0 = 1$
$e_1 = \text{Tom}$	$s_1 = \$30$	$sk_1 = \{ \text{C++, TCP/IP, HTTP} \}$	$h_1 = 8\text{ h}$	$md_1 = 1$
$e_2 = \text{Jack}$	$s_2 = \$25$	$sk_2 = \{ \text{Java, Javascript, SQL, JSON, Testing} \}$	$h_2 = 8\text{ h}$	$md_2 = 1$
$e_3 = \text{Claire}$	$s_3 = \$27$	$sk_3 = \{ \text{UML, Java, SQL, Testing, Mocks} \}$	$h_3 = 8\text{ h}$	$md_3 = 1$
$e_4 = \text{Mary}$	$s_4 = \$35$	$sk_4 = \{ \text{Data science, Python, SQL, JSON} \}$	$h_4 = 8\text{ h}$	$md_4 = 1$
$e_5 = \text{Junior}$	$s_5 = \$20$	$sk_5 = \{ \text{C++, TCP/IP, HTTP} \}$	$h_4 = 8\text{ h}$	$md_5 = 1$

software project, while the order with which tasks are performed is calculated by a separate deterministic algorithm based on the allocations and dependencies among tasks [2]. Different problem formulations can also take into account different levels of detail. For example, some formulations take into account different salaries for work in normal hours or overtime [9], the tasks' required skills and employees' skills [3, 9], the level of proficiency of employees on different skills [4, 9], the potential mistakes in the estimations of the effort required to complete tasks [4], the dynamic events that may affect the software project during its development [4], etc.

Section 2.2.1.1 presents a popular problem formulation that was proposed in [2], and briefly discusses its extension [4] to take into account additional aspects relevant to SPS. Section 2.2.1.2 briefly discusses the metaheuristic proposed in [3] to solve this problem and its extended version [4]. Even though the proposed problem formulations and metaheuristics have been proposed in [2–4], this book chapter concentrates on discussing them in a more didactic manner.

2.2.1.1 An SPS Problem Formulation

Alba and Chicano [2] formulated the SPS problem as the problem of finding an allocation of employees to tasks that minimises the cost and duration of the project. This formulation is a landmark formulation that has inspired other more detailed formulations [4]. As it is a simple formulation that is easy to understand, this section will focus on it, and then briefly explain how it was extended to include more details that are relevant to realistic SPS scenarios. We explain what information this formulation requires software managers to provide about the project and available employees, how a candidate solution to the SPS problem looks like in this formulation, what objectives are intended to be optimised and what constraints a solution must satisfy to be feasible.

Information About The Software Project And Available Employees

Alba and Chicano's problem formulation [2] considers that the following information about the software project and the employees available to work on it is provided by the software manager:

- **Employees**—there are n employees e_0, e_1, \dots, e_{n-1} available for the project, with salaries s_0, s_1, \dots, s_{n-1} , sets of skills $sk_0, sk_1, \dots, sk_{n-1}$, number of normal working hours in a day h_0, h_1, \dots, h_{n-1} and maximum dedication $md_0, md_1, \dots, md_{n-1}$. Table 2.1 provides an illustrative example of information that could have been provided by a software manager for $n = 6$ employees.

The maximum dedication can be specified as a percentage of the normal working day. For instance, 1 means 100% of a normal working day, whereas 1.25 means 125% of a normal working day. Assuming that a normal working day has 8h as in Table 2.1, a maximum dedication of 1 would mean $1 \cdot 8 = 8$ h per day, i.e. the employee is not allowed to work overtime. A dedication of 1.25 would mean $1.25 \cdot 8 = 10$ h per day, i.e. the employee is allowed to work up to 2h overtime.

In practice, a much larger number of employees may be available than those in the example provided in Table 2.1. More details about the employees could also be specified by extending this problem formulation. For instance, Shen et al. [4] also considered each employee's overtime salary and level of proficiency on each skill. Such extra information can be important as it would affect the cost and duration of the project. In addition, employees' availabilities may be affected by dynamic events that may occur over time. For instance, an employee may become unavailable to work due to illness, or may leave the company. Therefore Shen et al. [4] also associated each employee to a status indicating whether this employee is currently available or not to work. A change in such status is considered as a critical event

Table 2.2 Example of tasks in a cattle monitoring and tracking project

Task name	Required effort	Required skills
$t_0 = \text{Design Database}$	$ef_0 = 24$ person-hours	$rsk_0 = \{\text{SQL}\}$
$t_1 = \text{Implement Database}$	$ef_1 = 24$ person-hours	$rsk_1 = \{\text{SQL}\}$
$t_2 = \text{Design GUI}$	$ef_2 = 32$ person-hours	$rsk_2 = \{\text{Javascript}\}$
$t_3 = \text{Implement GUI}$	$ef_3 = 40$ person-hours	$rsk_3 = \{\text{Javascript}\}$
$t_4 = \text{Design Communications Protocol}$	$ef_4 = 80$ person-hours	$rsk_4 = \{\text{TCP/IP}\}$
$t_5 = \text{Implement Communications Protocol Class}$	$ef_5 = 40$ person-hours	$rsk_5 = \{\text{TCP/IP, C++}\}$
$t_6 = \text{Implement Cattle Class}$	$ef_6 = 40$ person-hours	$rsk_6 = \{\text{Java, Javascript}\}$
$t_7 = \text{Implement Monitor Class}$	$ef_7 = 40$ person-hours	$rsk_7 = \{\text{C++}\}$
$t_8 = \text{Implement Data Transmission Class}$	$ef_8 = 80$ person-hours	$rsk_8 = \{\text{C++}\}$
$t_9 = \text{Test System}$	$ef_9 = 160$ person-hours	$rsk_9 = \{\text{Testing}\}$

that immediately triggers rescheduling of the project, so that the project schedule remains feasible and its cost and duration remains competitive.

- **Tasks**—the project has m tasks t_0, t_1, \dots, t_{m-1} with required efforts $ef_0, ef_1, \dots, ef_{m-1}$ and sets of required skills $rsk_0, rsk_1, \dots, rsk_{m-1}$. Table 2.2 provides an illustrative example of tasks for a project with $m = 10$ tasks to develop a system to monitor and track cattle in a farm.

The tasks are determined by the software project manager, and could be either coarser grained tasks (e.g. architecture design, database design, communication protocol design, front end development, back end development, front end testing, etc.), or more detailed tasks resulting from the initial design of the system (e.g. develop a specific class, a specific table for the database, etc.). The required efforts are also determined by the software manager, potentially with the support of AI approaches for software effort estimation such as the ones presented in Sect. 2.3. Coarser task granularity may mean that each task is actually composed of many sub-tasks that have dependencies with each other, potentially making their effort estimation more difficult. Finer granularities may require more detailed design, but their required effort may be easier to estimate.

In the example shown in Table 2.2, we assume that a UML diagram for the classes that compose the system has already been designed by the software project manager, leading to a more detailed set of tasks. In practice, a much larger number of tasks may need to be scheduled, especially when dealing with large projects and when a more detailed level of granularity is used. There may also be much more tasks related to software testing, e.g. testing each of the classes separately before an integration test.

Shen et al. [4] extended this problem formulation to also consider that new tasks may be added as a result of changes in the requirements of the project during the software development lifecycle. New tasks can be critical or non-critical tasks. New critical tasks immediately trigger rescheduling of the project, so that they can be incorporated into the project schedule as soon as possible. Regular tasks are accumulated over time and are only incorporated into the project schedule when another critical event occurs or when they need to start so that other existing tasks that depend on it can commence.

- **Task Precedence Graph**—tasks in software projects typically have precedence relations, i.e. some tasks cannot start before other tasks are completed. For instance, the implementation of a GUI cannot start before its design is completed. This could be captured in the form of a task precedence graph, where each node represents a task and an arc going from a given task t_i to t_j means that task t_j depends on the completion of task t_i . Figure 2.2 shows an illustrative example of task precedence graph that could have been generated by the software manager for our illustrative project.

It is worth noting that a software project may also suffer from other types of uncertainty and dynamic events in addition to changes in employees' status (available or non-available) and new tasks. For instance, there may be changes in task precedence, new employees being hired by the company, removal of tasks from the project due

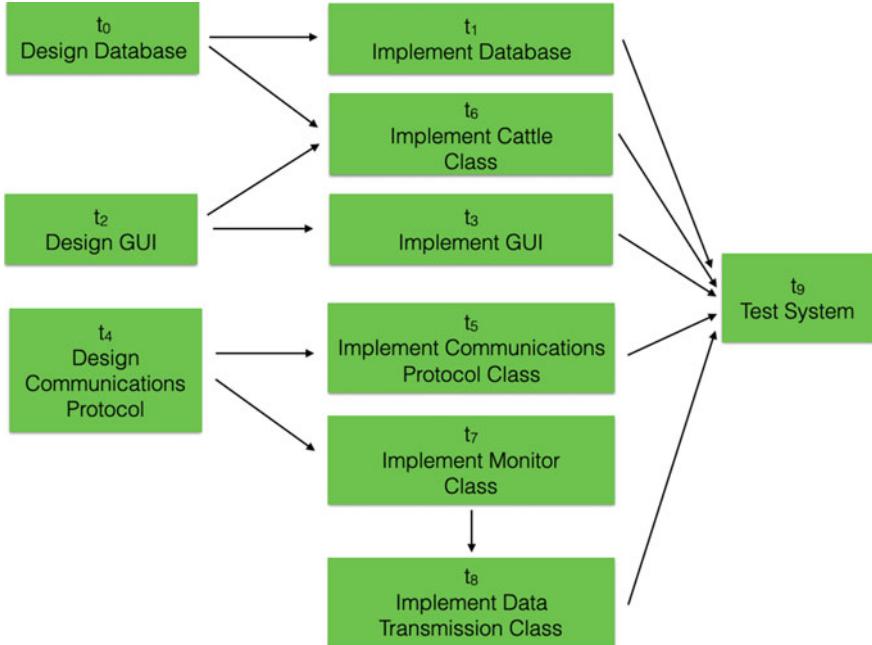


Fig. 2.2 Example of task precedence graph for a cattle monitoring and tracking project

to changes in requirements, etc. These other dynamic events were not taken into account in Shen et al. [4]’s work, but could potentially be dealt with in a similar way to changes in employees’ status and new tasks.

Candidate Solutions and Gantt Charts

A candidate solution to the SPS problem corresponds to a matrix of dedications of employees to tasks. An AI algorithm to solve the SPS problem will thus attempt to automatically find a good allocation of employees to tasks, specifying the dedication of each employee to each task.

An example of possible (not necessarily optimal) solution generated manually for our Cattle Monitoring and Tracking project is shown in Table 2.3. Each cell $[i, j]$ contains a numeric value corresponding to the percentage of employee e_i ’s normal working time dedicated to task t_j . For example, employee e_0 will dedicate 100% of their time (i.e. 8 h per day) to task t_0 when this task is active, whereas employee e_5 will dedicate 50% of their time (i.e. 4 h per day) to task t_5 when this task is active. The dedications have a pre-defined granularity that prevents assigning employees to tasks with too fine grained dedications, which would be difficult to follow in practice. For instance, we could say that dedications can only take the values of 0, 0.25, 0.50, 0.75 and 1. It is possible for a given employee not to be allocated to any task. For example, employees e_1 and e_4 were not allocated to this project despite being available. This could have been done, for example, because these two employees are the ones with

Table 2.3 Example of solution for scheduling a cattle monitoring and tracking project. The numbers in each cell $[i, j]$ correspond to the percentage of employee e_i 's normal working time dedicated to task t_j

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
e_0	1	1	0	0	0	0	0	0	0	0
e_1	0	0	0	0	0	0	0	0	0	0
e_2	0	0	1	0.25	0	0	0.5	0	0	1
e_3	0	0	0	0	0	0	0	0	0	1
e_4	0	0	0	0	0	0	0	0	0	0
e_5	0	0	0	0	1	0.5	0	0.5	1	0

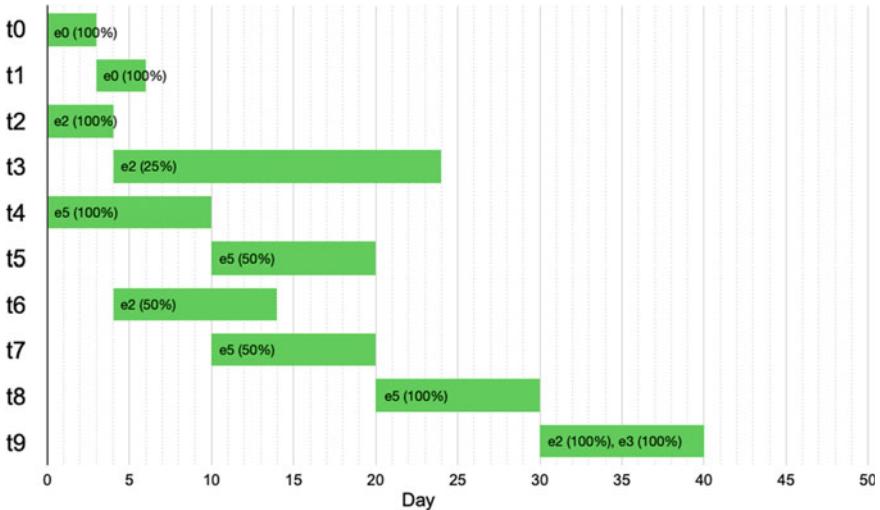


Fig. 2.3 Gantt Chart for the Solution from Table 2.3. Task dependencies are illustrated in Fig. 2.2 and are omitted here so that the Gantt Chart is easier to read. The duration of the project is 40 days and the project cost is \$12,240 based on this solution

the highest salaries, and the software manager might be trying to reduce the cost of the project.

Based on the project tasks, task precedence graph and a given solution, it is possible to generate a Gantt chart for the project in an automated manner. The Gantt chart for the solution in Table 2.3 is shown in Fig. 2.3. An algorithm that can be used to generate this Gantt chart is presented in [3]. This algorithm iteratively adds tasks that can commence given the task precedence graph to the Gantt chart, with their lengths calculated based on the tasks' required effort and the total amount of dedication of employees allocated to them.

The time it takes for a task to complete is calculated by dividing the task's required effort by the sum of the dedications of all employees allocated to this task [2]. For

instance, in our example project, task t_0 has no dependency from the beginning, and can thus start on Day 1 as shown in Fig. 2.3. This task takes 3 days to complete. This is because it requires 24 person-hours (Table 2.2) and one employee (e_0) is allocated to it with $1 = 100\%$ of their time (Table 2.3). So, this task will require $24/1 \text{ h} (=3 \text{ days})$ to be completed. If there were two employees assigned to this task, one with a dedication of 1 and the other with a dedication of 0.5, then this task would require $24/(1 + 0.5) = 16 \text{ h} (=2 \text{ days})$ to be completed.

Simply dividing the task's required effort by the sum of the dedications of the employees allocated to it means that the time it takes to complete a task decreases linearly with the total amount of dedication allocated to it. This means that AI SPS algorithms following this problem formulation could suggest software managers to allocate a very large number of employees to tasks in an attempt to reduce the project duration. However, it is well known that larger teams lead to communication overheads [10]. A larger total dedication resulting from a large number of employees allocated to a task would result in such communication overhead. Shen et al. [4] extended this problem formulation by applying a penalty which increases the task's required effort if the team allocated to it exceeds a maximum size. They also extended the formulation to consider that employees with lower proficiency on the skills required for the task would cause this task to take longer to complete. Such extensions are important because they would prevent AI algorithms from simply allocating an excessively large number of employees to a given task to reduce its duration.

Objectives

Alba and Chicano's formulation [2] considers two objectives to be minimised: project cost and duration. An AI algorithm for the SPS problem based on this formulation would thus help the software manager to find an allocation that minimises the project cost and duration. These objectives are computed as follows:

- **Duration**—the duration of the project can be automatically obtained when the Gantt chart is generated. For example, in Fig. 2.3, we can see that the project will take in total 40 days to be completed, as task t_9 is the last task to be completed and it finishes at the end of Day 40.
- **Cost**—this formulation considers that the cost of the project corresponds to the total amount of salaries paid to the employees allocated to the project. In practice, other costs may also be incurred, depending on whether other resources are required for the development of the project. However, these other costs would normally be unrelated to the project schedule itself, and so are not taken into account in the formulation. The total amount of salaries paid can be determined automatically based on the Gantt chart and the information about the employees' salaries. In particular, for each employee, we need to compute the total number of hours that they will spend on the project. We can then calculate the total salary paid for this employee. For example, for the solution presented in Table 2.3, employee e_0 works in total 3 days for t_0 and 3 days for t_1 ($= 6 \text{ days} \cdot 8 \text{ h} = 48 \text{ h}$). So, their total payment is $48 \cdot \$10 = \480 . After computing the salaries paid to each employee,

we can sum the salaries paid for all employees to determine the cost of the project. The total cost of the project based on this Gantt chart is \$12,240.

Besides the two objectives above, Shen et al. [4] also considered two additional objectives:

- **Robustness**—the task effort estimations may involve uncertainty, i.e. the actual task efforts may not be equal to the estimated ones. Therefore, the allocation of employees to tasks should ideally be as robust as possible to potential variations in task effort. The robustness objective evaluates such robustness. In general terms, it represents the average increase in project cost and duration obtained when simulating variations in task effort based on a Gaussian distribution. The higher the increase in cost and duration, the worse the robustness of the candidate solution to task effort uncertainty.
- **Stability**—software projects may suffer changes over time, such as changes in the status (available or unavailable) of employees and new tasks being added. Whenever a critical event occurs, the project needs to be immediately rescheduled so that its cost and duration remains feasible and competitive. To avoid project disruption, the new allocation of employees to tasks should ideally not be too different from the previous one. The stability objective measures the amount of changes in the dedication of employees to tasks that were previously available in the project. The larger the amount of changes, the more disruptive and undesirable the new project schedule is.

Constraints

Alba and Chicano's formulation [2] also takes into account certain constraints. A solution can only be feasible if it satisfies these constraints. If a given solution does not satisfy these constraints, it is considered infeasible and cannot be adopted for the software project being scheduled. An AI algorithm would thus attempt to find a solution that not only minimises the project cost and duration but also satisfies the constraints. These constraints are explained below:

- **Maximum Dedication**—at any point during the project, the total dedication of a given employee e_i to tasks should not exceed the maximum dedication md_i .

For example, based on the Gantt chart from Fig. 2.3, employee e_5 works concurrently on tasks t_5 and t_7 from days 11 to 20. As this employee's dedication to each of these tasks is 0.5, this employee is working with a total dedication of $0.5 + 0.5 = 1$ from days 11 to 20. This does not exceed the maximum dedication of this employee ($md_1 = 1$). Therefore, the maximum dedication constraint is satisfied during this period of time. However, had the dedication of this employee to task t_7 been 0.75, this employee's total dedication during this period would have been $0.5 + 0.75 = 1.25$, which exceeds their maximum dedication of $md_1 = 1$. In this case, the maximum dedication constraint would have been violated, i.e. the solution would have been infeasible.

Minku et al. [3] proposed to fix infeasible solutions by adjusting their corresponding Gantt charts. This means that the AI algorithm itself would not need to try

and find a solution that satisfies this constraint. Instead, any solution that violates this constraint would be fixed into a Gantt chart that satisfies this constraint. The strategy to fix the Gantt chart is called ‘normalisation’ and its idea is as follows. Consider the infeasible solution discussed above, where the dedication of employee e_5 to tasks t_5 and t_7 is 0.5 and 0.75, respectively. The total dedication of this employee to tasks from Days 11 to 20 is 1.25. If we divide the dedications to tasks t_5 and t_7 by 1.25 when creating the Gantt chart from days 11 to 20, the resulting dedications will be $0.5/1.25 = 0.4$ to task t_5 and $0.75/1.25 = 0.6$ to task t_7 from days 11 to 20. Therefore, the constraint would not be violated anymore. Such fixes are only applied for the periods of time when violations would have occurred. Therefore, an employee may initially have a given dedication to a task, and then decrease this dedication at a later date in order to avoid a violation. For instance, in the Gantt chart from Fig. 2.1, employee e_0 started to work on task t_1 with a dedication of 1 and then reduced it to 0.5 once task t_4 started. Such reduction in the dedication to task t_1 avoided a total dedication larger than 1 from Day 16 to Day 20. It also enabled the project to complete faster, because employee e_0 did not need to work on the whole task t_1 with a dedication of 0.5, only from Day 16 onward. This enabled this task to complete faster than if employee e_1 was working with dedication 0.5 during this whole task.

- **Required Skills**—the team (group of employees) allocated to a given task must, together, have all the skills required to conduct that task. For example, a team composed of one employee who knows Java and one employee who knows SQL, together, holds the skills Java and SQL. This team could thus be allocated to a task that requires Java and SQL, but could not be allocated to a task that requires Java and Javascript. Any solution that assigns a team of employees that do not, together, hold all skills required to develop that task is an infeasible solution.

This definition for the required skills constraint has some problems. In particular, each employee in the team does not need to have all the skills required by the task, so long as the team as a whole has all skills. For instance, it could happen that all employees from Table 2.1 are allocated to task t_9 of Table 2.2, even though only employees e_2 and e_3 have testing skills. Employees with no proficiency on the skills required for a given task would be unable to efficiently work on this task, as they would have to learn the skill on the go. Shen et al. [4] extended this problem formulation to take this into account when computing the duration of each task as mentioned in the explanation of how to generate the Gantt chart. Therefore, an AI algorithm to solve the SPS problem based on the extended formulation would only allocate an employee who is not proficient on a given task’s required skill only if this would bring a worthwhile benefit to the cost and duration of the project as a whole.

Alternatively, it is possible to replace Alba and Chicano [2]’s required skills constraint by a constraint that states that each and every employee allocated to a given task must have all the skills required to develop that task. For instance, if a given task requires Testing and SQL in our example project, the only employees that could be assigned to this task would be e_3 and e_4 .

2.2.1.2 An AI Algorithm for Solving SPS

Given a specific SPS problem formulation, AI algorithms can be adopted to solve it. Different approximate search algorithms have been applied to the SPS problem [7], among which evolutionary algorithms are among the most popular. Evolutionary algorithms are explained in Sect. 5.1 of chapter “Metaheuristics in a Nutshell”. The current section explains an example of evolutionary algorithm design for SPS. This algorithm design was proposed by Minku et al. [3] and is based on the problem formulation introduced by Alba and Chicano [2], but adopting the normalisation strategy to fix the Gantt chart to avoid violations of the maximum dedication constraint as explained in Sect. 2.2.1.1. This strategy, together with some other design choices, enabled this algorithm to significantly outperform other existing algorithms designed for the same problem formulation.

As explained in Sect. 2.2.1.1, this problem formulation does not take into account certain details that are important when scheduling software projects in practice. Therefore, it was extended in subsequent work such as the work of Shen et al. [4]. However, the more detailed problem formulation proposed by Shen et al. requires a more complex evolutionary algorithm to be solved, which in turn requires advanced knowledge of evolutionary algorithms to understand. Therefore, this section focuses on Minku et al.’s algorithm [3]. In practice, software managers would not need to understand in detail how different evolutionary algorithms for SPS work to be able to adopt them, as they would only need to interact with a software tool that implements such algorithms by inputting information about the employees and tasks in order to obtain an allocation.

The evolutionary algorithm proposed by Minku et al. [3] for SPS is depicted in Algorithm 1. It requires us to pre-define certain parameters, namely the population size μ , number of parents λ , probability of crossover P_c and maximum number of iterations $MaxIt$. The best values for these parameters may vary depending on the software project in hands. However, default values of $\mu = 64$, $\lambda = 64$, $P_c = 0.75$, $MaxIt = 69$ led to good results in experiments with several simulated software projects with different features in [3]. Therefore, if the software manager is not familiar with evolutionary algorithms, we recommend the use of these default parameter values.

The algorithm first initialises a population of candidate solutions P with μ candidate solutions (Line 1). Then, a loop is repeated until the maximum number of iterations $MaxIt$ is reached. In this loop, λ parent solutions are selected based on binary tournament selection (Line 3). For each pair of parents A and B , crossover is applied with probability P_c to generate children A' and B' (Line 5). If crossover is not applied, the children A' and B' are clones of the parents (Line 6). Mutation is then applied to the children with a probability $P_m = 1/(mn)$ (Line 7), where m is the number of tasks and n is the number of available employees. The children are then added to the population P (Line 8). The μ best candidate solutions (i.e. those with the best fitness values) from the population P are then selected to survive for the next generation.

Algorithm 1 Evolutionary Algorithm for SPS

Parameters: population size μ , number of parents to be selected λ , probability of crossover P_c , maximum number of iterations $MaxIt$.

- 1: Initialise population P with μ candidate solutions.
 - 2: **repeat**
 - 3: Select λ parents from P using binary tournament selection.
 - 4: **for** each pair of parents A and B **do**
 - 5: With probability P_c , apply crossover between A and B to generate A' and B' .
 - 6: Otherwise, $A' \leftarrow A$ and $B' \leftarrow B$
 - 7: Apply mutation to each cell of A' and B' using probability $P_m = 1/(mn)$.
 - 8: $P \leftarrow P \cup \{A', B'\}$.
 - 9: **end for**
 - 10: Select the μ best candidate solutions from P to survive for the next generation.
 - 11: **until** maximum number of iterations $MaxIt$ is reached
-

As mentioned in Sect. 5.1 of chapter “Metaheuristics in a Nutshell”, some of the key design choices of evolutionary algorithms involve deciding on an appropriate representation, crossover and mutation operators and fitness function. A strategy to deal with the constraints of the problem is also necessary. The algorithm proposed by Minku et al. [3] makes use of a direct representation of the candidate solutions, i.e. a matrix of dedications of employees to tasks with a pre-determined granularity as explained in Sect. 2.2.1.1 and illustrated in Table 2.3.

Crossover between two parent solutions A and B is applied with a pre-defined probability P_c . When crossover is applied, one of the following strategies is randomly used to generate two children solutions A' and B' :

- Exchange Rows—for each employee, select its corresponding dedications to tasks from one randomly chosen parent to compose child A' , and from the other parent to compose child B' . This can be seen as exchanging rows of the matrix of dedications between the parents A and B . An example of crossover by exchanging rows is given in Fig. 2.4 for a project with three employees and four tasks.
- Exchange Columns—for each task, select its corresponding employees’ dedications from one randomly chosen parent to compose child A' , and from the other parent to compose child B' . This can be seen as exchanging columns of the matrix of dedications. An example of crossover by exchanging columns is given in Fig. 2.5 for a project with three employees and four tasks.

Mutation is applied independently to each cell of a child with probability $P_m = 1/(mn)$, where n is the number of employees and m is the number of tasks. This means that, on average, one cell of the matrix of dedications gets mutated for each child. The mutation consists in replacing the current dedication value of the cell by any other possible dedication value. For example, consider the dedication granularity of 0, 0.25, 0.5, 0.75, 1 and a given cell with value 0.75. If this cell is mutated, another value is picked randomly from 0, 0.25, 0.5, 1, e.g. 0. An example of that is shown in Fig. 2.6.

Another important component of an evolutionary algorithm is its fitness function and its strategy to deal with constraints. The fitness function adopted in [3] is the

weighted average between the project cost and duration incurred by the solution, as shown below:

$$f(X) = w_{cost} \cdot cost(X) + w_{duration} \cdot duration(X) \quad (2.1)$$

where $cost(X)$ and $duration(X)$ are the cost and duration of solution X , and w_{cost} and $w_{duration}$ are pre-defined positive real values used to control how much emphasis the software manager would like to place on the cost and duration of the project, respectively. These weights can also work to make the magnitude of the cost values similar to that of the duration values. For example, if we know the project cost would be in the region of ten thousands and the duration would be in the region of hundreds, we could set the weights as $w_{cost} = 0.001$ and $w_{duration} = 0.1$ so that the cost does not dominate the computation of fitness values. Setting the weights as $w_{cost} = 0.01$ and $w_{duration} = 0.1$ would increase the importance of cost when scheduling this project. This would guide the algorithm towards finding an allocation that leads to a cheaper, but possibly longer project.

Good solutions have smaller fitness values according to this function. To deal with the required skills constraint, the $cost(X)$ and $duration(X)$ of the fitness function is forced into a very large value which makes the fitness of any infeasible solution worse than that of any feasible solution. If Alba and Chicano [2]’s original required skills constraint is adopted, this value gets worse when larger numbers of skills are missing in the teams allocated to the tasks. Similarly, if the alternative required skills constraint that states that each and every employee of a team needs to have all skills required by the task is adopted, this value gets worse when larger number of tasks are allocated inadequate teams. This helps guiding the algorithm to find feasible solutions.

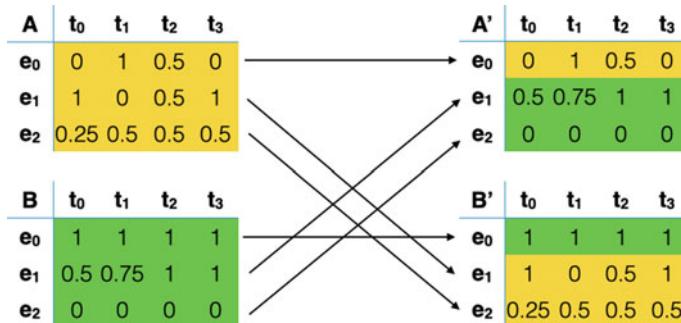


Fig. 2.4 Example of crossover by exchanging rows being applied to two parents A and B shown (left), leading to two children A' and B' (right). The rows used to compose child A' are picked randomly from parent A or B, and the other rows are used to compose child B'

A	t_0	t_1	t_2	t_3
e_0	0	1	0.5	0
e_1	1	0	0.5	1
e_2	0.25	0.5	0.5	0.5

B	t_0	t_1	t_2	t_3
e_0	1	1	1	1
e_1	0.5	0.75	1	1
e_2	0	0	0	0

A'	t_0	t_1	t_2	t_3
e_0	1	1	0.5	1
e_1	0.5	0	0.5	1
e_2	0	0.5	0.5	0

B'	t_0	t_1	t_2	t_3
e_0	0	1	1	0
e_1	1	0.75	1	1
e_2	0.25	0	0	0.5

Fig. 2.5 Example of crossover by exchanging columns being applied to two parents A and B (top), leading to two children A' and B' (bottom). The columns used to compose child A' are picked randomly from parent A or B, and the other columns are used to compose child B'

B'	t_0	t_1	t_2	t_3
e_0	0	1	1	0
e_1	1	0.75	1	1
e_2	0.25	0	0	0.5

Fig. 2.6 Example of mutation being applied to child B' (left), leading to a mutated child (right). Each cell has a probability of $1/(3 \cdot 4) \approx 0.08$ of being mutated to a new randomly picked dedication value. In this example, this resulted in the dedication of employee e_1 to task t_1 being mutated from 0.75 to 0

The maximum dedication constraint is automatically dealt with by fixing the Gantt chart of infeasible solutions by using the normalisation strategy explained in Sect. 2.2.1.1.

2.2.2 Running an AI Algorithm for SPS

This section gives an example of how to run an AI algorithm for SPS. The tool used here implements Minku et al. [3]’s evolutionary algorithm for SPS and also includes some variations of it. The variation that we will experiment with in this section is the one that adopts the alternative required skills constraint that requires each and every employee allocated to a given task to have all the skills required for that task.

The tool was implemented for research purposes, making use of text files to collect information about the project and available employees, and for retrieving the solution in the form of an allocation of employees to tasks. The tool can either be run from

the command line or via a user interface, but this user interface is mainly used for the purpose of setting up the evolutionary algorithm to be adopted and its parameters.

In practice, an intelligent SPS tool would have a user interface for collecting information about the project (such as the information from Table 2.2 and Fig. 2.2) and available employees (as in Table 2.1), and for displaying the solution in the format of an allocation of employees to tasks (as in Table 2.3) and its corresponding Gantt chart (as in Fig. 2.3). The setting of the evolutionary algorithm itself would be considered as an advanced setting that software managers would not need to change unless they are familiar with evolutionary algorithms and would like to investigate whether different settings would lead to better solutions.

The tool is implemented in Java and is available at [11] under GNU GLP 3.0 license.¹ It makes use of the Opt4j framework for meta-heuristic optimisation [12], which supports the implementation of different evolutionary algorithms and is available under the MIT license. Once you download the tool, you will see that there is a folder called *problem-instance-examples*. This folder contains nine different examples of toy software projects to be scheduled. One of them is called *instance_sample_book.txt*, which we will use as a running example in this book. This example corresponds to the illustrative project discussed in Sect. 2.2.1.2. In particular, the employees are the ones listed in Table 2.1, the tasks are the ones in Table 2.2 and the task precedence graph is the one in Fig. 2.2.

The format of this file follows the format of the generator introduced by Alba and Chicano [2]. Comment lines can be added by starting a line with #. When opening this file, you will note that the first lines contain comments listing the numeric identifiers that are being used to represent each skill. The numeric identifiers used for the employees and tasks are the same as the ones used in Tables 2.1 and 2.2.

After the commented lines, the file contains a number of statements that define the information from Tables 2.1 and 2.2 and Fig. 2.2, i.e. the information about the employees, tasks and task precedence graph, respectively. The order of the statements in the file does not matter. The file requires the following statements:

- employee.number—number of employees n .
- task.number—number of tasks m .
- skill.number—number of skills.
- graph.arc.number—number of arcs in the task precedence graph (e.g. the graph from Fig. 2.2 has 12 arcs).
- employee. i .skill.number—number of skills for employee e_i .
- employee. i .salary—salary of employee e_i .
- employee. i .skill. j —skill sk_j of employee i . The number j varies from 0 to employee. i .skill.number-1 and is used just to list employee. i .skill.number different skills for the employee. The skills themselves (e.g. Java, SQL, etc.) are represented by other numeric identifiers which have been listed in the first commented lines of the file for our information.

¹ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

- $\text{task}.i.\text{skill}.\text{number}$ —number of skills required by task t_i .
- $\text{task}.i.\text{cost}$ —required effort of task t_i .
- $\text{task}.i.\text{skill}.j$ —skill rsk_j required by task t_i . The number j varies from 0 to $\text{task}.i.\text{skill}.\text{number}-1$ and is used just to list the different skills required by the task. The skills themselves (e.g. Java, SQL, etc.) are identified by other numeric values which have been listed in the first commented lines of the file for our information.
- $\text{graph.arc}.i$ —definition of arc i of the task precedence graph. The number i varies from 0 to $\text{graph.arc}.\text{number}-1$ and is used just to list the different arcs that define the task precedence graph. A line ‘ $\text{graph.arc}.i=j\ k$ ’ means an arc going from task t_j to task t_k , meaning that task t_k depends on task t_j .

This implementation assumes that all employees always have a maximum dedication of 1. In addition, the unit used for the employees’ salaries must be consistent with the unit of required effort for the tasks. For instance, if the required effort is in person-hours, the salary must be an hourly rate. The monetary unit (e.g. USD) of the salary itself does not matter, so long as all salaries are using the same monetary unit. The project cost will be calculated using the same monetary unit as the salaries. The number of hours of a normal working day is not used by this implementation, as it computes the Gantt chart using the same time unit as the required effort. For instance, if the required effort is in person-hours, the x-axis of the Gantt chart is in hours, rather than days.

The file *GA.xml* contains the setup of the evolutionary algorithm explained in Sect. 2.2.1.2. The other files *OnePlusOneEA.xml* and *RLS.xml* contain other algorithms that have also been analysed by Minku et al. [3]. Most information in this file would not need to be set by the software manager, except for the following:

- Weights w_{cost} and $w_{duration}$ explained in Sect. 2.2.1.2. These weights can be set in the following lines of the *GA.xml* file, where $wCost$ refers to w_{cost} and $wDuration$ refers to $w_{duration}$:

```
<property name="wCost">0.01</property>
<property name="wDuration">0.1</property>
```

- The granularity of the dedications of employees to tasks. The granularity is set so that the dedications can assume values $0, 1/k, 2/k, \dots, 1$. This is specified through the k value in the following line:

```
<property name="k">4</property>
```

- Name of the file containing information about the project to be scheduled. In our example, this is the file *instance_sample_book.txt*. This is specified in the following line:

```
<property name="pspInstanceFileName">
instance_sample_book.txt</property>
```

- Name of the output log file where the solution will be saved. This is specified in the following line:

```
<property name="filename">outputGALog.csv</property>
```

Should the software manager wish to further modify this file to adopt different parameters for the evolutionary algorithm, or to change to a different evolutionary algorithm, further instructions are given in the file *readme.pdf* found with the code.

To run the approach, the following command can be used:

```
java -cp pspea.jar:opt4j-2.4.jar:junit.jar \
org.opt4j.start.Opt4JStarter GA.xml
```

The output log file obtained when running the command line above is given in the file *example-output-log/outputGALog-wcost0.01.csv*. Each row corresponds to a different iteration (generation) of the evolutionary algorithm and logs information about:

- Iteration—this is the number of the iteration whose information is being recorded.
- Evaluations—this is the number of fitness evaluations performed up to the point when the information is recorded.
- Runtime[s]—the runtime elapsed until then.
- COSTDUR[MIN]—the value of the fitness function for the best solution (i.e. the solution with the smallest fitness value) of this iteration.
- Cost—the cost of the project based on the best solution of the population in this iteration.
- Duration—the duration of the project based on the best solution of the population in this iteration.
- Undt—number of tasks for which at least one employee in the team does not have *all* the skills required by the task. For example, if a given task t_j requires skills 0, 1 and 2, and at least one employee allocated to this task does not have all these three skills, this task will count as one of such tasks. When $undt > 0$, this means that the required skills constraint is violated. This column is only used when the alternative required skills constraint that states that each and every employee allocated to a task must have all skills required by that task is adopted.
- Reqsk—is unused for the problem formulation discussed in this section.
- Overwork—is unused for the problem formulation discussed in this section, as the normalisation strategy is adopted to deal with this constraint when creating the Gantt chart associated to the project. So, there is never overwork under this problem formulation.
- PhenotypeBeforeNormalisation—columns from this one onward contain the dedications of employees to tasks for the best solution of this iteration. This is printed based on the code below, where allocation is the matrix of dedications of employees to tasks, n is the number of employees and m is the number of tasks:

Table 2.4 Solution for the running example when using $w_{cost} = 0.01$ and $w_{duration} = 0.1$

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
e_0	0.75	1	0	0	0	0	0	0	0	0
e_1	0	0	0	0	0.5	0.5	0	0	0.25	0
e_2	0	0	1	1	0	0	1	0	0	1
e_3	0	0	0	0	0	0	0	0	0	1
e_4	0	0	0	0	0	0	0	0	0	0
e_5	0	0	0	0	1	0	0	1	1	0

```

for (e=0; e<n; ++e)
    for (t=0; t<m; ++t)
        print(allocation[e][t] + " , ");
    
```

As a research tool, this implementation does not produce a plot containing the Gantt chart for the solution, even though it computes the Gantt chart internally in order to calculate the fitness value. Instead of plotting the Gantt chart, this implementation only records the solution itself. In practice, a decision support tool implementing this algorithm would also produce a plot with the Gantt chart.

In matrix format, the solution retrieved by the algorithm based on $w_{cost} = 0.01$ and $w_{duration} = 0.1$ is shown in Table 2.4. This solution places more emphasis on cost than duration, but still allows duration to play a significant role. The project cost resulting from this configuration was \$13,066.67 and the duration was 237.33 h (29.66 days, if we assume that each working day has 8 h). This is 25.82% (10.33 days) shorter than the duration based on the manual solution from Fig. 2.3, with a cost just 6.75% (\$826.67) higher.

When compared to the manual solution presented in Table 2.3, we can see that this allocation includes employee e_1 with some dedication to tasks t_4 , t_5 and t_8 . This enabled such large speed ups. As the dedications were not high, the cost of the project did not increase much. Moreover, the evolutionary algorithm did not recommend to allocate employee e_4 to any task. This is because adding this employee would cause an increase in the cost of the project without reducing its duration.

The Gantt chart corresponding to this solution is shown in Fig. 2.7. This Gantt chart figure was not outputted by the research tool adopted in this section. It was generated manually for the purpose of this book. However, in practice, an SPS tool would have generated such Gantt chart as part of the output. From the Gantt chart, we can see that the algorithm adopted the normalisation strategy. In particular, employee e_2 is the only employee who can work on tasks t_3 and t_6 , as these tasks require the Javascript skill. These tasks co-occur from around Day 6 to Day 11. The normalisation strategy enabled this employee to be allocated full-time (dedication of 1) to both tasks without leading to overwork. In particular, it adjusted the Gantt chart so that employee e_2 starts the work on task t_3 with dedication of 1, then switches to dedication of 0.5 to

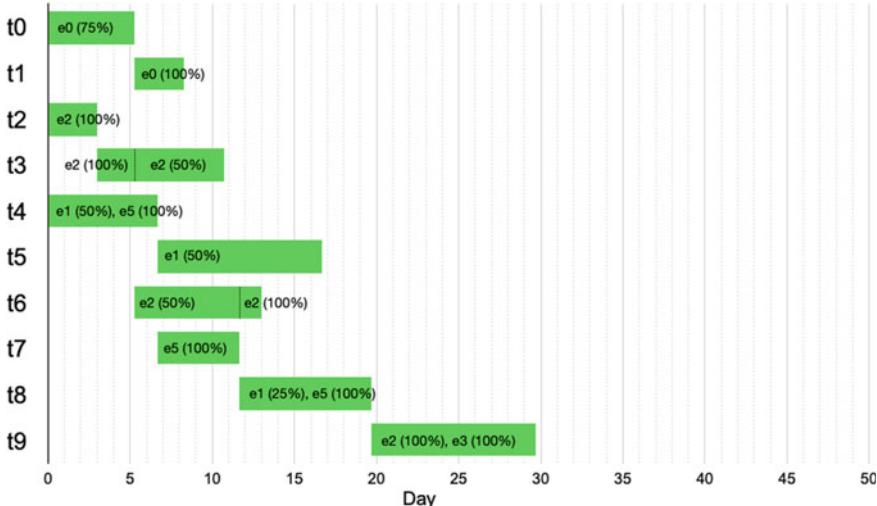


Fig. 2.7 Gantt Chart for the running example when using $w_{cost} = 0.01$ and $w_{duration} = 0.1$. This project schedule has a duration of 29.66 days and a cost of \$13,066.67

task t_3 and 0.5 to task t_6 when these tasks co-occur, then switches to dedication of 1 to task t_6 when task t_3 finishes.

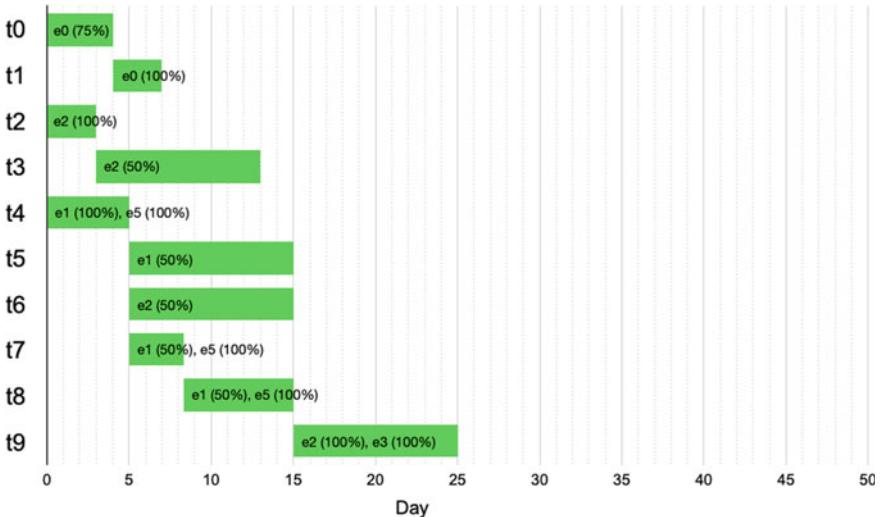
If we assume a more extreme scenario where the software manager would like to save cost as much as possible no matter the increases that this could cause to the duration of the project, we could adopt an even higher weight for the cost (e.g. $w_{cost} = 1$). This can be achieved by replacing the value of the property $wCost$ in $GA.xml$ by 1. In this case, the tool would help the software manager to identify that employees e_1 , e_3 and e_4 do not need to be allocated to this project as their skills are covered by other employees, and that employee e_2 should be allocated to task t_9 , as this is the cheapest employee who can do this task. This would lead to a cost of \$12,080 and a duration of 400h (= 50 days). The output generated by the tool for such scenario is available in the file *example-output-log/outputGALog-wcost1.csv*.

If we assume a scenario where the software manager would like to give a more balanced importance to cost and duration, we could adopt a weight $w_{cost} = 0.003$ instead of 0.01 or 1. In matrix format, the solution retrieved by the algorithm based on this weight is shown in Table 2.5. The algorithm produced a solution where the project cost was \$13,440 (9.8% more expensive than in the manual solution presented in Table 2.3) but with a duration of just 200h (25 days, 37.5% shorter than in the manual solution).

This was achieved by allocating more employees to each task, but only allocating more expensive employees when this was really useful to reduce the duration, as the project cost still plays a significant role in the fitness calculation. Therefore, employees e_3 and e_4 were still not allocated to the project, whereas employee e_1 was allocated full time to tasks t_4 , t_5 , t_7 and t_8 . From the Gantt chart presented in Fig. 2.8,

Table 2.5 Solution for the Running Example when Using $w_{cost} = 0.003$ and $w_{duration} = 0.1$

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
e_0	0.75	1	0	0	0	0	0	0	0	0
e_1	0	0	0	0	1	1	0	1	1	0
e_2	0	0	1	0.5	0	0	0.5	0	0	1
e_3	0	0	0	0	0	0	0	0	0	1
e_4	0	0	0	0	0	0	0	0	0	0
e_5	0	0	0	0	1	0	0	1	1	0

**Fig. 2.8** Gantt Chart for the running example when using $w_{cost} = 0.003$ and $w_{duration} = 0.1$. This project has a duration of 25 days and a cost of \$13,440

we can see that the allocation of employee e_1 was very helpful to reduce the duration of the project by working together with the cheaper employee e_5 on tasks t_4, t_7 and t_8 , which have a direct impact on the duration of the project due to their dependencies to each other. In terms of task t_5 , one might have thought that it would have been better to swap the allocations of employees e_1 and e_5 , so that task t_5 is performed by the cheaper employee e_5 . However this would not really have reduced the cost of the project. Indeed both employees e_1 and e_5 are working full time from Day 1 to Day 15, meaning that swapping the roles of e_1 and e_5 would not help.

The illustrative project being scheduled here is a small project, not being too challenging to allocate manually. However, even this way, a significant amount of reflections need to be done to decide who is worth allocating or not. With larger projects in companies with a larger number of employees, this challenge increases. By using an AI tool, software managers would be able to get different suggestions

with different trade-offs between cost and duration (based on different weights w_{cost} and $w_{duration}$), helping them to decide on a good schedule to adopt.

Shen et al.'s algorithm [4] also considers that the project may suffer disruptions during its execution. For example, a given employee may need a sick leave. This algorithm enables project managers to enter this information during the project, and then produces an alternative schedule that takes such disruptions into account while not causing too many changes to the original schedule. This can be very helpful to maintain the cost and duration of the project as similar to the original one as possible, while not changing the original schedule too much.

Even though the evolutionary algorithm proposed by Shen et al. [4] is more complex than the ones proposed by Minku et al. [3], the way to run the two algorithms would be similar, i.e. the tool would require software managers to input information about the employees, tasks and task precedence, and would output an allocation of employees to tasks.

2.3 Software Effort Estimation (SEE)

Software Effort Estimation (SEE) is the process of predicting the effort (e.g. in person-months) required to develop a software project. It is typically the first step of the software development process and can contribute to a successful software project management, constituting the basis for the subsequent steps such as planning, estimating cost, managing and reducing project risks [13–15].

When estimating the effort of a software project as a whole in the traditional waterfall-like software development, software requirements of the customers are elicited at the beginning of the project and typically remain the same or do not change much throughout the software development process. Therefore, SEE often takes place in the early stages of software development, and based on it project managers make important decisions such as the budget, the bidding price and the subsequent planning and control [16]. The estimation is typically made based on information regarding the project and the team intended to work on the project, such as functional size (a metric estimating the size of the software to be developed), software development type (new development, enhancement, re-development) and development team skills (low, medium, high) [17, 18].

When adopting agile software development processes, software is typically developed incrementally in iterations, catering for feedback and requirements from the customers [19]. Specifically, at the end of each developing iteration, developers would release the software at the current status to the customer requiring for feedback such as new functionalities and changes to implemented functionalities. Developers and customers then negotiate and agree on the requirements to be developed in the next iteration. In this scenario, the effort for each iteration (not for the remaining iterations of the project) is typically estimated right before starting to develop this iteration. Therefore, SEE often takes place multiple times throughout the entire process [20, 21]. Similar to the traditional development process, the estimation of an agile project

to be developed is typically made based on information regarding the project's iteration and the team assigned to work it, such as task size in this iteration, development team's skills (low, medium, high) and development team's experience (with / without prior experience on the task) [20, 22, 23].

Good effort estimation is important for software project management. Both over-and under-estimation can cause serious problems to the organisation. Over-estimation may result in a company losing bids for contracts or wasting resources. Under-estimation may lead to poor product quality, unsatisfied customers, delayed or even incomplete software systems [14, 15]. For example, NASA had to shut down its incomplete Checkout & Launch Control System project after it massively exceeded the budget [24]. Nevertheless, providing accurate effort estimations is very challenging. Project managers have mainly relied on expert judgement to produce estimations. As a result, the estimation quality is highly affected by the experience and the capability of the experts [25, 26]. Such expert-based estimation typically suffers from a number of issues: they can be costly, the prediction process is not explicit, they are not repeatable, they can be influenced by irrelevant factors such as being sensitive to political pressure, and they can have personal bias [25, 27].

Artificial Intelligence (AI) can be used for SEE as a decision support tool. Based on estimations provided by AI models, software project managers can potentially justify, criticise or adjust the estimation derived by the experts. Such model-based estimation can potentially circumvent problems faced by the experts: they are repeatable, objective, efficient and can often provide better understanding of the estimation process [15]. Therefore, AI approaches could potentially provide a more justifiable effort estimation than those decided based on 'gut feeling'. They could also be used to investigate 'what-if' scenarios, where required efforts when using different development resources, different development teams and so on could be compared to each other, enabling practitioners to make more informed planning decisions. Section 2.3.1 explains the main process of applying AI for SEE, and Sect. 2.3.2 presents an example of how to run an AI approach for SEE.

2.3.1 *AI Approaches for SEE*

Many AI approaches have been investigated for SEE. They are typically machine learning approaches. Examples of machine learning approaches applied to waterfall-like software development processes include k -nearest neighbours [28–30], linear regression [31, 32], regression tree [33], linear programming [34] and ensembles of learning machines [35–38]. Examples of machine learning approaches applied to agile software development processes are similar to that in the traditional development scenario, including decision tree, support vector machine, neural networks and ensemble of learning machines [20, 21]. There are also approaches making use of natural language processing and deep learning for effort estimation in agile software development [39, 40].

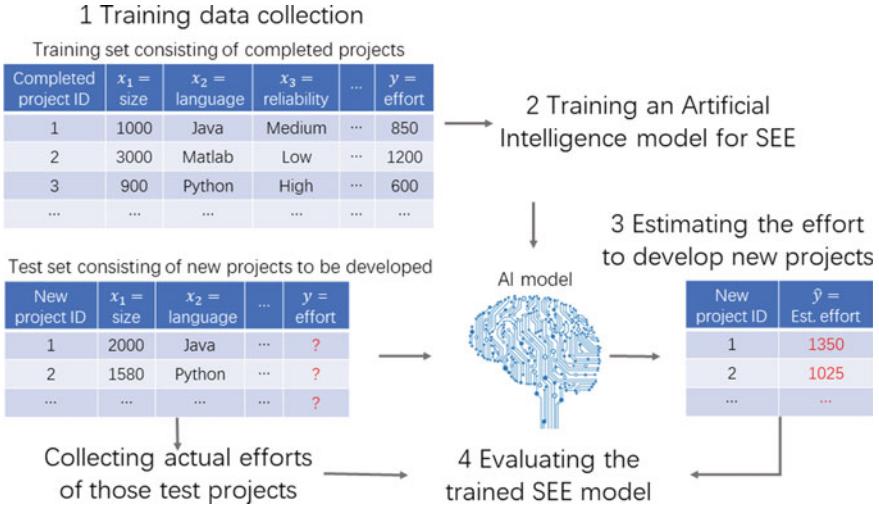


Fig. 2.9 Typical procedures of SEE from the AI viewpoint. Model adaptation is not illustrated here

This section takes the approach proposed by Song et al. [5] as an example to explain the typical procedures of using machine learning techniques for SEE. To build an AI approach for SEE, one needs to extract data features describing previously completed software projects (or agile iterations) and their project team(s), and to collect the actual effort spent to fulfil these projects. This can form a training set, based on which an SEE model can be constructed, following a machine learning algorithm. After that, software project managers can use the SEE model to estimate the effort to develop a new software project (or agile iteration). Figure 2.9 illustrates the typical process of adopting machine learning approaches to solve a given problem. Each of the steps in this process is described in the context of SEE in the rest of this section. For an explanation of machine learning approaches in general, please refer to chapter ‘Foundations of Machine Learning for Software Engineering’.

2.3.1.1 Collecting the Training Set

Features to Describe Software Projects or Agile Iterations

To use AI approaches for SEE, one needs to adopt some quantitative and qualitative metrics, as data features, to describe the software projects and their allocated development teams. Based on them, the effort required to develop the projects can be estimated. Table 2.6 lists the features of projects in Nasa93, a popular open-source data set for SEE in a waterfall-like development process [42]. It follows the famous COCOMO data format using characteristics such as the (estimated) size of the code, constraints that the software must satisfy and information about the development team allocated to the project [43]. Other features could also be adopted for SEE.

Table 2.6 COCOMO-format features of software projects in Nasa93 [41]

Feature	Description	Type	Value
loc	Line of codes	Numerical	continuous
dev	Develop type	Categorical	$\left\{ \begin{array}{l} \text{organic, embedded} \\ \text{semidetached} \end{array} \right\}$
rely	Required software reliability	Ordinal	$\left\{ \begin{array}{l} 1 = \text{very low} \\ 2 = \text{low} \\ 3 = \text{normal} \\ 4 = \text{high} \\ 5 = \text{very high} \\ 6 = \text{extra high} \end{array} \right\}$
data	Data base size	Ordinal	
cplx	Process complexity	Ordinal	
time	Time constraint for CPU	Ordinal	
stor	Main memory constraint	Ordinal	
virt	Machine volatility	Ordinal	
turn	Turnaround time	Ordinal	
acap	Analysts capability	Ordinal	
aexp	Application experience	Ordinal	
pcap	Programmers capability	Ordinal	
vexp	Virtual machine experience	Ordinal	
lexp	Language experience	Ordinal	
modp	Modern programming practices	Ordinal	
tool	Use of software tools	Ordinal	
sced	Schedule constraint	Ordinal	

For example, Kitchenham, another popular data set in SEE, uses features such as adjusted functional size, project type (A, C, D, P, Pr, U) and client code (C1, ..., C6) to describe software projects [44].

As pointed out in Usman et al.'s [20] and Marta et al.'s [21], there has been little consensus on the best features to describe software projects in different agile contexts. Nevertheless, development team skills (low, medium, high), size of the task to be estimated (a numerical value) and team's prior experience (with/without prior experience) are often highlighted as playing an important role in the estimation process in the agile context. Besides that, some work uses the textual description of user stories or issue reports as input features [39, 40].

Unit of Required Effort

To adopt AI techniques for SEE, one also needs to decide on the unit of the required effort. Required effort is typically a positive real value, measured in (e.g.) person-months. Alternatively, some agile SEE approaches measure the effort to develop user stories or issue reports in story points [39, 40].

Training Set

To build an SEE model based on machine learning, a training set containing examples of completed projects described by the input features and their actual required efforts is necessary. This is illustrated in Step 1 of Fig. 2.9. Each project described by its input features and actual required effort is referred to as a training example. Typically, SEE data sets have from around 20 to 200 examples, though larger data sets would be desirable. Such data sets usually contain projects developed by the single company for which estimations are to be provided. This means that procedures need to be put in place for this company to collect features describing its software projects and the actual effort that the development team had spent on them. However, some specific SEE approaches have demonstrated success when adopting mixed data coming from different companies [45–47].

2.3.1.2 Building SEE Models Based on the Training Set

As shown in Step 2 of Fig. 2.9, an SEE model can be constructed based on the training set by using a machine learning approach. Song et al. [5]’s study uses Relevance Vector Machines (RVMs) for this purpose, as they have shown good estimation accuracy compared to other approaches in the context of SEE [48]. An explanation of RVMs is provided in Sect. 4.1.2 of chapter “Foundations of Machine Learning for Software Engineering”.

Typically, machine learning approaches for SEE are used to provide a point estimation, i.e. to estimate a single value representing the required effort to develop a software project [15, 49]. However, as uncertainty is inherent to SEE [50, 51], point estimates make it difficult to manage risks associated to SEE, potentially leading project managers to wrong decision-making. Song et al. [5] proposed a modification to RVMs enabling them to provide not only a point estimation but also an interval of values, within which one would have a high confidence that the actual required effort will fall. This allows for risk management, helping project managers to make better-informed decisions. For example, when bidding for a project, if the competition is very fierce the project manager can report a lower price within the interval to enhance the winning chances; when the competition is less fierce, he/she can propose a higher price. Moreover, such estimation can be a more reasonable representation of reality that embraces the fact that effort estimates are probabilistic assessments of a future condition [52].

Song et al. [5]’s approach showed competitive performance compared to others able to provide prediction intervals [5]. The rest of this section thus concentrates on explaining how Song et al. [5] extended RVMs to provide prediction intervals in addition to point estimates. If the reader is not interested in learning more about the techniques behind this extension, it is possible to jump to Sect. 2.3.1.3.

The extension of RVMs is a new approach named *Synthetic Bootstrap ensemble of Relevance Vector Machines (SynB-RVM)*. The basic idea of SynB-RVM is to create several SEE RVM models by training them on different samples of the training set. The predictions given by these SEE models can be aggregated to provide a point esti-

Algorithm 2 Training algorithm for SynB-RVM.

Inputs: number of Bootstrap bags M , degree of synthetic displacement ρ .
 Output: M trained RVM models and their training errors.

- 1: *Bootstrap training bag construction*: create M Bootstrap training bags from the training set using Bootstrap re-sampling with replacement.
- 2: **repeat**
- 3: **for** each Bootstrap training bag **do**
- 4: *Synthetic project generation*: replace the repeated training projects with their synthetic counterparts. The degree of the dispersion of the synthetic project from the original one is determined by the hyperparameter ρ .
- 5: *RVM training*: train the RVM model based on the revised Bootstrap training set.
- 6: Calculate training error of the RVM model according to some performance metric.
- 7: **end for**
- 8: **until** the maximum number of Bootstrap training bags is reached

mate corresponding to the most likely required effort value, and to create an interval of other possible values around it. The training algorithm of SynB-RVM is depicted in Algorithm 2, consisting of four steps: (1) Bootstrap training bag construction, (2) synthetic project displacement, (3) training RVMs and (4) calculating the training error for each RVM model. Each of these steps is explained below.

Bootstrap Training Bag construction

SynB-RVM first creates M different samples ('bags') of the training set based on a procedure called Bootstrap re-sampling with replacement. Each bag has the same size as the original training set, meaning that some training examples from the original training set will be missing and some will appear multiple times in the bag. The number of bags M is a hyperparameter that needs to be chosen beforehand. This can be done based on classical hyperparameter tuning techniques such as k -fold cross-validation. For a discussion of existing hyperparameter tuning techniques, please see Sect. 11.3.2.3 of chapter "Foundations of Machine Learning for Software Engineering".

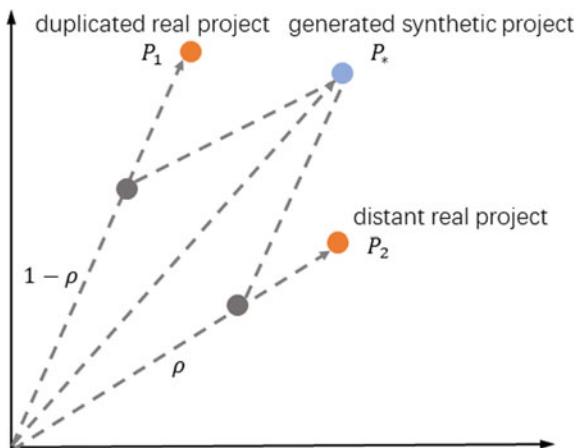
Synthetic Project Displacement

Due to sampling with replacement, each Bootstrap training bag contains duplicated training examples, which would cause invertibility problem of the kernel matrix when training RVMs [53]. To this end, the training algorithm has a procedure to generate synthetic counterparts for these repeated training examples by displacing some of their features. The effectiveness of this displacement technique was verified in [5].

Figure 2.10 illustrates the mechanism of synthetic displacement where there is only one input feature and the output effort composing two dimensions of the vector space of projects. Given a duplicated project P_1 , the synthetic project P_* is generated by displacing it along a different direction, influenced by a different project P_2 from the original training set. Specifically, the synthetic project is created based on the parallelogram formed in the vector space, as shown in this figure.

Project P_2 is chosen as the most distant project from P_1 in the training set. The reason for choosing the furthest project are two-fold: (1) this leads to a more diverse

Fig. 2.10 Illustration of synthetic project generation for a duplicated real project P_1 in a 2-dimensional data space (one for the input feature and the other for the output effort). The synthetic project P_* is generated as a linear summation following *Parallelogram Law* in the 2-D vector space based on real projects P_1 and P_2 that are scaled according to the degree of synthetic displacement ρ



Bootstrap bag which is more likely to relieve the invertibility problem and (2) disturbance along a real project will avoid the synthetic project to be too far away from the real one. For a project that has more than one replication, the learning algorithm will repeat this procedure multiple times. In this situation, rather than choosing the furthest project, the algorithm would choose the second or the third furthest project and so on.

RVM Training and Computation of Their Training Error

Based on the revised Bootstrap training bags, the training algorithm constructs M RVM models, each corresponding to one Bootstrap bag, using the RVM algorithm explained in Sect. 4.1.2 of chapter “Foundations of Machine Learning for Software Engineering”. Practically, these RVM models can be constructed in parallel to speed up the training process. The training error of each RVM is also computed by comparing the actual and the estimated effort values across the training set. Each RVM has one particular training error, which is passed on to the prediction algorithm explained in Sect. 2.3.1.3.

2.3.1.3 Estimating the Effort for New Projects to be Developed

After getting a trained SEE model, we can use it to estimate the effort required to develop new software projects, for which only input features are known. The aim is to predict their effort values. Figure 2.9 illustrates a test set containing a number of projects whose effort is to be estimated. Once the actual effort used for such projects becomes known, one can evaluate the predictive performance of the SEE model.

Given a new project to be developed, the project manager needs to first produce the values of the input features describing this project. These input features need to be the same features used for the training set. For instance, let's assume that these features are (code size=2000, language=Java, reliability = ‘high’, ...) as illustrated

in Fig. 2.9. Then, the project manager can feed these feature values to the SEE model that was built based on the procedure explained in Sect. 2.3.1.2, so that it provides an effort estimation for this project. The estimation will use the same unit (e.g. person-months) as defined at the data collection step.

Song et al.'s approach [5] provides a prediction interval associated to a confidence level. The centre of this interval corresponds to the most likely effort value. For instance, the model may predict the interval [60, 100] person-months with a confidence level of 95%, together with the most likely required effort of 80 person-months. An approach that only provides point estimates would provide a single value (e.g. 80 person-months). The rest of this section concentrates on explaining how SynB-RVM produces both a point prediction and a prediction interval with a certain confidence level. If the reader is not interested in learning more about the machine learning technique behind SynB-RVM, it is possible to jump to Sect. 2.3.1.4. Algorithm 3 demonstrates the SynB-RVM procedure of producing an effort estimation for a new project to be developed [5]. Each step of this algorithm is explained below.

Algorithm 3 Prediction algorithm of SynB-RVM.

Inputs: confidence level α , pruning rate $\tau \in [0, 1]$.

Output: prediction interval with respect to the given α .

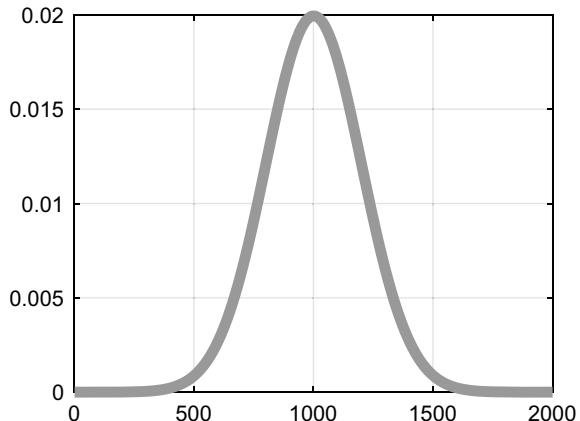
- 1: **repeat**
 - 2: Produce the values of input features describing test projects.
 - 3: **for** each test project **do**
 - 4: *Multiple probabilistic prediction:* employ the M RVM models constructed by the training algorithm to produce M Gaussian PDF predictions.
 - 5: *Model pruning:* prune these RVMs with (a) negative estimated mean values and (b) unsatisfactory training performance according the pruning rate τ .
 - 6: *Integrating the remaining estimates:* integrate the prediction of the remaining RVMs into a unified prediction result.
 - 7: *Construct prediction interval in line with α :* derive the required prediction interval and the most likely point estimate based on the integrated prediction.
 - 8: **end for**
 - 9: **until** all test project are predicted
-

Multiple Probabilistic Prediction

The output of an RVM model when estimating the effort of a new project comes in the format of a Probability Density Function (PDF) of the possible effort values to develop the new project. Specifically, it follows a Gaussian distribution [53]. Figure 2.11 shows a probabilistic (Gaussian) effort estimation for a new project. The x-axis represents the effort and the y-axis represents the relative likelihood of observing different effort values. The corresponding point estimation is the mean of the Gaussian (1000 person-months), as it is the most likely effort value. When using SynB-RVM for effort estimation, we would get M predicted Gaussian PDFs, based on which the prediction interval with a certain confidence level is produced.

Model Pruning

Fig. 2.11 Illustration of a Gaussian distribution from RVM as the estimations of the possible effort required to develop a software project. The y-axis represents the relative likelihood of effort values



The mean values of the Gaussian PDFs produced by some RVMs may be improperly negative, whereas effort values cannot be negative. This happens because each RVM itself is a weak model, which may not perform so well. SynB-RVM will combine such weak models together to produce a strong, better model. However, it is prudent to eliminate RVMs that produce negative means when estimating a given project. The remaining RVMs are ranked based on their training predictive performance, and the worst τ percentage models are also pruned. This pruning prevents poor models from contributing towards the final estimation given by SynB-RVM.

Project managers can choose the pruning performance metric that best reflects their preference, based on the practical meaning of those metrics. Mean absolute error was recommended as the default performance metric for being symmetric and having no bias against over-/under-estimation [5].

Combining Uncertain Predictions

To generate the final estimation, we need to combine the PDFs provided by the remaining M' RVMs into a unified one. As Gaussian distribution is uniquely determined by its mean and variance, this issue can be simplified into combining M' pairs of mean-variance. Song et al. [5] proposed three slightly different ways to derive the final probabilistic prediction on the test project, namely *empirical mean*, *uni-variate empirical PDFs* and *bi-variate empirical PDFs*. We discuss the version of applying *bi-variate empirical PDFs* in this book chapter as it can usually produce better prediction intervals when one has got an adequately large number of training projects. For a full description, please refer to [5].

To combine the PDFs based on the bi-variate empirical PDF method, the algorithm first creates a 2-dimensional frequency histogram $f(i, j)$ of the mean and standard deviation of the Gaussians outputted by all remaining RVM models. This can be implemented by applying MATLAB function *histcounts2()*, by which the numbers of bins are automatically determined by the binning algorithm to cover the data range and reveal the shape of the underlying distribution. Then, the geometric middle point $(y(i), \sigma(j))$ of the mean and standard deviations within each rectangle bin of the

histogram is computed. The frequency $f(i, j)$ of each bin is also computed. Finally, the sum of the middle points multiplied by the frequencies is used as the mean y and standard deviation σ of the combined PDF, which is calculated as

$$(y, \sigma) = \sum_{i,j} (y(i), \sigma(j)) \cdot f(i, j)$$

Construct Prediction Interval

Denote \hat{y} as the final estimated mean and $\hat{\sigma}^2$ as the final estimated variance of the Gaussian PDF. Prediction intervals with respect to confidence levels of 68%, 95% and 99.7% can be easily derived as

$$[\max(0, \hat{y} - j\hat{\sigma}), \hat{y} + j\hat{\sigma}],$$

according to the ‘68-95-99.7’ rule of Gaussian distribution [54], where $j = 1, 2, 3$ corresponds to confidence levels 68%, 95% and 99.7%, respectively. For example, if the predicted final mean is 80 person-months, the most likely effort value is 80 person-months. If the variance is 5^2 , the project manager would have 95% confidence level that the actual effort of the test project falls within [70, 90] person-months.

Deriving prediction intervals with respect to other confidence levels is a bit more complex. An explanation can be found in [5]. It is worth noting that the whole training and prediction process of machine learning algorithms can be automated, i.e. the derivation of the prediction intervals does not need to be done manually by the software project manager as will be shown in Sect. 2.3.2.

2.3.1.4 Evaluating the Predictive Performance of an SEE Model

Once software projects are completed, their actual required effort can become known, so long as the software developers working on the project are recording information on how much effort they have spent on the project. Projects that have not been used to train the predictive model can be used to evaluate this model once their actual required efforts become known. Evaluating the model is important so that the software project manager has some idea of how good the estimations given by this model are expected to be. In particular, before starting to use a given model in practice, it is worth evaluating it on a number of completed projects that have not been used for training. This section discusses some popular performance metrics for evaluating models that produce point estimations and prediction intervals.

Metrics for Point Effort Estimation

There are several performance metrics for evaluation of point effort estimation. Popular examples are mean/median magnitude of relative error, mean/median absolute error, and percentage of estimates within $p\%$ of actual values.

Magnitude of Relative Error (MRE) [28, 55] measures the error ratio between an actual effort y_i and its corresponding estimated point effort \hat{y}_i as

$$MRE_i = |\hat{y}_i - y_i|/y_i.$$

The smaller the MRE_i , the better the prediction performance of the SEE model performing on this project. For example, if an SEE method predicts that the development team will take 120 person-months to develop a project and it turns out that the team has spent 100 person-months in reality, MRE is $|120 - 100|/100 = 20\%$ meaning that this prediction is 20% away from the actual effort in magnitude.

When computing $\{MRE_i\}$ for several projects, one can compute the Mean MRE (MMRE) or Median MRE (MdMRE) to get an idea of the typical relative error of the model. The advantage of these metrics is that they are interpretable as a percentage. However, they can be biased towards predictive models that underestimate effort, i.e. a model that underestimates effort has a lower MMRE or MdMRE than another model that overestimates effort by the same amount [55–58].

Absolute Error (AE) measures the difference in magnitude between an actual effort y_i and its estimated point effort \hat{y}_i as

$$AE_i = |\hat{y}_i - y_i|.$$

The smaller the AE_i , the better the prediction performance of the SEE model performing on this project. For the same example having the predicted and actual efforts of 120 and 100 person-months, AE is $|120 - 100| = 20$ person-months, meaning that this prediction has a difference of 20 person-months from the actual effort.

When computing $\{AE_i\}$ for several projects, one can compute the Mean AE (MAE) or Median AE (MdAE) to get an idea of the typical absolute error of the model. MAE has been recommended by Shepperd and MacDonell for SEE studies, for being a symmetric metric and not biased towards under or overestimation [58]. For instance, ‘MAE=250 person-months’ means that on average, the predicted effort would be larger / smaller than the actual value by a magnitude of 250 person-months. MdAE has shown to be less sensitive than MAE to occasional projects with very large efforts and is thus a useful addition to MAE [55]. But MdAE is less straightforward than MAE to be interpreted. One of the disadvantages of MAE and MdAE is that they cannot be interpreted as a percentage.

In summary, different performance metrics emphasise different factors and can behave differently in evaluating point effort estimation models [38]. It is highly unlikely to exist a single, simple-to-use and universal goodness-of-fit performance metric for SEE [55]. In practice, practitioners need to choose the performance metrics according to their particular emphasis and preferences.

Metrics for Uncertain Effort Estimation

Prediction intervals should be wide enough to capture the actual effort and at the same time sufficiently narrow to be informative for practical use. Therefore, metrics to evaluate the quality of prediction intervals need to take this into account. The following two metrics are typically used for that purpose.

Hit rate is the most common evaluation metric for prediction intervals [59, 60]. The idea is as follows. If prediction intervals with confidence level α are evaluated

based on N software projects, it is expected that around $\alpha \cdot N$ projects have actual efforts falling within the corresponding prediction intervals. Therefore, the hit rate can be calculated by first counting the number of projects whose estimated efforts are within the prediction intervals, and then dividing that by the total number of projects.

When the number of test examples is sufficiently large, the obtained hit rate should be around the chosen confidence level. When the hit rate is higher, the estimated prediction intervals are too wide; otherwise, the estimated prediction intervals are too narrow. For example, consider that there are plenty of test projects to evaluate an SEE model for a confidence level of 90%. If the hit rate is as low as 60%, this means that only 60%, rather than the desired 90%, of the actual efforts were within the prediction interval. This indicates that the model is not performing well for having too narrow prediction intervals or/and cannot achieve good point estimations. It is worth noting that due to the small SEE data sets, we usually do not have sufficient test examples. Hence, hit rate may deviate from its corresponding confidence level although the two values could be very close in essence.

Relative width is another useful evaluation metric for prediction intervals [61]. From two sets of prediction intervals with similar hit rates, the set with the narrower intervals is more informative. For example, given that two SEE methods can produce a similar hit rate of 90%, method *A* has an average relative width of 1.5 and method *B* has an average relative width of 2.2, method *A* can be considered better than method *B* for providing narrower (more informative) prediction intervals.

Given a prediction interval of a project, the relative width can be calculated by first computing the ‘width’ of the prediction interval by subtracting the lower bound from the upper one, and then dividing that by the estimated point estimation that is the most likely to happen. For example, given a prediction interval of [500, 1000] person-months, if the most likely effort is 750 person-months, the relative width can be calculated as $(1000 - 500)/750 \approx 0.67$. This means that the width of the prediction interval was around 67% of the point estimation. This quantifies how informative the prediction interval is. One can compute the average relative width for the prediction intervals estimated for a given test set as a way to quantify the average amount of information of such prediction intervals.

It is important to note, though, that wider intervals may have better hit rates, whereas narrower intervals may have lower hit rates. Therefore, if two methods have different hit rates, the relative widths of their prediction intervals are not comparable. It is only possible to compare the relative widths of the intervals provided by two estimation approaches if their hit rates are similar.

2.3.1.5 Updating the SEE Model Based on New Incoming Data

Whenever a new project is completed and information on its actual effort is collected, a new example can be created. As mentioned in Sect. 2.3.1.4, these examples can be used to evaluate the predictive performance of the SEE model. Some approaches have also been proposed to further update this SEE model based on new examples, after such examples are used for evaluation purposes. In particular, the approaches

by Minku et al. [45] have been proposed to adapt SEE models to changes suffered by software companies and that may affect such models.

SynB-RVM is a typical offline effort estimator. To be able to incorporate new training examples into it, it needs to be re-trained from scratch with a new training set that includes the new completed projects.

2.3.2 *Running an AI Algorithm for SEE*

This section gives an example of how to run an AI algorithm for SEE. Specifically, we employ SynB-RVM proposed by Song et al. [5] as an example to demonstrate how to produce effort estimations for software projects.

The tool is implemented in Matlab and is available at [62] under GNU GLP 3.0 license. Once the tool is downloaded, there is a folder called *matlab*, containing the codes implementing this tool. Another folder called *data_example* contains the edited software projects for SEE produced based on Nasa93 [42].

To use the tool, one needs to run the script *config.m* or typing in the command window the following line:

```
>> config()
```

This function configures the directories between code scripts and the data set and among all scripts, so that the scripts can call each other and load the data set as if they are in a single one-layer directory. An example of running the overall implementation of SynB-RVM is given in *example_run_SynB_RVM.m* in the directory ‘*matlab/examples/*’. This example will be explained in the next subsections.

2.3.2.1 **Preparing the Training and Test Projects**

In the real-world, practitioners need to collect information of completed software projects (e.g. the features in Table 2.6) and the efforts spent to complete the development in their own organisations, making up the training set. The SEE model is then constructed based on this training set. Later on, when there comes a new project to be developed, the practitioner describes this project by the same features and applies the trained model to estimate the effort. This corresponds to the training and prediction processes of SEE as described in Sects. 2.3.1.2 and 2.3.1.3.

Our tool was implemented for research purposes, making use of text files to collect information about the features describing software projects and their actual efforts. In our running example, we use nasa93, a data set from the open-source repository SEACRAFT [17] (<https://zenodo.org/record/268419#.YLRQyKgzYUE>), to demonstrate the whole process.

To simulate a real-world SEE scenario, the original Nasa93 is split into a training and a test set according to the time order of the software projects. As

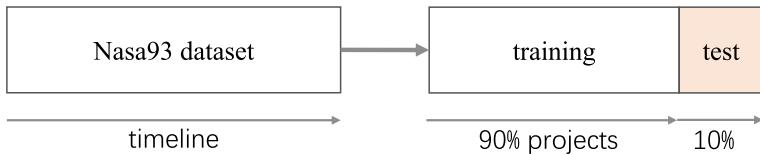


Fig. 2.12 Split Nasa93 into the training and test sets according to the project timeline, described as ‘year of development’ in the title file

shown in Fig. 2.12, 90% projects that completed earlier are used as the training data and the remaining 10% projects are used as the test examples to showcase point and interval predictions produced by SynB-RVM. The 17 features used to describe projects of Nasa93 are shown in Table 2.6 and are in the tool’s folder `data_example/nasa93_edited`. The suffix `edited` means that the data set has been revised to be used with Matlab. For example, the feature ‘stor’ (main memory constraint) with the values of `v1`, 1, `n`, `h`, `vh` and `xh` are converted to the values of 1, 2, 3, 4, 5 and 6, respectively.

To obtain the data division into training and test sets in one step, one can run `get_nasa93.m` or type in the command window the following line:

```
>> [X_train, y_train, X_test, y_test] = get_nasa93()
```

where `X_train` and `X_test` contain the input features of the training and test examples, respectively, and `y_train` and `y_test` contain their actual required efforts in person-months. Through `get_nasa93.m`, the training and test examples are pre-processed to prune out three outliers and to conduct feature normalisation. If one would like to run the pre-processing separately, this process is implemented in `data_preprocess.m` and can be run in the command window by typing the following line:

```
>> [years, X, y] = data_preprocess(Data);
```

where `Data` denotes the original projects of Nasa93, `years` denotes the time of completing each project and `X` and `y` are their features and actual effort for development.

2.3.2.2 Using the Training and Prediction Algorithms of SynB-RVM

The training and prediction algorithms of SynB-RVM (Algorithms 2 and 3, respectively) are implemented in `SynB_RVM.m` in the directory ‘`matlab/train-prediction/`’. Script `example_run_SynB_RVM.m` in the directory ‘`matlab/examples/`’ exemplifies the usage of the training and prediction algorithms, producing point and interval

predictions with a confidence level of 85% for the projects in the test set created in Sect. 2.3.2.1. Type the following command to run this script:

```
>> example_run_SynB_RVM()
```

The predicted point and interval prediction may improve or deteriorate with different hyperparameter settings. Section 2.3.2.5 shows how to tune hyperparameters. The hyperparameter setting of SynB-RVM adopted in this example is shown in the script *example_run_SynB_RVM.m* as

```
m_bags = 10;           % #(Bootstrap Bags)
pru_m = 0.1;          % prune rate
tho = 0.01;           % synthetic displacement
```

This script prints the results below:

Prediction intervals with CL 0.85 of test projects in Nasa93 are:					
id	actual	point estimate	prediction interval	hit/not	
1	210.00	233.33	109.37 - 357.29	1	
2	48.00	133.33	1.38 - 265.29	1	
3	50.00	55.00	0.00 - 162.96	1	
4	60.00	65.00	0.00 - 172.96	1	
5	42.00	122.22	6.26 - 238.18	1	
6	60.00	144.44	28.48 - 260.41	1	
7	444.00	516.67	408.70 - 624.63	1	
8	42.00	144.44	28.48 - 260.41	1	
9	114.00	183.33	67.37 - 299.30	1	

The second column presents actual efforts of test projects. These values would not have been available in practice at the prediction stage. They would become available only after the project finishes, if effort information is collected during the development process. The third column reports the point effort estimations. The fourth column reports the prediction intervals with a confidence level of 85%. The last column reports whether or not the prediction interval covers the actual effort for each test project. Again, this last column would not be available in practice at prediction time, only after the project is completed.

Let's take the seventh test project as an example of project being estimated. Its most probable point effort is 516.67 person-months. With 85% confidence, the actual effort falls within the interval of 408~625 person-months. This interval captures reasonably well the actual required effort for this project. Some interval predictions catch the lowest bound of the effort value (0 person-month), such as the third and the fourth test projects. In such cases, this often indicates that the actual effort values has higher probability to be a smaller value rather than a larger value.

2.3.2.3 Evaluating the Performance of a SynB-RVM Model

In practice, completed projects with known effort that have not been used for training purposes can be used to evaluate SEE models. Such performance evaluation for the point estimations and interval predictions are implemented in *eval_point_pf.m* and *eval_PI_pf.m* in the directory ‘matlab/evaluate/’, respectively. They will make use of the test set created in Sect. 2.3.2.1 for evaluation purposes. The evaluation of point predictions can be run with the following command:

```
>> eval_point_pf(y_true, y_pred)
```

The inputs *y_true* and *y_pred* are column vectors consisting of the true and predicted efforts of test projects, respectively. The estimated point efforts are obtained from the function *example_run_SynB_RVM()* as discussed in Sect. 2.3.2.2, and the actual efforts are collected when the projects finish and the effort spent on the development has been recorded. This function prints point prediction performance of SynB-RVM as shown below. Some of these performance metrics have been explained in Sect. 2.3.1.4. For a description of the other metrics, we refer the reader to [55].

```
ans = structures consisting of
      mae: 57.8214
      mdae: 82.3365
      mlgae: 3.5704
      mdlgae: 4.4108
      mmre: 0.9730
      mdmre: 0.8201
      pred25: 44.4444
      pred15: 44.4444
      pred10: 22.2222
      coor: 0.9536
      lsd: 0.6875
      rmse: 69.2337
      rmdse: 82.3365
      sa: 0.4656
```

The results mean that, the point estimations were around 57.82 person-months away from the actual required efforts on average (MAE), and that the point estimations were 97.30% away from the actual effort in magnitude on average (MMRE).

To evaluate the prediction intervals measured in *hit rate* and *relative width* as explained in Sect. 2.3.1.4, the following command can be used:

```
>> [hit_rate, relative_width] = eval_PI_pf(PI, y_pre_mean, y_true)
```

The input *PI* is a data matrix where each row represents the prediction interval of a test project and *y_pre_mean* is a column vector consisting of the point effort estimations. The output includes the information below:

```
hit_rate = 1
relative_width = 1.6957
```

The results mean that all prediction intervals covered the actual efforts, and that the widths of prediction intervals were around 170% of the value of the point estimates on average.

2.3.2.4 Using SynB-RVM in What-If Scenarios

Software managers could adopt AI SEE methods such as SynB-RVM to investigate what-if scenarios helping them to make more informed decisions. For example, assume a scenario where the project manager is considering whether to develop a software application that is as much space- and time-efficient as possible. However, they are unsure on whether this should be done, given the increase in effort that this could result in. As it is challenging to estimate the extent with which the effort would increase, artificial intelligence SEE models may be particularly helpful.

To investigate projects with higher computational and storage constraints, we can increase the values of the resource related features of test projects in Nasa93, including *data base size constraint*, *time constraint for cpu* and *main memory constraint*. For example, we can increase them to the highest value *xh* (extremely high) while retaining other features. The generation of such test projects is implemented in *get_nasa93_resource.m* in the directory ‘matlab/examples/’, and can be run by typing the command below:

```
>> get_nasa93_resource('highest')
```

After that, we can observe SynB-RVM’s effort estimations for these modified projects. The experiment is implemented in *experiment2_resource.m* in the same directory. One can run it by typing the following command:

```
>> experiment2_resource('highest')
```

where the input *highest* means that the highest resource constraint is evoked. Point and interval predictions of SynB-RVM are shown below:

```
=====
SynB-RVM_ht2d's prediction on the test projects in Nasa93 that
are reconstructed with the highest resource constraint.
Prediction intervals are with confidence level of 0.85.
id predicted prediction interval
 1   805.56    689.59 -    921.52
 2   472.22    372.25 -    572.19
 3   516.67    424.70 -    608.64
 4   516.67    424.70 -    608.64
 5   638.89    522.93 -    754.85
 6   527.78    411.82 -    643.74
 7   750.00    642.04 -    857.96
 8   527.78    411.82 -    643.74
 9   527.78    411.82 -    643.74
```

Comparing this against the previous estimations from *example_run_SynB_RVM.m*, SynB-RVM generally predicts that much higher efforts are required to develop a project with extremely high computational and storage constraints than to develop a similar project with less constraints. These estimations can help the software manager in deciding whether it is worth going ahead with such extreme constraints, given the increase in effort. Note that these projects have not been developed with such extreme constraints in practice. So, we cannot determine how accurate these estimations are.

If we assume another extreme scenario where the project manager is considering to have very light computational and storage constraints. In this scenario, we would take the lowest constraints on the database size, time and memory constraint features of Nasa93 projects. This can be implemented by running the function below:

```
>> get_nasa93_resource('lowest')
```

We can then use SynB-RVM to estimate the efforts required to develop those projects with the least computational and storage constraints, by running the command below:

```
>> experiment2_resource('lowest')
```

where the input *lowest* means that the lowest constraint of the development resource is evoked. The point and interval predictions of SynB-RVM are shown below:

```
=====
SynB-RVM_ht2d's prediction on the test projects in Nasa93
that are reconstructed with the lowest resource constraint.
Prediction intervals are with confidence level of 0.85.
```

id	predicted	prediction	interval
1	166.67	50.70	- 282.63
2	128.57	10.32	- 246.82
3	55.00	0.00	- 162.96
4	65.00	0.00	- 172.96
5	125.00	8.04	- 241.96
6	144.44	36.48	- 252.41
7	455.56	347.59	- 563.52
8	125.00	17.04	- 232.96
9	144.44	36.48	- 252.41

Comparing this against the previous estimations from *example_run_SynB_RVM.m*, SynB-RVM generally predicts that lower efforts are required to develop a project with light computational and storage constraints than to develop a similar project with higher constraints when the same development team is present. These estimations can help the project manager in deciding whether it is worth adopting lighter constraints.

Now assume a scenario where the project manager would like to give medium (nominal) computational and storage constraints. This can be achieved by replacing the resource related features of the projects in Nasa93 with the *nominal* value. We can implement this by running:

```
>> get_nasa93_resource('balanced')
```

We can then use SynB-RVM to estimate the efforts required to develop those projects with the balanced constraint by running the below command:

```
>> experiment2_resource('balanced')
```

where the input balanced means that a nominal constraints are evoked. Point and interval predictions of SynB-RVM are shown below:

```
=====
SynB-RVM_ht2d's prediction on the test projects in Nasa93
that are reconstructed with the balanced resource constraint.
Prediction intervals are with confidence level of 0.85.
```

id	predicted	prediction	interval
1	433.33	301.38	- 565.29
2	188.89	72.93	- 304.85
3	277.78	169.81	- 385.74
4	300.00	192.04	- 407.96
5	383.33	267.37	- 499.30
6	283.33	167.37	- 399.30
7	616.67	508.70	- 724.63
8	300.00	192.04	- 407.96
9	316.67	200.70	- 432.63

Comparing this against the estimations produced by `experiment2_resource('highest')` and `experiment2_resource('lowest')`, SynB-RVM generally predicts that the required efforts to develop a project with nominal resource constraint are smaller (larger) than that to develop a similar project with lowest (highest) constraint on the resources when the same development tea is present.

2.3.2.5 Choose Hyperparameter Values for SynB-SVM

To use SynB-RVM, one needs to pre-define three hyperparameters: the number of Bootstrap bags M , the degree of synthetic displacement ρ in the training algorithm (Algorithm 2), and the pruning rate τ in the prediction algorithm (Algorithm 3).

In the previous explanation on how to use the tool, we showed experimental results by running SynB-RVM with the default hyperparameter setting for Nasa93. Those values were decided based on our experience and preliminary experiments. In practice, practitioners may be unaware of the theoretical mechanisms behind the approach and find it hard to decide what hyperparameter values to adopt in order to obtain good effort estimations. This section aims to demonstrate how to tune the hyperparameters of SynB-RVM. Choosing the *kernel width*, a hyperparameter specific to RVM, is not included in this process to save computational resources. However, it can be tuned based on a similar procedure to the one shown below.

The hyperparameter tuning process is implemented in the directory `/matlab/example-para_tune/`. Hyperparameter values investigated in `experiment_para_tune.m` are in Table 2.7. Their default values are emphasised in bold and correspond to the hyperparameters that can usually lead to good predictive performance accord-

Table 2.7 Hyperparameter Values Investigated in *experiment_para_tune.m*

Hyperparameter	Values	Description
M	10 , 20, 30	The number of Bootstrap bags in the training algorithm
ρ	0.01 , 0.05, 0.1	The degree of synthetic displacement in the training algorithm
τ	0.1 , 0.2, 0.4	The pruning rate in the prediction algorithm

ing to the theoretical meaning of these hyperparameters and also to our experiences of using this approach. Hyperparameter settings are composed by enumerating all values for each hyperparameter with all the others set to their default values. Thus, we have 9 hyperparameter settings in total. We believe that the values shown above form a good range of each of the hyperparameters. We run SynB-RVM with all the hyperparameter settings to calculate the performance on the validation set, which is a set of completed projects with known required efforts and that have not been used for training the predictive model. From that we can determine the best hyperparameter setting in terms of point prediction.

Given a combination of hyperparameter values, in this example, we will randomly select 90% of the projects in the training set to construct the SynB-RVM model, and the remaining 10% to compose the *validation* set on which this hyperparameter configuration will be evaluated. From that, we can evaluate the (point) predictive performance regarding this specific hyperparameter setting. This process is conducted ten times to cancel out the randomness of the process of splitting the training set into a training and a validation set. The average (point) predictive performance is used as the indicator of how good this hyperparameter configuration is. Finally, the hyperparameter configuration that leads to the best predictive performance on the validation set is selected. The ultimate SynB-RVM model to be adopted in the next prediction process is produced based on the entire training set with the chosen hyperparameter configuration.

To run the approach, the following command can be used:

```
>> experiment_para_tune()
```

The best hyperparameter configuration decided by this experiment, the point and interval predictions of SynB-RVM with the chosen hyperparameter setting and their predictive performance are shown below:

```
=====
The best hyperparameter setting based on this experiment is as below:
    the best number of Bootstrap bags is 30
    the best degree of synthetic displacement is 0.01
    the best pruning rate is 0.1
=====
```

```
=====
Predictive performance of "Nasa93" with SynB_RVM_ht2d.
```

```
-----
Overall predictive performance is as below:
    mae = 64.1
    hit_rate = 1.00
    relative width = 1.88
-----
Prediction intervals of CL=0.85 of all test projects are as:
    id      actual     predicted   prediction interval      hit/not
    1       210.00     244.00     101.49 -      386.51          1
    2       48.00      166.67     17.57 -      315.76          1
    3       50.00      56.11      0.00 -      182.74          1
    4       60.00      67.78      0.00 -      194.40          1
    5       42.00      137.50     0.00 -      275.46          1
    6       60.00      146.15     7.74 -      284.57          1
    7       444.00     492.59     360.64 -      624.55          1
    8       42.00      140.00     0.37 -      279.63          1
    9       114.00     196.30     53.68 -      338.92          1
```

Further Considerations

This section presents further considerations when adopting a SEE approach such as SynB-RVM:

1. The data quality of the organisation is the upmost important factor for producing a good software effort estimation. If the data is of low quality (e.g. large amounts of noise) and not representative for the current status of software development process (e.g. the data is very much obsolete), it is highly unlikely to attain good effort estimations regardless of how intelligent an approach could be.
2. SynB-RVM is suitable to be adopted for interval prediction only when it is found to achieve good point prediction performance. Otherwise, the prediction intervals provided by this approach would be less reliable. This is valid to most approaches that can provide uncertain effort estimation as the preciseness of the prediction intervals typically depend on the point prediction, being an interval around it.
3. Among the hyperparameters of SynB-RVM, the kernel width of RVM is typically the most important. If the computational resources for hyperparameter tuning are limited, the value for this hyperparameter can be decided independent of the others and can be processed beforehand, by adopting the same procedures discussed in this section.
4. A larger number of Bootstrap bags M probably leads to more computational cost for training the predictive model, as more base learners need to be learned (though they can be trained in parallel to save computational time).
5. The assignment of degree of pruning rate τ should be related to the number of Bootstrap bags M . Given a large M , one can assign τ to as big as 0.4 because there are potentially more RVMs to be pruned out. For a small M , practitioners are suggested to confine τ to be less than 0.2 for retaining enough base learners.

2.4 Conclusion

SPS and SEE are tasks that play important roles in the software project management process, for which AI can provide a useful tool to support project managers for better decision-making. This chapter provided an introduction to these tasks and to AI approaches that can be used to support software managers in carrying them out.

We explained how the SPS task can be formulated so that AI approaches can be used to solve it, based on the work of Alba and Chicano [2] and Shen et al. [4]. Alba and Chicano [2] provide a simpler formulation which provides a good platform to start learning about the subject. Shen et al. [4] provide a more detailed problem formulation that takes into account realistic aspects of software projects such as uncertainties in the task required efforts and changes such as new tasks or employee's leaves. We then introduced the algorithm proposed by Minku et al. [3] as an example of AI algorithm able to solve SPS based on Alba and Chicano [2]'s problem formulation. Shen et al. [4] proposed another AI algorithm able to cope with uncertainties and changes that may occur during software development, being a good next approach to learn after understanding the approach explained in this chapter.

We took the approach proposed by Song et al. [5] as an example to explain the typical procedures for using machine learning techniques for SEE, consisting of four procedures as (1) collecting training set, (2) building prediction models based on the training set, (3) estimating the effort for new project and (4) evaluating predictive performance of the SEE model. One of the distinctive characteristics of Song et al. [5]'s approach is its ability to provide not only point estimates of required effort but also a prediction interval, within which one would have a high confidence that the actual required effort will fall. This kind of approach can help software managers to make better-informed decisions. This approach can also be used to investigate the 'what-if' scenarios as discussed in this chapter. When choosing an AI-based SEE approach to adopt in a software development company, we recommend evaluating the predictive performance of a number of different AI-based SEE approaches, to check which of them is better suited to the specific context of this company. For an overview of other AI-based SEE approaches, we refer the reader to [63].

References

1. I. Sommerville, *Software Engineering*, 10th edn. (Pearson, USA, 2016)
2. E. Alba, J.F. Chicano, Software project management with GAs. *Inf. Sci.* **177**, 2380–2401 (2007)
3. L.L. Minku, D. Sudholt, X. Yao, Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis. *IEEE Trans. Softw. Eng.* **40**(1), 83–102 (2014)
4. X. Shen, L.L. Minku, R. Bahsoon, X. Yao, Dynamic software project scheduling through a proactive-rescheduling method. *IEEE Trans. Softw. Eng.* **42**(7), 658–686 (2016)
5. L. Song, L.L. Minku, X. Yao, Software effort interval prediction via Bayesian inference and synthetic Bootstrap resampling. *ACM Trans. Softw. Eng. Methodol.* **28**(1), 1–46 (2019)

6. M. Di Penta, M. Harman, G. Antoniol, The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. *Softw. Pract. Exp.* **41**(5), 495–519 (2011)
7. F. Ferrucci, M. Harman, F. Sarro, Search-based software project management, in *Software Project Management in a Changing World*, ed. by G. Ruhe, C. Wohlin (Springer, Berlin Heidelberg, 2014), pp. 373–399
8. G. Antoniol, M. Di Penta, M. Harman, A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty, in *International Symposium on the Software Metrics* (2004), pp. 172–183
9. W.-N. Chen, J. Zhang, Ant colony optimization for software project scheduling and staffing with an event-based scheduler. *IEEE Trans. Softw. Eng.* **39**(1), 1–17 (2013)
10. F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Addison Wesley, Boston, 1995)
11. L.L. Minku. minkull/SoftwareProjectScheduling (2022). <https://doi.org/10.5281/zenodo.6308397>
12. M. Lukasiewycz, M. Gläß, F. Reimann, J. Teich, Opt4J - a modular framework for meta-heuristic optimization, in *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)* (Dublin, Ireland, 2011), pp. 1723–1730
13. A. Trendowica, R. Jeffery, *Software Project Effort Estimation: Foundations and Best Practice Guidelines for Success* (Springer, 2014)
14. J. Wen, S. Li, Z. Lin, Y. Hu, C. Huang, Systematic literature review of machine learning based software development effort estimation models. *Inf. Softw. Technol. (IST)* **54**(1), 41–59 (2012)
15. K. Dejaeger, W. Verbeke, D. Martens, B. Baesens, Data mining techniques for software effort estimation: a comparative study. *IEEE Trans. Softw. Eng. (TSE)* **38**(2), 375–397 (2012)
16. B. Baskeles, B. Turhan, A. Bener, Software effort estimation using machine learning methods, in *International symposium on Computer and Information Sciences* (2007), pp. 1–6
17. T. Menzies, R. Krishna, D. Pryor, The SEACRAFT repository of empirical software engineering data (2017). <https://zenodo.org/communities/seacraft>
18. K. Maxwell, *Applied Statistics for Software Managers* (Prentice Hall PTR, 2002)
19. P. Abrahamsson, R. Moser, W. Pedrycz, A. Sillitti, G. Succi, Effort prediction in iterative software development processes – incremental versus global prediction models, in *International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2007), pp. 344–353
20. M. Usman, E. Mendes, F. Weidt, R. Britto, Effort estimation in agile software development: a systematic literature review, in *International Conference on Predictive Models and Data Analysis in Software Engineering (PROMISE)* (2014), pp. 82–91
21. M. Fernandez-Diego, E.R. Mendez, F. Gonzalez-Ladron-De-Guevara, S. Abrahao, E. Insfran, An update on effort estimation in agile software development: a systematic literature review. *IEEE Access* **8**, 166768–166800 (2020)
22. S. Keaveney, K. Conboy, Cost estimation in agile development projects, in *European Conference on Information Systems(ECIS)* (2006)
23. N.C. Haugen, An empirical study of using planning poker for user story estimation, in *AGILE Conference* (2006), pp. 21–31
24. Spareref. NASA to shut down checkout & launch control system (2002). <http://bit.ly/eiYxlf>
25. M. Jørgensen, A review of studies on expert estimation of software development effort. *J. Syst. Softw. (JSS)* **70**(1), 37–60 (2004)
26. M. Jørgensen, Forecasting of software development work effort: evidence on expert judgement and formal models. *Int. J. Forecast.* **23**(3), 449–462 (2007)
27. M. Jorgensen, S. Grimstad, The impact of irrelevant and misleading information on software development effort estimates: a randomized controlled field experiment. *IEEE Trans. Softw. Eng. (TSE)* **37**(5), 695–707 (2011)
28. M. Shepperd, C. Schofield, Estimating software project effort using analogies. *IEEE Trans. Softw. Eng. (TSE)* **23**(12), 736–743 (1997)
29. Y. Li, M. Xie, T. Goh, A study of project selection and feature weighting for analogy based software cost estimation. *J. Syst. Softw. (JSS)* **82**(2), 241–252 (2009)

30. E. Kocaguneli, T.J. Menzies, Exploiting the essential assumptions of analogy-based effort estimation. *IEEE Trans. Softw. Eng. (TSE)* **38**(2), 425–438 (2012)
31. P.A. Whigham, C.A. Owen, S.G. Macdonell, A baseline model for software effort estimation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **24**(3), 20:1–20:11 (2015)
32. B. Kitchenham, E. Mendes, Why comparative effort prediction studies may be invalid, in *International Conference on Predictor Models in Software Engineering (PROMISE)* (2009), pp. 4:1–4:5
33. C. Briand, E. Emam, D. Surmann, I. Wieczorek, D. Maxwell, An assessment and comparison of common software cost estimation modelling techniques, in *ICSE* (New York, USA, 1999), pp. 313–322
34. F. Sarro, A. Petrozziello, Linear programming as a baseline for software effort estimation. *ACM TOSEM* **27**(3), 12.1–12.28 (2018)
35. P. Braga, A. Oliveira, G. Ribeiro, S. Meira, Bagging predictors for estimation of software project effort, in *International Joint Conference on Neural Networks (IJCNN)* (2007), pp. 1595–1600
36. L.L. Minku, X. Yao, Ensembles and locality: insight on improving software effort estimation. *Inf. Softw. Technol. (IST)* **55**(8), 1512–1528 (2012)
37. E. Kocaguneli, T. Menzies, J. Keung, On the value of ensemble effort estimation. *IEEE Trans. Softw. Eng. (TSE)* **38**, 1403–1416 (2012)
38. L.L. Minku, X. Yao, Software effort estimation as a multi-objective learning problem. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **22** (2013)
39. R.G.F. Soares, Effort estimation via text classification and autoencoders, in *International Joint Conference on Neural Networks* (2018), pp. 1–8
40. M. Choetkiertikul, H.K. Dam, T. Tran, T. Pham, A. Ghose, T. Menzies, A deep learning model for estimating story points. *IEEE Trans. Softw. Eng.* **45**(7), 637–656 (2019)
41. B.W. Boehm, Software engineering economics. *IEEE Trans. Softw. Eng. (TSE)* **10**(1), 4–21 (1984)
42. Nasa93 (2008). <https://doi.org/10.5281/zenodo.268419>
43. B.W. Barry, *Software Engineering Economics* (Prentice-Hall, Englewood Cliffs, NJ, 1981)
44. B. Kitchenham, S.L. Pfleeger, B. McColl, S. Eagan, An empirical study of maintenance and development estimation accuracy. *J. Syst. Softw. (JSS)* **64**(1), 57–77 (2002)
45. L.L. Minku, X. Yao, How to make best use of cross-company data in software effort estimation?, in *International Conference on Software Engineering (ICSE)* (New York, NY, USA, 2014), pp. 446–456
46. L.L. Minku, F. Sarro, E. Mendes, F. Ferrucci, How to make best use of cross-company data for web effort estimation?, in *International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2015), pp. 1–10
47. E. Kocaguneli, B. Cukic, T. Menzies, H. Lu, Building a second opinion: learning cross-company data, in *International Conference on Predictor Models in Software Engineering (PROMISE)* (Baltimore, USA, 2013)
48. L. Song, L. Minku, X. Yao, The potential benefit of relevance vector machine to software effort estimation, in *International Conference on Predictor Models in Software Engineering (PROMISE)* (Turin, Italy, 2014), pp. 52–61
49. L. Song, L. Minku, X. Yao, A novel automated approach for software effort estimation based on data augmentation, in *ACM Symposium on the Foundations of Software Engineering (FSE)* (Lake Buena Vista, Florida, USA, 2018)
50. B. Kitchenham, L.M. Pickard, S. Linkman, P.W. Jones, Modeling software bidding risks. *IEEE Trans. Softw. Eng. (TSE)* **29**(6), 542–554 (2003)
51. M. Jorgensen, Realism in assessment of effort estimation uncertainty: it matters how you ask. *IEEE Trans. Softw. Eng. (TSE)* **30**(4), 209–217 (2004)
52. M. Klas, A. Trendowicz, A. Wickenkamp, J. Munch, N. Kikuchi, Y. Ishigai, The use of simulation techniques for hybrid software cost estimation and risk analysis, in *Advances in Computers*, vol. 74 *Software Development* (Academic Press, 2008), pp. 115–174
53. M. Tipping, Sparse Bayesian learning and the relevance vector machine. *J. Mach. Learn.* **1**, 211–244 (2001)

54. B. Włodzimierz, *The Normal Distribution: Characterizations with Applications* (Springer, 1995)
55. T. Foss, E. Stensrud, B. Kitchenham, I. Myrtveit, A simulation study of the model evaluation criterion MMRE. IEEE Trans. Softw. Eng. (TSE) **29**, 985–995 (2003)
56. B. Kitchenham, L. Pickard, S. MacDonell, M. Shepperd, What accuracy statistics really measure. IEE Proc. Softw. Eng. **148**(3), 81–85 (2001)
57. I. Myrtveit, E. Stensrud, M. Shepperd, Reliability and validity in comparative studies of software prediction models. IEEE Trans. Softw. Eng. (TSE) **31**(5), 380–391 (2005)
58. M. Shepperd, S. McDonell, Evaluating prediction systems in software project estimation. Inf. Softw. Technol. (IST) **54**, 820–827 (2012)
59. M. Klas, A. Trendowicz, Y. Ishigai, H. Nakao, Handling estimation uncertainty with bootstrapping: Empirical evaluation in the context of hybrid prediction methods, in *International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2011), pp. 245–254
60. M. Jørgensen, Evidence-based guidelines for assessment of software development cost uncertainty. IEEE Trans. Softw. Eng. (TSE) **32**(11), 942–954 (2005)
61. M. Jørgensen, K.H. Teigen, K. Molokken, Better sure than safe? Overconfidence in judgement based software development effort prediction intervals. J. Syst. Softw. (JSS) **70**, 79–93 (2004)
62. L. Song. sunnysong14/SoftwareEffortEstimation (2022). <https://github.com/sunnysong14/SoftwareEffortEstimation>
63. A. Ali, C. Gravino, A systematic literature review of software effort prediction using machine learning methods. J. Softw. Evol. Process **31**(10) (2019)

Chapter 3

Requirements Engineering



Fitsum Kifetew, Anna Perini, and Angelo Susi

Abstract Requirements Engineering aims at supporting the understanding of the purpose of a software system to be built, and at keeping the whole design and development process aligned with it. Research in Requirements Engineering (RE) provides methods and techniques to support various activities in the requirements life cycle, from requirements elicitation to requirements verification and validation. Artificial Intelligence (AI) techniques are more and more exploited in such methods, including natural language processing techniques, since many RE artefacts are expressed as natural language text; techniques based on optimisation, machine learning, and deep learning with the objective of improving the efficiency of the analysts and decision-makers performing RE activities. In this chapter, we focus on two specific use cases in RE, namely, requirements elicitation from textual user feedback and requirements prioritisation. We present solutions to the two problems based on AI techniques, specifically machine learning, natural language processing, and Genetic Algorithms. The application of the proposed methods in industrial contexts allowed us to validate their usefulness in terms of increased efficiency of organisations during their decision-making processes. Finally, we discuss the use cases in the broader context of the RE management process, highlighting opportunities and limits of the AI approaches and current trends in the use of AI in RE.

F. Kifetew · A. Perini · A. Susi (✉)
Fondazione Bruno Kessler, Trento, Povo, Italy
e-mail: susi@fbk.eu

F. Kifetew
e-mail: kifetew@fbk.eu

A. Perini
e-mail: perini@fbk.eu

3.1 Introduction

Requirements Engineering (RE) aims at understanding the purpose of a software system to be built, and at keeping the whole development project aligned with it. RE's core question can be stated as *What's the right system to be built?*, while the overall purpose of software engineering concerns how to build such a system in the right way. RE tasks are usually performed at the beginning of each iteration in a software system development project, but are also directly linked to specific software engineering tasks, such as testing. Many decisions in software development projects are affected by how well RE is performed. Most failures of a software project are due to errors introduced during requirements engineering activities.

Research in RE provides techniques to support various activities in the requirements life cycle, from requirements elicitation to requirements validation and verification. In this chapter, we focus on those techniques that exploit Artificial Intelligence (AI).

AI for RE has become a specific research area called *Artificial Intelligence and Requirements Engineering (AIRE)* [4] in which a key objective is to investigate the applicability of AI techniques to support or automate RE tasks. Among the applications of AI to RE are the following.

Knowledge representation and reasoning techniques have been applied to extend visual modelling languages that are used to build requirements models, thus allowing automated reasoning on such models during requirements analysis. Examples of automated reasoning in goal-oriented modelling are simulation through model checking, to explore the properties of the model's entities over their lifetime [10], and satisfiability analysis using a SAT solver [11].

Natural Language Processing techniques (NLP) are used to support the analysis for RE purposes of textual documents containing knowledge about the application domain. Zhao et al. [29] presented an analysis of 370 research publications on NLP for RE. One of the objectives of the analysis was to understand which RE tasks were considered in these research works. Results indicate that requirements analysis is one of the tasks that can receive more support from NLP (42% of the considered publications propose NLP approaches in support of the analysis task), followed by requirements management, elicitation and modelling (16% each), and validation and verification tasks (4%). NLP techniques are also used in combination with machine learning techniques to analyse huge amounts of text messages provided by users as feedback upon having used a software system (also called explicit user feedback), with the purpose of extracting relevant information for eliciting, validating, and prioritising requirements [13, 17].

In this chapter, we focus on two specific use cases in RE, namely, requirements elicitation from textual user feedback, and requirements prioritisation. We provide their descriptions as isolated activities then we introduce the connections between the two cases in the general context of agile software development processes.

The rest of the chapter is organised as follows. In Sect. 3.2, we give an overview of RE and key concepts. In Sect. 3.3, we present the requirements elicitation use case.

In Sect. 3.4, we describe the requirements prioritisation use case. In Sect. 3.5, we discuss how the two use cases relate to the overall RE process. Section 3.6 offers a discussion on the limits of the described techniques, and on recent trends in AI for RE, and finally Sect. 3.7 concludes the chapter.

3.2 Requirements Engineering

Requirements Engineering, as a specific discipline arose in the seventies, when scientific publications started introducing the term *requirements* and proposing systematic ways of managing them in the context of the software development process, e.g. [2]. This terminology was later framed in the context of dedicated standards, such as the IEEE 830-1984¹ which is considered a milestone in RE practice.

Requirements of a software system can be defined as the specification of *how the system should behave, or of a system property or attribute* [24]. The basis for building good quality requirements specification is to understand the application domain, also called the *world* by Zave and Jackson in their formulation of the *requirement problem* [27], and to differentiate domain knowledge from the artefacts, such as software design, and technological platforms that we use to build the system-to-be, which is called the *machine* domain. Requirements specifications are artefacts belonging to the intersection of these two domains, as depicted in Fig. 3.1. By exploring the problem world, we can identify the stakeholders—users, customers, suppliers, developers, and business organisations—with their goals and needs, and the alternative ways the system to be built can satisfy these goals and meet the needs. We can also well identify other phenomena in the problem space, such as constraints and regulations that the new system to be built must comply with. That is, the requirements engineer’s perspective focuses on the purpose of the system to be designed and built.

Requirements can be differentiated depending on the stakeholder’s perspective. For instance, *business requirements* correspond to the goals of the organisation that builds a software product, while *user requirements* correspond to goals and tasks that users want to be able to perform with the system to be built, or to system qualities the users expect. Software requirements are also differentiated depending on what they describe, namely, *functional requirements* describe the behaviour that the software system must exhibit under specific conditions. On the other hand, *non-functional requirements* describe qualities that characterise such behaviours as performance and scalability, or properties that software systems must exhibit, or still constraints that it has to meet, e.g. portability. Different taxonomies of requirement types can be found in the literature, e.g. [12].

Requirements engineering offers a set of methods and techniques for managing requirements in a systematic way. Such methods and techniques are used by

¹ IEEE 830-1984—IEEE Guide for Software Requirements Specifications, <https://standards.ieee.org/>.

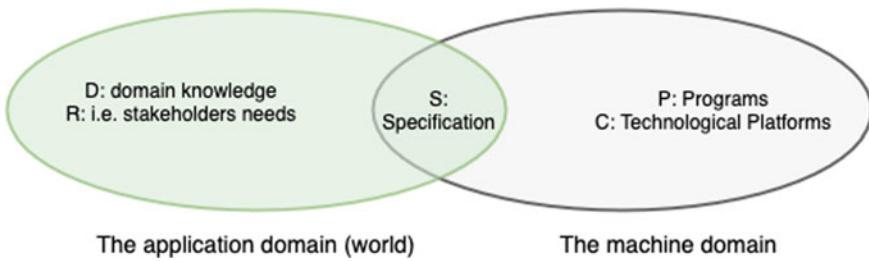


Fig. 3.1 The Requirements Problem according to Zave and Jackson [27]

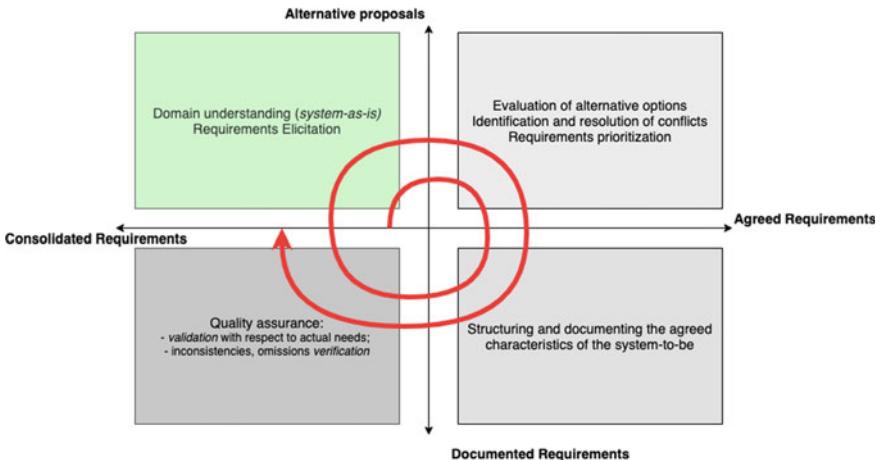


Fig. 3.2 Requirements Engineering life cycle. Adapted from [25]

requirements engineers to perform RE tasks in each specific phase of the requirements process, which in turn involves a set of iterations over four main phases [25], as depicted in Fig. 3.2.

Domain stakeholders with their goals and their relationships. Requirements elicitation corresponds to the discovery of the needs of the stakeholders that can motivate the change from the system-as-is to the *system-to-be*, and to the analysis of alternative ways in which the system-to-be can meet them. Different methods can be used when performing these tasks, including introspective analysis of domain-specific documents, interviews of domain experts and other stakeholders, as well as collaborative methods such as focus groups, or approaches based on ethnographic techniques. Modelling methods, such as goal-oriented modelling can help consolidate and validate the findings in terms of a set of alternative proposals, which will be input to the following phase that concerns evaluation and prioritisation.

Requirements evaluation encompasses several tasks including the identification and resolution of conflicting concerns, the identification and assessment of risks, as well as the evaluation of the alternative options resulting from the previous analysis

tasks. Requirements prioritisation is performed in this phase and may require a trade-off between business opportunities and project constraints, such as time and budget constraints.

The resulting requirements for the system-to-be, which have been agreed upon with the stakeholders, need to be described in specific project artefacts. This is the objective of the third phase in the requirements life cycle. The resulting artefacts should describe in a coherent structure the objectives, concept definitions, domain properties, system and software requirements, and environmental assumptions.

The fourth phase concerns quality assurance, which involves two key tasks: *validation* of the specified requirements with stakeholders, whose objective is to check the adequacy of the stated requirements with respect to the actual needs, and *verification* whose purpose is to detect inconsistencies among requirements, omissions, and fixing errors before starting development. The output of this phase are consolidated requirements.

The requirements engineering process consists of several iterations along these four phases, depending on the complexity of the application domain, and of the system to be built.

Each of the described RE tasks is characterised by decision-making activities, which can be supported or automated by AI-based methods and techniques.

3.3 Use Case About Requirements Elicitation

Requirements elicitation is an important phase of RE where the analyst tries to capture the requirements for the system to be built from the involved stakeholders. The stakeholders could be the clients, end users, domain experts, investors, etc. As a result, understanding the needs of the stakeholders and capturing them in a set of clearly defined requirements is typically a challenging task. The task is made more difficult by the fact that the requirements analyst usually does not have sufficient knowledge about the domain of the system being built. Hence, the analyst cannot be sure to have elicited all requirements. All these factors make requirements elicitation a deceptively easy and straightforward but actually difficult task.

With the proliferation of the Internet and mobile applications as well as online platforms that support end user discussions, there is increasing feedback from end users regarding a specific system (software) they are using. Hence, in the context of software evolution, the requirements analyst has an additional source of information that comes directly from the end users of the system. Depending on the type of software, the platforms could be different. For instance, large and widely used software have dedicated online platforms where end users discuss potential problems they faced while using the software. There are also systems that have issue tracking systems that are directly available to end users so that they can report issues directly via the tracking system. Additionally, most mobile applications provide a mechanism for end users to submit feedback about the application in the form of user reviews. Such

reviews, typically containing short textual messages and ratings/stars, are facilitated by the application marketplaces that host the applications, e.g., the Google Play Store or the Apple App Store.

A natural consequence of this abundant availability of end user feedback is to see how it could be useful for the purpose of eliciting new requirements for a given system. This topic has been the focus of growing research efforts and a number of interesting approaches have been proposed in order to make use of end user feedback as a source of new requirements. This section presents a use case in which information relevant to requirements elicitation is identified from end user feedback by applying NLP and machine learning (ML) techniques.

3.3.1 The Problem of Requirements Elicitation from User Feedback

As mentioned earlier, end user feedback is potentially a useful source of information for deriving new requirements that are important for the user. However, by its nature, user feedback is unstructured and is typically expressed in a textual format using natural language. Furthermore, the natural language typically used by end users to express their feedback is rather informal and could also include particular expressions, such as slang, short notations, etc. Hence, extracting the relevant information that could be useful for the purpose of deriving requirements is quite challenging, even more so when performed in an automated manner.

An important characteristic of (online) user feedback is that it is bulky. For many systems, user feedback could arrive at the rate of hundreds or even thousands per day. Hence, analysing the feedback messages manually becomes a daunting task. Therefore, automating the process of extracting requirements relevant information from the user feedback characterises the problem of requirements elicitation from user feedback.

More specifically, the ultimate objective of automated requirements elicitation, given a set of user feedback messages, is to extract information from the feedback messages which is relevant for eliciting one or more requirements for the system under consideration.

Achieving this objective requires applying methods and techniques from different areas of NLP as well as AI, among others, in a coordinated manner so as to filter out the less relevant information and identify the likely ones for further inspection and decision by the requirements analyst. The next section discusses one such effort which is based on identifying the intention of the user when giving feedback.

3.3.2 A Solution to Requirements Elicitation Based on NLP Techniques

As previously mentioned, user feedback is expressed in natural language and predominantly in a textual manner. Hence, techniques for dealing with natural language content are suitable for the task. However, NLP techniques are generic and work at a lower level of granularity with respect to the conceptual elements one would like to extract for the purpose of requirements elicitation. For this purpose, additional concepts concerning natural language are typically introduced for processing user feedback and eventually extracting relevant information at a higher level of abstraction. One such approach is speech acts analysis [19]. Speech acts capture what a person is trying to communicate when he/she addresses another person. For example, if a user writes “I would like to be able to save a file to PDF.”, the user is communicating the desire to have a specific thing (*requestive* speech-act). There is a taxonomy of speech acts derived from the linguistics domain and adapted for the RE domain, as summarised in Table 3.1 [18].

Speech acts, identified from the user feedback messages, form the foundation on which the process of extracting requirements relevant information is built. Given a set of user feedback messages, the overall process is as follows:

1. Prepare the (raw) feedback messages by cleaning the text, and applying NLP-based low-level filtering operations, e.g., removing special characters.
2. Identify the speech-act(s) present in each feedback message.
3. Based on the identified speech acts, determine whether or not that particular feedback message is relevant for requirements elicitation or not.

Step 1 is implemented by applying pre-processing techniques to the text via pattern matching. Step 2 is implemented by means of a set of rules that can identify one or more speech acts in the given text (see Table 3.1). Each speech-act category has its own rules that determine whether or not it is present in a given text. Step 3 is implemented by training a ML classifier, which is based on the identified speech acts, that is able to categorise the feedback message into a predefined set of outcomes.

Table 3.1 Speech acts relevant for requirements elicitation

c-Assertive	c-Requestive	c-Responsive	c-Attachment	c-Other
Assertive	Requestive	Responsive	Attach	Descriptive
Confirmative	Requirement	Suggestive	Code line	Accept
Concessive	Questions	Suppositional	URL link Log file	Reject Negative opinion Positive opinion Thank Informative

3.3.3 Identifying Speech Acts

Identifying the various speech act types contained in given natural language text requires the definition of lexico-syntactic rules that correspond to each speech act type. The rules are mainly based on the presence of specific linguistic elements (e.g., verbs) that characterise a given speech act. For example, *requestive* speech acts are characterised by verbs such as “insist”, “solicit”, “urge”, etc. Wordnet² is used to expand terms (e.g., verbs) using synonyms. Overall, we have developed a total of 142 rules for identifying speech acts in user feedback, available in the replication package.

Besides speech acts, we also used *sentiment analysis* to determine the sentiment of the user when giving a particular feedback. We use the outcome of the sentiment analysis for defining, together with speech acts, the set of properties that characterise feedback messages interesting for requirements elicitation, discussed in the next section.

3.3.4 Training a Classifier

In order to achieve the ultimate objective of extracting information relevant to RE from user feedback, an important step is building a classifier capable of distinguishing feedback messages with relevant information for eliciting requirements. To train a classifier, it is important to define the set of properties (aka *features* in the ML world) that characterise the domain with respect to the classification task at hand. Hence, the first step is to build a set of candidate properties that are important for distinguishing a given user feedback as useful for RE or not. For our purpose of extracting information relevant to RE from user feedback, we need to first outline the target classes (categories) into which we would like to classify the user feedback messages. For the current use case, we consider two categories: enhancement and other. The *enhancement* category is intended for those feedback messages that contain a request for a new functionality or eventual enhancement of existing ones. This category is expected to contain the feedback messages which would serve as input to the analyst for identifying new requirements. The *other* category is expected to contain feedback messages that do not have characteristics interesting for requirements elicitation, e.g., bug reports, praise, or compliment. They may not be immediately useful for the analyst but still they could be consulted for further insight with a lower priority.

The next step is to identify the properties that could help in distinguishing the feedback messages that should fall into one of the aforementioned categories. The process is potentially iterative, starting out with a large (conservative) list of properties, then refining the list by means of preliminary experimentation as well as automated feature selection mechanisms available in most ML tools.

² <https://wordnet.princeton.edu/>.

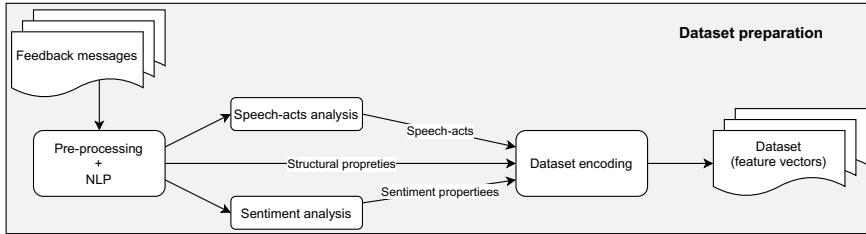


Fig. 3.3 Dataset preparation

We identify 42 properties, in three categories, that could help us in building an effective classifier. The categories of properties are:

1. speech acts related properties: 18 properties identified from the speech acts contained in the feedback messages, calculated as the frequency of occurrence of each speech acts category in the feedback, e.g., number of requestive speech acts
2. sentiment related properties: 9 properties derived from the feedback messages by means of sentiment analysis, e.g., number of positive/negative nouns/verbs
3. structural properties: 15 properties that represent various linguistic aspects of the feedback messages, e.g., number of sentences in the feedback message.

Consequently, a given feedback message is transformed into a vector of 42 values, each containing the corresponding value of the property described above. Hence, given a set of feedback messages, they can be transformed into a set of vectors each corresponding to a particular feedback message. This process of extracting properties from the feedback and encoding them into feature vectors (dataset preparation) is depicted in Fig. 3.3.

Once the properties are encoded as vectors (aka feature vectors), then standard ML tools could be used to train models. For this purpose, we need to prepare a set of feedback messages (hence corresponding feature vectors) for which we know the classification outcome. Such a set is known as the *training set*, and is used to train and evaluate an ML model. In general, the bigger the training set, the more accurate the trained model. However, preparing a training set, i.e., a set of feedback messages together with their classification, is typically a task that requires manual effort and is costly.

The overall process of training ML models, given a set of labelled feedback messages, is depicted in Fig. 3.4. Different ML algorithms are readily available for use. We use the Weka³ toolset for training the classifiers, and in particular, we experimented with *RandomForest*, *J48*, and *SMT* algorithms. During the training phase, typically the same training dataset is also used for testing the models, and eventually improving the training process. There are two commonly used approaches for doing that: (1) splitting the training set, based on some percentage e.g., 70 and 30%, into a training set and a test set, the model will be trained on 70% of the training

³ <https://www.cs.waikato.ac.nz/ml/weka/>.

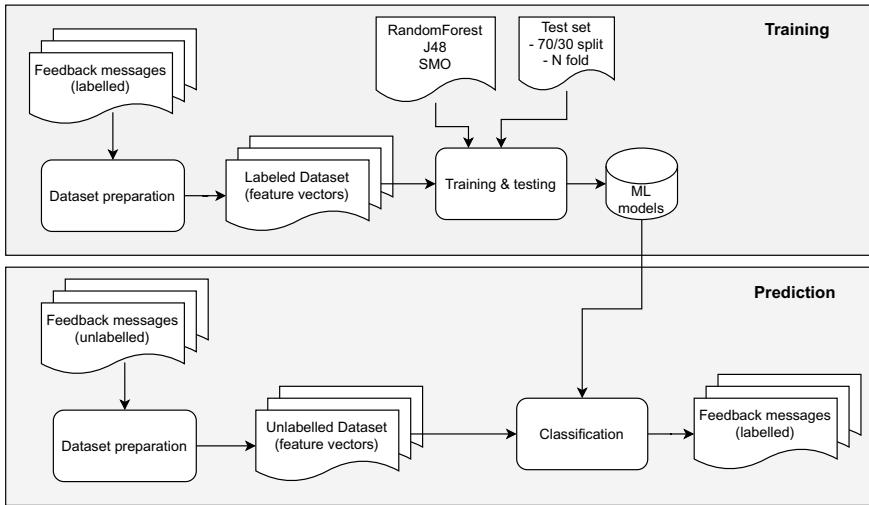


Fig. 3.4 Training ML models (upper) and predicting (lower)

set and the remaining 30% will be used to test the accuracy of the model; (2) apply n -fold cross-validation by splitting the training set into n equal subsets and using the $n - 1$ subsets for training and the n th subset for testing, then repeating for all the n subsets.

An important issue worth mentioning here is that in the training set it is common to have unbalanced groups. For instance, in our case, the number of feedback messages labelled as *enhancement* are typically smaller than those labelled as *other*. Such disparity in the dataset creates a bias in the ML algorithm, and hence the trained model will be biased by the fact that some category has more samples than others. There are a number of solutions for dealing with this problem. For instance, simply discarding the excess data elements in the larger category so that the categories become equal in size. Given the difficulty of preparing labelled data, this may not always be a good alternative. Another approach is to apply *oversampling* of the minority category, based on the existing data elements in that category. One such approach is SMOTE (Synthetic Minority Over-sampling Technique) [3]. It is implemented in Weka and we have used it for our dataset.

3.3.5 Applying on Two Case Studies

Given a set of feedback messages for a given system under consideration, the identification of those messages relevant for the RE analyst is depicted in Fig. 3.4 (lower part).

In a typical usage scenario, as depicted in the lower section of Fig. 3.4, one or more feedback messages are fed into the prediction pipeline where they are pre-processed, then the corresponding dataset (feature vectors) are prepared, the dataset is then passed through the previously trained ML model which predicts its category (label) into one of the predefined categories (in our case *enhancement* or *other*). Eventually, those feedback messages classified as *enhancement* are those that the RE analyst would take into consideration for the task of eliciting new requirements.

In the subsequent sub-sections, we present the results of applying the pipeline described above in two case studies. The first case study is based on comments taken from the *Apache OpenOffice* community issue tracking system,⁴ while the second case study is based on feedback messages taken from the issue tracking system of *SEnerCON*, an SME working in the home energy management domain. In both cases, we train classifier models using three algorithms (*RandomForest*, *J48*, *SMT*) in Weka and assess their accuracy by means of 10-fold cross-validation. Accuracy is measured by means of *precision*, *recall*, and *f-measure*.

3.3.5.1 OpenOffice Dataset

The labels (categories) of the comments were determined based on their status in the issue tracking system. We take the category assigned to the comments by the experts as their category. We considered only those comments in issues with *status* acknowledged as *confirmed* or *accepted*. After preparing the dataset (see Fig. 3.3), since the dataset was unbalanced, we applied oversampling with SMOTE. The size of the resulting dataset is 29,074 (14,504 labelled as *Enhancement* and 14,570 labelled as *Other*). We trained ML models (see Fig. 3.4) using the dataset and applied 10-fold cross-validation. The results are reported in Table 3.2.

3.3.5.2 SEnerCON Dataset

The dataset in this case was labelled by an expert in the domain working in the company. Similarly, as in the case of *OpenOffice* after applying SMOTE, the dataset contained a total of 812 instances, 395 labelled as *Enhancement* and 417 labelled as *Other*. The results of applying 10-fold cross-validation are shown in Table 3.3.

3.3.6 Discussion

The results in Tables 3.2 and 3.3 show that the models trained with *RandomForest* outperform the other two, with similar performance, in terms of *f-measure*, for both datasets. In particular, we see that the precision is higher for the *Enhancement* cate-

⁴ <https://bz.apache.org/ooo/>.

Table 3.2 Results for *OpenOffice*

	RandomForest			J48			SMO		
	P	R	F-M	P	R	F-M	P	R	F-M
Enhancement	0.87	0.76	0.81	0.79	0.74	0.77	0.77	0.53	0.63
Other	0.79	0.89	0.84	0.76	0.81	0.78	0.64	0.84	0.73

Table 3.3 Results for *SEnerCON*

	RandomForest			J48			SMO		
	P	R	F-M	P	R	F-M	P	R	F-M
Enhancement	0.86	0.77	0.81	0.75	0.70	0.72	0.73	0.52	0.61
Other	0.80	0.88	0.84	0.73	0.77	0.75	0.64	0.82	0.72

gory while the recall is higher for the *Other* category. Given the fact that the training sets are limited in size, the results obtained are reasonably good, which in turn shows the effectiveness of the properties used for characterising the feedback messages (discussed in Sect. 3.3.4). For both datasets, we have experimented with the various ML algorithms available in the Weka toolset and chose the ones that showed better performance. Clearly, given the datasets, any algorithm in any analysis toolset could be used.

The results reported in Tables 3.2 and 3.3 show the values of *precision*, *recall*, and *f-measure*, which are typically reported in scientific publications. While in general *f-measure* gives a good estimate of the overall performance of an algorithm in terms of both *precision* and *recall*, users could take a closer look at the results and make decisions more suited to the problem at hand. For instance, if the application domain has a large quantity of user feedback and scrutinising false positives could be demanding, an algorithm with a higher *precision* could be preferred. On the other hand, if we give more weight to finding as many potential requirements as possible from the given set of user feedback, then an algorithm with higher *recall* would be preferable.

3.4 Use Case About Requirements Prioritisation

Requirements prioritisation plays an important role during the various iterations of traditional and agile software development processes. Several aspects affect the decision of which requirements to implement in the next release of a software system and in what order they have to be implemented. These aspects include the availability of resources, time constraints, technical constraints, and the end user's expectations and feedback. In this section, we introduce the problem of requirements prioritisation,

we propose a solution based on multi-objective algorithms and illustrate its evaluation in an industrial setting.

3.4.1 The Problem of Requirements Prioritisation Using User Feedback

The requirements prioritisation problem concerns the task of finding an order relation on a given set of requirements. The related prioritisation process can be based on information available *a priori* (i.e., independently of the specific requirement instances) or *a posteriori* (i.e., depending on the specific features of the requirements at hand) [23]. In the former case, the preferences are formulated before the specification of the set of requirements via predefined mathematical models, for example, by defining the requirement attributes which determine the final ranking and by specifying how the final ranking is computed from those attributes. In the *a posteriori* approaches, the ranking is formulated on the basis of the characteristics of the set of requirements under analysis, hence emerging from the preferences specified by the decision-makers combined with other ranking criteria. An important set of criteria comes from the analysis of user feedback. Those criteria concern the sentiment, the intention or the severity embedded in the feedback related to the existing requirements of the system that have to be prioritised. This knowledge from the actual usage of the system is exploited by human decision-makers while specifying their rankings. Moreover, user feedback is also the source of new requirements for the evolution and enhancement of the system as also envisaged in Sect. 3.3. Here we focus on this second aspect when the organisation adopts agile software development processes.

Focusing on the decision-making processes in organisations that adopt agile processes, the work in agile teams involves several activities and roles interested in expressing criteria and preferences for the prioritisation of requirements. The activities are mainly related to the identification of the relevant requirements, their prioritisation, and negotiation in case of conflicts (Fig. 3.5). The roles involved include developers, analysts, product engineers, and usually a team leader who is in charge of coordinating the planning and development of the piece of software. These team members have heterogeneous competencies and expertise and can access to different kind of information sources related to technical or business characteristics of the product under development. Thus, when involved in shared prioritisation of a set of candidate requirements (i.e., the current backlog) to be implemented as part of an upcoming release of the software, they decide upon the given prioritisation criteria, on the basis of different expertise levels. Moreover, since they are in a continuous development or fast development approach, such decisions need to be taken rapidly and on a frequent basis also considering the feedback from the actual use of the system by the users and the requirements they may express in their feedback.

Here we focus on a scenario related to the prioritisation process, that we call *shared requirements prioritisation process* in which a team of project managers, designers, and developers in an organisation aim at searching for the maximum consensus among their preferences with respect to the requirements or for detecting strong differences among their preferences. The team members are the decision-makers responsible for contributing to the final prioritisation decision and providing their own prioritisation of the requirements. At least one of the decision-makers in the team plays the role of *supervisor* of the entire process who takes care of making the ultimate decision regarding the priorities, considering the individual decisions of the involved decision-makers (Fig. 3.6). The supervisor is also responsible for configuring the overall requirements prioritisation process. This configuration task involves defining: a set of requirements to be prioritised, including the relevant new

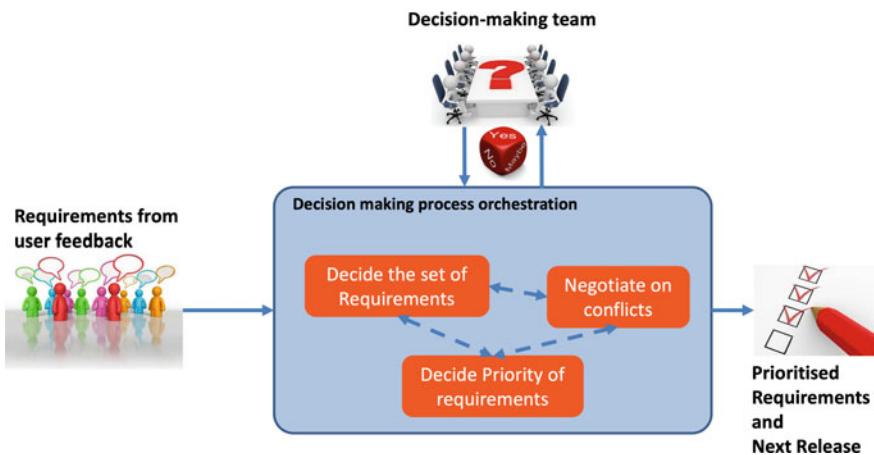


Fig. 3.5 The Requirements prioritisation process considering collaborative aspects and user feedback

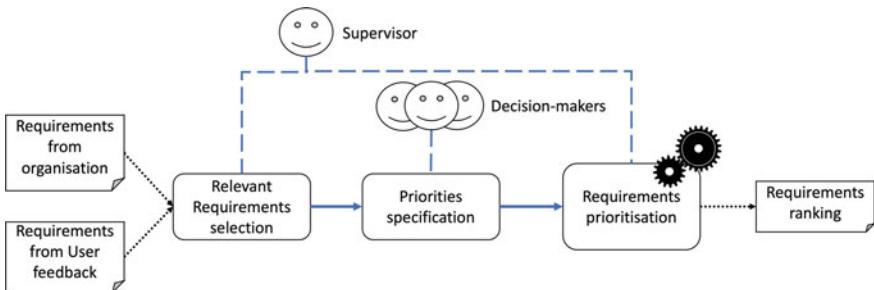


Fig. 3.6 The Requirements prioritisation process. The supervisor is in charge of selecting and preparing the relevant requirements from organisation and from user feedback and to decide the final prioritisation, while decision-makers provide their own priorities. The computation of the final requirements ranking is performed by a search-based evolutionary algorithm

requirements extracted from the user feedback to be included in the prioritisation process considering, for example, the sentiment expressed by the users with respect to the new functionalities of the system; a set of decision-makers to be involved in the prioritisation; a set of prioritisation criteria, such as cost, value or effort to implement the requirements, to be specified by the decision-makers; the possible dependencies among the requirements. More formally, we consider:

- A set of m requirements $R = \{r_1, r_2, \dots, r_m\}$ from the analysts in the organisation or extracted from the user feedback to be prioritised by the decision-makers;
- A set of n decision-makers $DM = \{dm_1, dm_2, \dots, dm_n\}$;
- A set of p criteria $\{c_1, \dots, c_p\}$;
- The ranking of the set of requirements specified by a decision-maker dm_i with respect to the criterion c_j that is $Rank[dm_i, c_j]$;
- A set of dependencies between pairs of requirements $\{(r_i \rightarrow r_j), \dots\}$ where $(r_i \rightarrow r_j)$ implies that requirement r_j is a prerequisite for requirement r_i .

Once the *shared requirements prioritisation process* has been set up, the decision-makers in the team provide their preferences regarding the ordering of the requirements with respect to the criteria defined by the supervisor. Each decision-maker expresses, independently, his/her own priority list, in the form of an absolute ordering of the requirements.

The resulting shared requirements prioritisation problem consists of finding the best ordering of the given requirements, namely $Prio_{final}$, taking into account all the individual orderings of the decision-makers. Reaching a consensus regarding the final prioritisation of the requirements involves: (i) identifying the alternative global rankings that can be extracted from the decision-makers' preferences; and (ii) selecting the final prioritisation, through a decision process that takes into account additional contextual and strategic information.

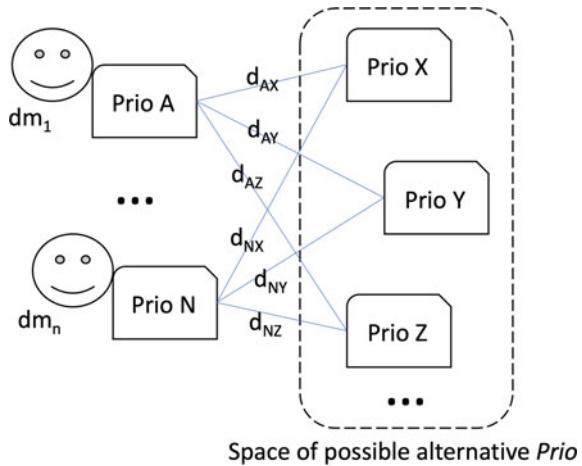
3.4.2 A Solution to Requirements Prioritisation Based on Genetic Algorithms

A characterising aspect of the proposed formulation is that of supporting the decision-making related to the prioritisation of the requirements and in parallel the support of a consensus/agreement process to search for a common middle ground among all decision-makers.

3.4.2.1 Searching for the Middle Ground

Given m requirements and n decision-makers, the proposed approach first quantifies the distance, as an approximation of dissimilarity, between a pair of orderings from decision-makers. Then, it searches for the ordering which has the minimum distance

Fig. 3.7 The shared requirements prioritisation process considering the ranking from the decision-makers (*Prio*) the objective is that of choosing one (or more) ranking(s) from the space of possible alternative rankings that minimises the distances d_{ij} with those from the decision-makers



from all decision-maker rankings as shown in Fig. 3.7, so optimising the consensus among the decision-makers.

In Fig. 3.7, the priorities from the decision-makers *Prio A*, ..., *Prio N* refer to the simplified case of a single prioritisation criterion. However, in our formulation, each decision-maker may decide on the bases of several different (p) prioritisation criteria. Therefore, the calculation of the distances d_{ij} shown in Fig. 3.7 becomes a potentially very complex activity.

Given a set of m requirements to be prioritised, we represent a prioritisation (ordering) of these requirements as $Prio = (rank_1, rank_2, \dots, rank_m)$ where $rank_i$ is the *rank* of *requirement i*. For example, for $m = 6$, a prioritisation could be: (5,3,4,1,2,6); that is, the rank of *requirement 1* is 5, the rank of *requirement 2* is 3, the rank of *requirement 3* is 4, and so on.

Given two rankings $Prio_1$ and $Prio_2$ of m requirements, the *distance* between them could be computed in a number of ways. Among the others, we propose a distance based on the *Kendall's τ* statistic [16]. Computing *Kendall's τ* between two rankings gives a value in the range $[-1, 1]$ corresponding to the level of *agreement* (*similarity*) between them. A value of 1 represents full agreement while a value of -1 represents full disagreement.

Once the mechanism of finding the middle ground illustrated in Fig. 3.7 is defined, and the dissimilarity of rankings is quantified, we rely on multi-objective optimisation algorithms [6] to find optimal solutions to the shared requirements prioritisation in which each prioritisation criterion is considered as an objective to be minimised.

3.4.2.2 Multi-objective Optimisation for Requirements Prioritisation

The main goal of the multi-objective optimisation is that of exploring alternative optimal solutions in the various objectives that are considered in the process. In our

case, we explore solutions that are prioritisations of requirements that represent low levels of dissimilarity among the decision-makers involved with respect to the prioritisation criteria considered. For instance, if the prioritisation criteria are development effort and user impact, then multi-objective optimisation gives us solutions that represent minimal dissimilarity with respect to each of these criteria. Such solutions constitute the Pareto front [5] resulting from the multi-objective optimisation. The ultimate decision could then be made by exploring these Pareto optimal solutions and picking one of the equivalent optimal solutions in the front.

To this aim, we employ Evolutionary Algorithms (EAs), a class of metaheuristics algorithms in which a *population* of candidate solutions (*individuals*) interact with each other and evolve through *generations* following the principle of survival of the fittest [7]. Further details about the general principles and characteristics of the metaheuristics and specifically of the Genetic Algorithms are given in the CHAPTER “EVOLUTIONARY ALGORITHMS” of this book. In our approach, we used the NSGA-II multi-objective genetic algorithm [6].

An EA starts by creating an initial population of individuals in our case the ranking of requirements; it then evaluates each individual by means of a *fitness function* and assigns it *fitness values*. It then proceeds by selecting “fitter” individuals (*parents*) from the population and subjecting them to *recombination (crossover)* resulting in *offspring*. The offspring could further be *mutated* to introduce diversity into the population. The EA then selects the survivors that constitute the next generation. This process iterates until some predefined *stopping condition* is reached, e.g., time budget expires. The result of the whole process is a set of equivalent solutions in the Pareto front from which to pick the final solution, $Prio_{final}$. In the following, we report the different aspects of the multi-objective algorithm in our specific use case.

Individual encoding. A candidate individual in the search is represented as an ordering of requirements: given m requirements to be prioritised, a candidate I is encoded as $(rank_1, rank_2, \dots, rank_m)$, where $rank_i$ is the *rank* of requirement i in the ranking I .

Fitness function. The fitness of an individual I is defined by computing the *distance* of I from the rankings of the n decision-makers, with respect to each *criterion* and from the rankings induced by the feedback characteristics, *feedback*; resulting in the set of objective functions f shown in Eq. 3.1, corresponding to each criterion c_1, \dots, c_p . Each of these objectives is to be minimised as specified in Eq. 3.2. So, we search for a rank I that has a minimal distance from all the rankings specified by the n decision-makers.

$$\left\{ \begin{array}{l} f(I, c_1) = \frac{1}{n} * \sum_{i=1}^n distance(I, Rank[dm_i, c_1]) \\ \vdots \\ f(I, c_p) = \frac{1}{n} * \sum_{i=1}^n distance(I, Rank[dm_i, c_p]) \end{array} \right. \quad (3.1)$$

$$\min (f(I, c_1), \dots, f(I, c_p)) \quad (3.2)$$

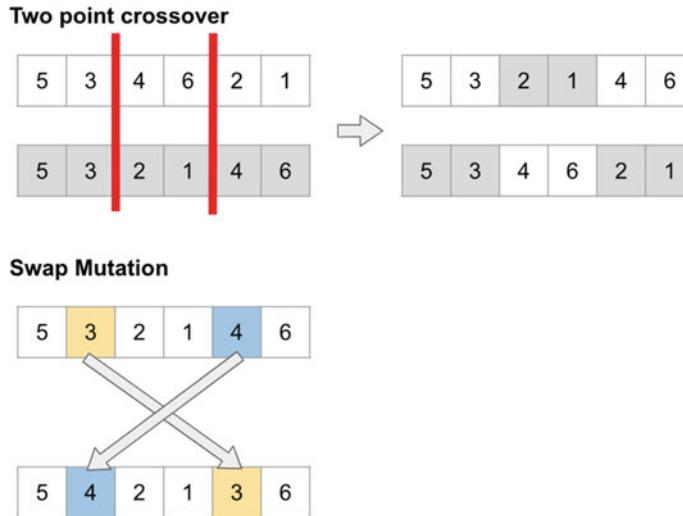


Fig. 3.8 Evolutionary operators used in the shared requirements prioritisation process

where c , n , p , dm and $Rank$ are defined in the problem formulation in Sect. 3.4.1 and the function $distance$ measures the distance between two rankings.

Evolutionary operators In this work, we used two operators that enable the EA to evolve individuals, rankings of requirements, through generations are *crossover* and *mutation*.

- *Two point crossover:* In each ranking, two points, representing two requirements, are randomly picked, and parts of the ranking are exchanged, as illustrated in Fig. 3.8. In case the crossover operator results in an invalid ranking (e.g., duplicated ranks), the operation will be performed again, choosing different crossover points.
- *Swap mutation:* Given two rankings, two points are randomly picked and the values at those selected points are *swapped*, as illustrated in Fig. 3.8.

Dependencies As discussed in Sect. 3.4.1, there could exist dependencies among requirements. In our approach, we consider the dependencies as constraints and eliminate those candidates that violate them from the population. To this end, (1) during population initialisation and after crossover/mutation, individuals that violate dependency constraints are dropped, (2) during fitness evaluation, if an individual violates dependency constraints, it is assigned the worst possible fitness value, hence it will be eventually dropped from the search.

3.4.3 Applying the Prioritisation Method

We assessed the overall approach in industrial settings involving two companies: a large enterprise (LE), and the SME *SEnerCON* introduced in Sect. 3.3.5. The LE operates in the domain of smart cities and provides a platform where application developers can build innovative applications utilising multiple data sources and provide useful services to end users. The SME works in the domain of household energy saving and management via a web application which allows users to monitor their household energy consumption. In each of these companies, there is a team of developers and managers that follow an agile methodology for maintaining and evolving their respective products. The different sizes of the companies translate into different implementations of agile methodologies, making the evaluation valid under multiple setups. In both companies, the problem was that of converging to shared plans for requirements implementation, especially in case new requirements have to be considered that have been extracted from the user feedback and that should be merged with those identified internally in the organisation.

The evaluation followed the guidelines and methodology defined in the ISO/IEC SQuaRE series of standards for System and Software quality requirements and evaluation [14]. The overall objective was to assess the impact of introducing the proposed tool-supported shared requirements prioritisation via measurements and questionnaires by customising the quality-in-use model [14] so as to measure relevant Key Performance Indicators (KPIs) such as *Efficiency*, which measures the amount of time spent on prioritisation using our tool as compared to historical observations.

The SME team was composed of nine people: six developers, a product owner, a product manager, and a development team head. The LE team was composed of four people: three developers and one product owner. In the SME the process was repeated for three sets of requirements, exhibiting different levels of difficulty and containing at least two cross-dependencies among requirements, amounting to a set of 20 requirements overall; while LE considered 15 requirements.

The efficiency is measured in terms of time spent on prioritisation. It is composed of three parts: T_{setup} —time for setting up the prioritisation process, T_{prio} —average time spent by team members to participate in the prioritisation, and T_{close} —time spent by the final decision-maker to analyse and confirm the final priorities. Table 3.4 summarises the time spent by each team in the two companies (SME and LE). We can observe from Table 3.4 that entire prioritisation process, from start to end, took about 1.7 h for the SME and about 2 h for LE. We can also note that the setup configuration time seems to be relatively high. However, time spent on setup tasks (the insertion of requirements, definition of prioritisation criteria, registration of team members' information, etc.) is expected to be amortised over time as the process is repeated, and the team familiarises itself with the process and tool. Finally, both teams have reported that by using the tool, they were able to improve the time spent on the prioritisation process (e.g., LE reported a 25% reduction), with respect to their traditional practices which typically involve physical meetings.

3.4.4 Discussion

The need for supporting the complex prioritisation processes expressed by the companies led us to formulate the problem as a search problem in the space of possible alternative rankings of the requirements. The possibility to configure the search space and the prioritisation process allows the analyst to identify and control the range of possible requirements rankings, and therefore, the possible alternative ways of evolving the set of requirements and the design and implementation of the system with respect to his/her perspective. Within the space of alternatives, optimal solutions are considered those alternative rankings that better meet the decision-makers preferences and user feedback maximising the consensus. Limiting the choice to such optimal solutions should improve the overall system quality by selecting the best requirements, and at the same time, reduces the effort of the decision-makers that can focus only on relevant information during their decision process.

The results of the application of the approach in the industrial settings allowed us to collect empirical evidence about the performance of the algorithm and, more importantly, gave indications on the practical usage of the proposed approach in the context of real agile processes. The feedback in terms of its efficiency is encouraging, especially considering the aspect related to the time saved by the decision-makers with respect to the decision processes that are currently in place in the two analysed companies.

3.5 The Two Use Cases in a Requirements Management Process

The use cases described above provide examples of applications of AI techniques in support of RE tasks that are performed in the first and second phase of the requirements life cycle, as depicted in Fig. 3.9 (left). They can be integrated with other tools to support the entire life cycle as done, for instance, in the SUPERSEDE⁵ platform [22], which supports planning new software releases on the basis of the analysis of user feedback and usage data, see Fig. 3.9 (right).

Table 3.4 Time spent (in minutes) on the various tasks for the two enterprises

	Time spent (mins)			
	T_{setup}	T_{prio}	T_{close}	Total time
SME	76	18	10	104
LE	96	20	5	121

⁵ Supporting evolution and adaptation of PERsonalized Software by Exploiting contextual Data and End-user feedback, an H2020 EU funded project, 2015–2018.

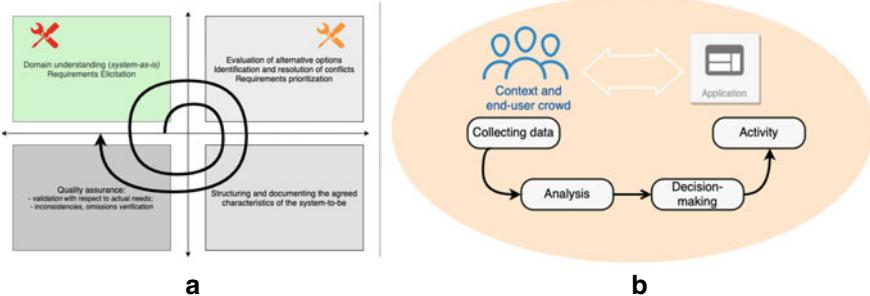


Fig. 3.9 a Use cases in the requirements life cycle; b tool chain in the SUPERSEDE platform

The data-driven software evolution process of SUPERSEDE consists of the following four main steps that are performed in a loop:

- Collecting data: run-time and context data are collected, together with explicit user feedback that the user might deliver upon having used the software. The feedback gathering tool of SUPERSEDE integrates multi-modal feedback gathering techniques, which allow users to express feedback as a combination of textual comments, emoticons, ratings, and pictures, with flexible and configurable monitoring components that collect data from the context and system usage [21]. The collected data are stored in big data storage, which includes a semantic model of the software application domain at the support of data analysis [20].
- Analysis: the purpose of the analysis of the collected data is that of supporting software evolution decisions. Different analysis techniques are provided by the SUPERSEDE tool-suite, including NLP, sentiment analysis, speech-act based analysis, and ML classification techniques, which can be combined for extracting feature requests and bug reports from user textual comments, as described in Sect. 3.3. The tool-suite also supports developers to perform combined analyses of end user feedback and contextual data [21].
- Decision-making: data analysis supports project managers and the development team to take informed decisions about how to evolve the software applications. For instance, identifying new requirements and prioritising them with respect to multiple criteria, as described in Sect. 3.4.
- Activity: the decisions taken (i.e., prioritised requirements) are used as input to define a concrete release plan that takes into account available resources, deadlines, and organisation's priorities [1].

This data-driven process can be customised depending on specific project factors, such as project size, scope, and privacy issues. Guidelines for performing such customisation are given by a methodology that builds on Situational Method Engineering, an engineering approach where a method can be described as a composition of reusable components called method chunks [9]. Indeed, this data-driven tool-supported process has been instantiated in three different industrial case studies

during the SUPERSEDE project, two medium-large enterprises and one small enterprise, with the purpose of conducting an evaluation on industrial case studies. This empirical evaluation was designed according to a customised version of the ISO/IEC SQuaRE [15] quality-in-use model, and a set of quantitative and qualitative metrics were defined to assess effectiveness, efficiency, and acceptance of the tool-supported process against predefined KPIs. Overall, the evaluation confirmed that developers can get benefits by using the tools, reducing the need for manual tasks and lowering their effort. Most importantly, ideas for consolidating and extending the tools were collected.

3.6 Discussion

In recent years, we experienced an increasing interest of companies and the RE research community in the introduction of AI techniques in the software development process and in requirements management activities in particular. In our study, we had the possibility to verify that AI can support and improve traditional and agile requirements management processes. In fact, AI may enable new ways of performing the activities related to requirements management thanks to the possibility of analysing and reasoning on huge quantities of complex and unstructured data, such as user feedback, and promoting the use of the results of this analysis in collaborative decision-making activities, such as those related to the prioritisation of requirements. It is also important to notice that AI techniques may support the recognition of relevant aspects to be considered in the decision-making process, so increasing the awareness of decision-makers on important problems to be addressed that are difficult to be detected without the use of these techniques.

Moreover, the use of AI techniques facilitated the integration of different sources of knowledge to support decision-making, some of them internal to the organisation, in general structured and easy to access, and others external, such as the feedback from the users, that can be very difficult to process without automated techniques.

The companies involved in our study promoted the use of AI-based techniques in their internal processes also thanks to the possibility of using tools and techniques that simplify the exploitation of such methods.

In the near future, we expect an extended use of novel AI techniques to be applied in the requirements management activities. Recent trends in RE research have seen the application of deep learning (DL) techniques for various RE activities, as presented in a recent mapping study by Zhao et al. [28]. On the other hand, human-based approaches are also proposed for classifying user feedback in RE. One such approach makes use of crowdsourcing and outlines how crowdworkers could be used for the purpose of identifying RE-relevant information from user feedback [26].

These new methods and techniques will probably enable novel activities in the requirements process other than those related to decision-making. An example in this direction is the possibility of translating requirements specified in natural language into models or formal requirements representations [8], hence supporting the analysts

in managing problems such as the language ambiguity or the inconsistency of a set of requirements.

3.7 Conclusions

In this chapter, we introduced the Requirements Engineering problem and focused on two specific use cases supporting decision-making related to the requirements elicitation from user feedback and to requirements prioritisation.

We illustrated some solutions to the two problems that are based on metaheuristics and AI techniques, specifically machine learning, natural language processing, and genetic algorithms. The solutions have been evaluated in the field in industrial settings highlighting the importance of applying AI solutions to existing organisational problems.

Based on the two use cases, we have also glimpsed at recent trends to support requirements management activities by exploiting AI techniques based on optimisation, machine learning, and deep learning, with the objective of improving the overall efficiency of the process and to facilitate the work of the analysts and decision-makers performing those activities.

We discussed the use cases in the context of the RE management process, highlighting opportunities and limits of the AI approaches and future possibilities to apply AI in many other activities of the RE process.

Finally, a replication package containing all the tools and datasets presented in this chapter is available on Github at the following address⁶: <https://github.com/se-fbk/ai4re>.

Acknowledgements Part of this work is a result of the SUPERSEDE project, funded by the H2020 EU Framework Programme under agreement number 644018 (<https://cordis.europa.eu/project/id/644018>).

References

1. D. Ameller, C. Farré, X. Franch, A. Cassarino, D. Valerio, V. Elvassore, Replan: a release planning tool, in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (IEEE, 2017), pp. 516–520
2. B.W. Boehm, Software engineering. *IEEE Trans. Comput.* **25**(12), 1226–1241 (1976). <https://doi.org/10.1109/TC.1976.1674590>
3. N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority oversampling technique. *J. Artif. Intell. Res.* **16**(1), 321–357 (2002). <https://www.jair.org/media/953/live-953-2037-jair.pdf>

⁶ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

4. F. Dalpiaz, N. Niu, Requirements engineering in the days of artificial intelligence. *IEEE Softw.* **37**(4), 7–10 (2020)
5. K. Deb, K. Deb, *Multi-objective Optimization* (Springer US, Boston, MA, 2014), pp. 403–449. https://doi.org/10.1007/978-1-4614-6940-7_15
6. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comp.* **6**, 182–197 (2000)
7. A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing* (Springer Science & Business Media, 2003)
8. A. Fantechi, A. Ferrari, S. Gnesi, L. Semini, Requirement engineering of software product lines: extracting variability using NLP, in *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*. ed. by G. Ruhe, W. Maalej, D. Amyot (IEEE Computer Society, 2018), pp. 418–423. <https://doi.org/10.1109/RE.2018.00053>
9. X. Franch, J. Ralyté, A. Perini, A. Abelló, D. Ameller, J. Gorroño-goitia, S. Nadal, M. Oriol, N. Seyff, A. Siena, A. Susi, A situational approach for the definition and tailoring of a data-driven software evolution method, in *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings* (2018), pp. 603–618. https://doi.org/10.1007/978-3-319-91563-0_37
10. A. Fuxman, L. Liu, J. Mylopoulos, M. Roveri, P. Traverso, Specifying and analyzing early requirements in tropos. *Requir. Eng.* **9**(2), 132–150 (2004). <https://doi.org/10.1007/s00766-004-0191-7>
11. P. Giorgini, J. Mylopoulos, R. Sebastiani, Goal modeling and reasoning in tropos, in *Social Modeling for Requirements Engineering*. ed. by E.S.K. Yu, P. Giorgini, N.A.M. Maiden, J. Mylopoulos (MIT Press, Cooperative information systems, 2011), pp. 645–668
12. M. Glinz, On non-functional requirements, in *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India* (IEEE Computer Society, 2007), pp. 21–26. <https://doi.org/10.1109/RE.2007.45>
13. E.C. Groen, N. Seyff, R. Ali, F. Dalpiaz, J. Dörr, E. Guzman, M. Hosseini, J. Marco, M. Oriol, A. Perini, M.J.C. Stade, The crowd in requirements engineering: the landscape and challenges. *IEEE Softw.* **34**(2), 44–52 (2017). <https://doi.org/10.1109/MS.2017.33>
14. ISO/IEC 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Technical report, ISO (2011)
15. ISO/IEC 25000:2014 “Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE” (2014)
16. M.G. Kendall, A new measure of rank correlation. *Biometrika* **30**(1/2), 81–93 (1938)
17. W. Maalej, M. Nayebi, T. Johann, G. Ruhe, Toward data-driven requirements engineering. *IEEE Softw.* **33**(1), 48–54 (2016). <https://doi.org/10.1109/MS.2015.153>
18. I. Morales-Ramirez, F.M. Kifetew, A. Perini, Speech-acts based analysis for requirements discovery from online discussions. *Inf. Syst.* **86**, 94–112 (2019). <https://doi.org/10.1016/j.is.2018.08.003>
19. I. Morales-Ramirez, A. Perini, M. Ceccato, Towards supporting the analysis of online discussions in OSS communities: a speech-act based approach, in *Information Systems Engineering in Complex Environments - CAiSE Forum 2014, Thessaloniki, Greece, June 16-20, 2014, Selected Extended Papers*, ed. by S. Nurcan, E. Pimenidis. Lecture Notes in Business Information Processing, vol. 204 (Springer, 2014), pp. 215–232. https://doi.org/10.1007/978-3-319-19270-3_14
20. S. Nadal, V. Herrero, O. Romero, A. Abelló, X. Franch, S. Vansummeren, D. Valerio, A software reference architecture for semantic-aware big data systems. *Inf. Softw. Technol.* **90**, 75–92 (2017)
21. M. Oriol, M. Stade, F. Fotrousi, S. Nadal, J. Varga, N. Seyff, A. Abello, X. Franch, J. Marco, O. Schmidt, Fame: supporting continuous requirements elicitation by combining user feedback and monitoring, in *2018 IEEE 26th International Requirements Engineering Conference (RE)* (IEEE, 2018), pp. 217–227

22. A. Perini, Data-driven requirements engineering. the SUPERSEDE way, in *Information Management and Big Data, 5th International Conference, SIMBig 2018, Lima, Peru, September 3-5, 2018, Proceedings. Communications in Computer and Information Science*, vol. 898, ed. by J.A. Lossio-Ventura, D. Muñante, H. Alatrista-Salas (Springer, 2018), pp. 13–18. https://doi.org/10.1007/978-3-030-11680-4_3
23. A. Perini, A. Susi, P. Avesani, A machine learning approach to software requirements prioritization. *IEEE Trans. Softw. Eng.* **39**(4), 445–461 (2013). <https://doi.org/10.1109/TSE.2012.52>
24. I. Sommerville, *Software Engineering*, International computer science series, 8th edn. (Addison-Wesley, 2007). <https://www.worldcat.org/oclc/65978675>
25. A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications* (Wiley, 2009). <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000863.html>
26. M. van Vliet, E.C. Groen, F. Dalpiaz, S. Brinkkemper, Identifying and classifying user requirements in online feedback via crowdsourcing, in *Requirements Engineering: Foundation for Software Quality - 26th International Working Conference, REFSQ 2020, Pisa, Italy, March 24-27, 2020, Proceedings [REFSQ 2020 was postponed]*, ed. by N.H. Madhavji, L. Pasquale, A. Ferrari, S. Gnesi. Lecture Notes in Computer Science, vol. 12045 (Springer, 2020), pp.143–159. https://doi.org/10.1007/978-3-030-44429-7_11
27. C.A. Gunter, E.L. Gunter, M. Jackson, P. Zave, A reference model for requirements and specifications. *IEEE Softw.* **17**(3), 37–43 (2000)
28. L. Zhao, W. Alhoshan, A. Ferrari, K.J. Letsholo, M. Ajagbe, E.V. Chioasca, R.T. Batista-Navarro, Natural language processing (nlp) for requirements engineering (re): A systematic mapping study. *ACM Computing Surveys* (2020)
29. L. Zhao, W. Alhoshan, A. Ferrari, K.J. Letsholo, M.A. Ajagbe, E. Chioasca, R.T. Batista-Navarro, Natural language processing (NLP) for requirements engineering: a systematic mapping study. *CoRR* (2020). <https://arXiv.org/abs/2004.01099>

Chapter 4

Leveraging Artificial Intelligence for Model-based Software Analysis and Design



**Antonio Garmendia, Dominik Bork, Martin Eisenberg, Thiago Ferreira,
Marouane Kessentini, and Manuel Wimmer**

Abstract Fundamental decisions are made in the early phases of software development. The typical outcomes of these phases are models of different kinds, such as architectural models, data models, and process models. Automation support is required to efficiently and effectively handle large models and conduct continuous quality improvement processes. Thus, several approaches have been proposed that integrate modeling with Artificial Intelligence (AI) methods such as Genetic Algorithms (GAs), among others. These approaches, e.g., transform models to improve their quality by searching for good solutions within the potential solution space. In this chapter, we first review existing applications of AI methods to model-based software engineering problems. Subsequently, we show a representative use case of how a model-based software analysis and design problem can be solved using GAs. In particular, we focus on the well-known and challenging modularization problem: splitting an overarching, monolithic model into smaller modules. We present two

A. Garmendia (✉)

JKU Linz, Institute of Business Informatics - Software Engineering,
Altenberger Straße 69, 4040 Linz, Austria

e-mail: antonio.garmendia@jku.at

D. Bork

TU Wien, Business Informatics Group, Favoritenstrasse 9-11, 1040 Vienna, Austria
e-mail: dominik.bork@tuwien.ac.at

M. Eisenberg · M. Wimmer (✉)

JKU Linz, CDL-MINT, Institute of Business Informatics - Software Engineering,
Altenberger Straße 69, 4040 Linz, Austria

e-mail: manuel.wimmer@jku.at

M. Eisenberg

e-mail: martin.eisenberg@jku.at

T. Ferreira

University of Michigan-Flint, College of Innovation & Technology, Michigan, USA
e-mail: thiagod@umich.edu

M. Kessentini

Oakland University, School of Engineering and Computer Science, Michigan, USA
e-mail: kessentini@oakland.edu

encodings, the model-based and the transformation-based encoding, which are both applied for the modularization of Entity-Relationship (ER) diagrams. We further discuss how these encodings may be adapted to other structural models and conclude with an outlook on future research lines related to software modeling intelligence.

4.1 Introduction

Analysis and design phases comprise all engineering activities required to develop systems based on customer requirements [1]. The success of the software product depends on the decisions made within these phases. For instance, requirements engineering, software architecture design, and user-interface design are critical and complex tasks in which engineers can benefit from Artificial Intelligence (AI) methods to ease the burden of work [2].

Often, the outcomes of some early phases' engineering activities are models. In particular, the design of Entity-Relationship (ER) models for database design [3], structural models for defining components and connectors, and behavioral models for designing how a system changes over time [4] are frequently used. This is due to the fact that modeling approaches are an important cornerstone to communicate about engineering tasks, like adding new functionality, refining, restructuring, or fixing bugs in a semi-formalized way [5, 6]. Especially in inter-disciplinary settings such as the development of Cyber-Physical Systems (CPS), models provide a bridge between the different disciplines to discuss and consolidate designs [6].

Over time, the modeled systems evolve due to, e.g., continuous development, addition of new features, and improving existing ones. Consequently, models may become already too large, and engineering tasks, e.g., concerning refactoring or model repair, may become too complex to be solely performed based on manual inspection and revision. Therefore, this growing complexity of software systems and associated models demands the creation of approaches and complementary tools that assist engineers with human-centered activities and provide a higher level of automation. The use of such tools may lead to improvements with respect to the quality of the models and the systems as well.

There exists a long history of the application of Artificial Intelligence (AI) methods to software engineering problems [7], also including the early phases of analysis and design [8]. In this context, the research fields of Search-based Software Engineering (SBSE) [9] and Model-Driven Engineering (MDE) [10] are of utmost importance. While SBSE converts engineering activities into optimization problems by searching for accurate solutions within the whole range of possible solutions [9], MDE tackles the complexity of software systems by applying abstraction and automation based on models and model transformations, respectively [11]. Thus, their combination seems promising to solve challenging analysis and design problems in large-scale industrial

settings.¹ From the optimization point of view, several model-based engineering problems may be formulated as *single objective* or *multiple objective* problems. The former approach uses just one objective (or multiple objectives are aggregated into a single fitness value which can lead to the loss of optimal results [12, 13]). The latter approach uses sophisticated algorithms which are fast and efficient when considering all objectives simultaneously. The outcome is a set of optimal solutions, i.e., the *Pareto set*, which allows to reason about trade-offs between different solutions. This is especially useful in the early phases of software development where several such decisions have to be made.

In this book chapter, we provide different strategies of how to apply AI techniques within the model-based software analysis and design context. In particular, we revisit two different encoding strategies: model-based and transformation-based, which are both suitable for optimization based on GAs. We apply both encodings to the modularization problem, i.e., splitting a large database schema into manageable pieces which can be assigned for design and maintenance to a developer or a developer team [14, 15], by formulating it as a multi-objective optimization problem. Based on these examples, we generalize a comparative discussion regarding the possibilities of adapting the approaches to other domains/problems/modeling languages.

Chapter organization. Section 4.2 provides the background regarding MDE, introduces the running example of this chapter: the modularization of ER diagrams, and summarizes selected applications of AI for model-based engineering. In Sect. 4.3, we focus on describing the model-based and transformation-based encoding strategies. In addition, we describe their usefulness by implementing solutions for modularizing ER diagrams. Finally, Sect. 4.4 concludes with future research lines.

4.2 Background

In this section, we introduce the foundations of MDE as well as the running example of this chapter before reporting on selected applications of AI in this particular field.

¹ Please note that we use the terms “model-based” and “model-driven” in an interchangeable manner in this book chapter. While model-driven approaches are focusing on using models as the main driver in the software engineering process, they can be also considered as model-based approaches which use models as supporting artefacts but not as the main driver. However, as we mostly focus in this book chapter on the techniques proposed by the model-driven engineering community, these techniques can be of course also used in a model-based setting.

4.2.1 Model-Driven Engineering: Models, Meta-models, and Model Transformations

Model-Driven Engineering (MDE) is a software development paradigm that integrates the concepts of Modeling Languages (MLs) to represent models and Model Transformations (MTs) to manipulate models [11]. Models in this context are being used in different software engineering tasks such as analysis and design, but in principle, can also be used in later phases such as validation and verification or reverse engineering of existing applications [10].

4.2.1.1 Modeling Languages

In MDE, General-purpose Modeling Languages (GPMLs), such as UML or SysML, can be differentiated from Domain-Specific Modeling Languages (DSMLs), such as ER or BPMN [10]. While the former benefit from wide adoption and the availability of mature tools, the latter provide languages that are particularly designed to account for the specific characteristics of an application domain and to support specific modeling purposes such as code generation. Irrespective of being general-purpose or domain-specific, MLs in MDE consist of three fundamental components: *abstract syntax*, *concrete syntax*, and *semantics*. The abstract syntax is defined by means of a meta-model that captures the relevant domain concepts [5]. In this context, models are also seen as graph-based representations that have to conform to the meta-model. With regard to the concrete syntax, usually the meta-model is taken as an input to build a textual or graphical notation, and sometimes even a combination of textual and graphical notations is developed. In Sect. 4.2.2, we will describe the abstract syntax of ER diagrams (Fig. 4.1) as well as a concrete graphical representation (Fig. 4.4).

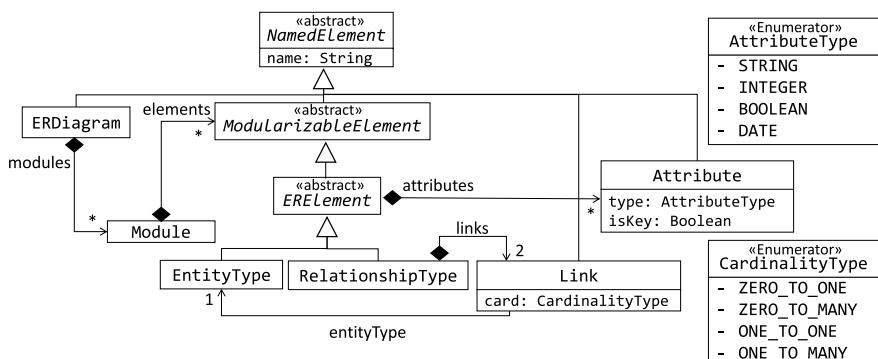


Fig. 4.1 Excerpt of the Entity-Relationship diagram meta-model

4.2.1.2 Model Transformations

Besides MLs, another key concept of MDE is Model Transformations (MTs) [16]. MTs are programs that take one or more models as an input, apply transformation rules, and produce one or more models as an output [17]. In this context, MTs allow the rule-based manipulation of artefacts (e.g., refactoring) or the generation of entirely new artefacts based on the application and execution of transformation rules on source artefacts (e.g., code generation). While the former is referred to as in-place transformations, the latter is known as out-place transformations [18]. An important characteristic of MTs is that they are realized on the meta-model of a modeling language which enables their application on any model that conforms to that meta-model. Due to the fact that models often follow a graph-based structure (e.g., see UML class diagrams, state machines, or process models), MT languages often rely on graph transformations (i.e., exploiting the underlying graph structures of models). In this sense, models are considered as graphs whereas the corresponding meta-models are considered as typed graphs [19]. In general, a graph transformation rule consists of a left-hand side (LHS) and a right-hand side (RHS). The LHS describes patterns to be matched on the graph (pre-conditions), while the RHS describes the effect of the changes to be applied on the graph (post-conditions). To apply a rule, a graph transformation engine finds a match for the LHS in the input graph and executes the RHS to add, update, and remove elements in the graph. So to speak, the result is a new graph on which further rules may be applied.

4.2.1.3 Selected Technologies for this Book Chapter

For the efficient application of MDE, several widely used tools such as JetBrains Meta Programming System (MPS),² MetaEdit+ [20], and Eclipse Modeling Framework (EMF) [21] are available. In this book chapter, we provide examples and tooling based on EMF. The abstract syntax of the ER language is defined in Ecore, the meta-modeling language of EMF. For MTs, we use Henshin [22] which enables the specification and execution of in-place MTs based on graph transformation concepts for EMF models. However, it should be also noted that there are many frameworks and languages to define MTs, e.g., see [23] for a recent survey. Next, we show the concrete application of EMF and Henshin for the ER modularization problem.

4.2.2 Running Example

For this book chapter, we selected the modularization problem for ER diagrams which has been also studied by several researchers in the past [15, 24–28]. We selected this example as ER diagrams are widely known by the computer science

² <https://www.jetbrains.com/mps/>.

community, and the general problem may be easily transferred to other structural modeling languages such as UML class diagrams and architectural models such as C&C diagrams. Finally, the modularization of an ER diagram can be concisely described as an in-place MT which helps to demonstrate the different encodings we present in this chapter.

For real-world systems, ER diagrams may soon become large and complex if they are represented by a single, monolithic diagram. Such a monolith representation no longer supports human comprehension which is one of the main goals of creating ER diagrams. As a mitigation, modularization of the overarching ER diagram into smaller modules has been proposed [15, 24–28].

MTs are applicable to automate the modularization process. As a basis to specify the necessary transformations, the abstract syntax of the ML (i.e., the ER modeling language in our example) needs to be defined. Figure 4.1 shows an excerpt of the ER meta-model in terms of the Ecore language [21] supported by EMF. The Ecore language follows the concepts of object-oriented modeling, you may think of it as the core of the UML class diagram language. Notably, the meta-model in Fig. 4.1 comes with an extension of ER diagrams which supports the modularization of large ER models into smaller modules. This is the reason why the meta-model comprises classes like `ModularizableElement` and `Module`.

Each class in the ER meta-model is inherited from the abstract class `NamedElement`, i.e., each class inherits a `name` attribute of type `String`. The `ERDiagram` class composes all objects of type `Module`. Each `Module` composes all entities (class `EntityType`) and relations (class `RelationshipType`) that are generalized into the abstract classes `ModularizableElement` and `ERElement`, the latter class further composing `Attributes`. Each `RelationshipType` further composes `Links` which specify the `CardinalityType` of a relationship.

Based on the ER meta-model described above, we developed an MT for performing the modularization. For instance, we defined the Henshin rule in Fig. 4.2. Please note that Henshin does not explicitly define LHS and RHS. Instead, Henshin supports a concise representation that merges both sides (LHS and RHS) into one single representation. For this purpose, Henshin is using stereotypes to mark the elements as *preserve*, *created*, or *deleted*. In Henshin, a transformation is defined by a set of rules. The rules may have an arbitrary number of parameters that need to be instantiated when applied. However, these parameters may be unset, and then, the Henshin engine is able to randomly compute matches for the parameters.

The rule shown in Fig. 4.2 moves an entity with a name equal to the name received as a parameter (i.e., `entityName`) to a newly created module whose name will be equal to the parameter `newModuleName`. When this rule is executed, it first aims to find a node of type `ERDiagram` (`node <<preserve>>:ERDiagram`), that composes a module (`node <<preserve>>:Module`), that itself composes an entity (`node <<preserve>>:EntityType`) with a name that is equal to the parameter `entityName`. Upon finding a match adhering to these characteristics, the rule creates a new module with a name equal to the value of `newModuleName` and then moves the matching entity to this module. Specifically, the move is performed by deleting the composition relationship between the entity

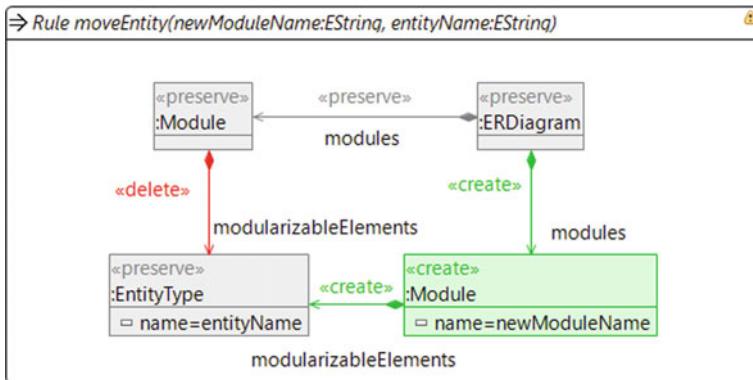


Fig. 4.2 Henshin rule that extracts an EntityType to a new module

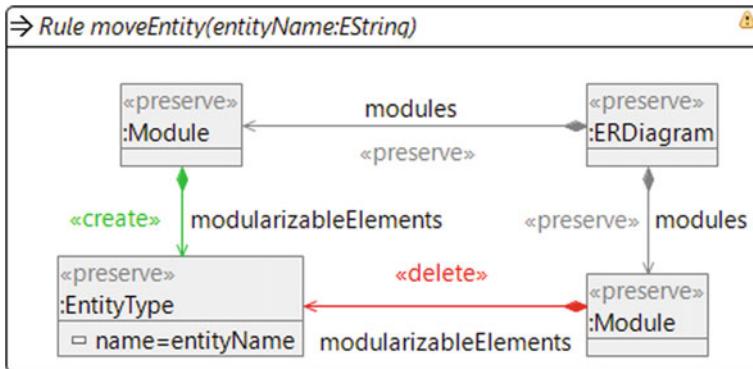


Fig. 4.3 Henshin rule that moves an EntityType to a different module

and the Module that contains it, and by creating a new composition relationship between this entity and the newly created Module.

Figure 4.3 defines another rule based on the ER meta-model, that moves an entity from its current module to another existing one. In theory, having these two Henshin rules enables to produce all possible solutions of modularizations. Of course, there may be other transformation rules used for computing the modularizations, e.g., using merge and split rules for modules or one could even reason about which rules are in general possible to be defined for a given language to optimize models [29].

An application of the rule shown in Fig. 4.2 can be observed in Fig. 4.4. The example model is represented in Fig. 4.4 (label 1) by using the UML object diagram notation [4] which is the computer-internal representation laying behind the graphical visualization of models for the human user. The actual visualizations of the source model and target model are below their object notations. The source model is an excerpt of a college database that contains a module with two entities (Student and Course) related by the Enrolls relationship. The execution (label 2) instantiates

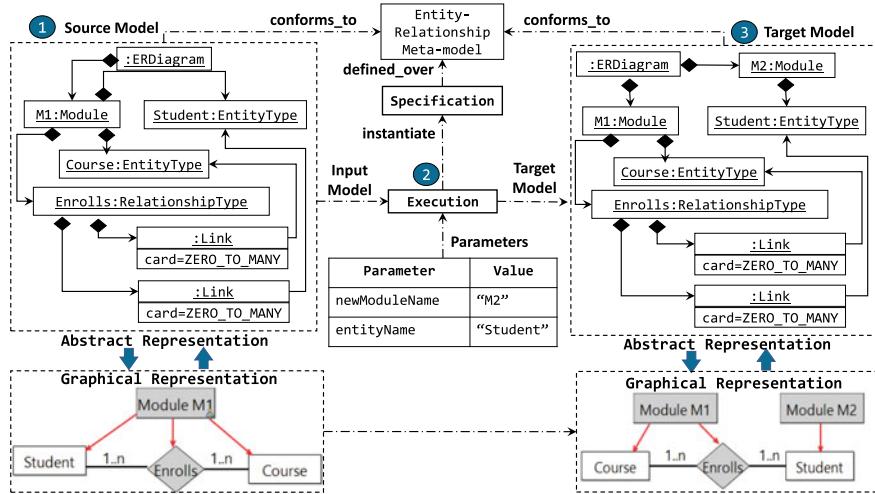


Fig. 4.4 Application of the Henshin rule on the example ER college database model

the specification of the rule by passing the parameters. The composition relationship between M1 and Student has been deleted in the produced target model (label 3). Moreover, M3 composes a new module named M2 that itself composes the entity Student.

Now the question arises what is a good transformation for a given source model. Thus, the following subsection will describe how the modularization of ER diagrams, and transformations in general, may be framed as an optimization problem.

4.2.3 Selected Applications of AI for Model-Based Engineering Problems

We now summarize recent work related to the application of AI in MDE. We specifically focus on AI-based solutions for modularizing software systems and models as well as on AI-based solutions for solving MT problems in general. The latter is of interest as the modularization of models can be framed as an MT as we have seen in the previous subsection.

4.2.3.1 AI-based Model Modularization

The modularization of overarching data models, mostly represented by ER diagrams, has been researched for decades. The first works have been introduced in the late 1980s by Feldman and Miller [30] and Simsion [31]. In the 1990s, among oth-

ers, Daniel Moody contributed significant works toward modularizing large ER diagrams [15, 24, 25]. Since 2000, research in this field did not decline with ongoing contributions that are based on applying AI techniques like GAs in combination with MDE techniques like MTs [26]. A differentiation can be made between approaches that produce *hierarchically structured* (e.g., [27]) and ones producing *flat* modularizations (e.g., [26]). Moreover, redundant (e.g., [25]) and non-redundant modularizations (e.g., [32]) as well as modularizations that aim to optimize a single objective (e.g., [32, 33]) and ones that aim to optimize multiple objectives (e.g., [28]) can be distinguished.

There exist several studies that employ SBSE [9] techniques to modularize software models [34, 35]. For instance, Mkaouer et al. [36] propose a many-objective search-based approach based on NSGA-III where the goal is to find the optimal re-modularization solutions (a system may already have a module structure which should be further improved) that optimize four objective functions, among them improving the package structure and minimizing the number of changes. In addition, Kessentini et al. [37] propose a multi-objective approach by using NSGA-II that dynamically adapts and interactively suggests edit operations to developers, and takes their feedback into consideration when addressing model evolution. Finally, and more recently, Kessentini and Alizadeh [38] extend the previous work by providing to the user, regions of interest, by clustering the set of recommended solutions. Thus, users can quickly provide their preferred cluster and give feedback on a smaller number of solutions. Finally, please note that the modularization problem can be considered as a refactoring problem. For a recent survey on this topic, we kindly refer the interested reader to [39].

4.2.3.2 AI-based Model Transformation

In recent years, AI methods have been applied for MTs in general [29, 40–42]. As noted already, modularization may be seen as a special kind of MT. Thus, these approaches may be applied as well for this particular problem but also for many others which can be framed as an MT problem. Specifically, model-driven optimization has recently gained much interest in the MDE community [29, 40–42]. The core idea is to optimize models by applying a set of predefined MT rules. This process is usually guided by meta-heuristic searchers such as genetics algorithms. Usually, most approaches are focusing on in-place transformations when it comes to model-driven optimization.

One of the first works that describe MTs as an optimization problem is the one of Kessentini et al. [43]. This approach describes out-place MTs as a combinatorial optimization problem, in which an optimal output model is reached by considering existing input/output model pairs. Of course, the number of possible solutions is huge; therefore, a deterministic search is unpractical.

In the Transformation Tool Contest (TTC) in 2016 [44], eight approaches were presented providing solutions for the Class Responsibility Assignment (CRA) problem [45]. The CRA problem is a software design problem in which one has to produce

a high-quality class diagram by taking as an input the set of methods and attributes as well as their relations. In this contest, six of the presented solutions draw on meta-heuristic algorithms for solving this problem, e.g., by combining transformation engines with meta-heuristic search algorithms or coordinating GAs with MTs.

4.2.3.3 Synopsis

To sum up, the application of AI techniques to MTs in general, and the modularization problem in particular, gained a lot of attraction in the last decades. Thus, we will present in the following section how the most used technique, namely GAs, can be applied for our running example.

4.3 Optimizing Models with AI Techniques: Two Encodings for the Modularization Case

In this section, we describe two distinct strategies for optimizing models with AI techniques: *model-based* and *transformation-based* encodings. We make use of the ER diagram modularization case which has been introduced previously and show how this case can be supported by the application of GAs as a particular AI technique. More specifically, we show how models and transformations can be encoded in order to apply GAs to solve the ER diagram modularization problem.

4.3.1 Model-based versus Transformation-based Encodings: An Overview

Before we start explaining how models and transformations are encoded for applying GAs, we shortly revisit how GAs work in general and introduce all the components to consider when implementing GAs. The core component that should be designed is the encoding. The GA encoding either represents the sequence of MTs to produce the output model or the actual model to be optimized. Next, we revisit the steps necessary to apply GAs for a particular encoding. Starting from the encoding, we can generate an *initial population* of MTs or models. Then, we perform the *evaluation* of each individual (also called *chromosome*) by calculating the values for all fitness functions. If the ending condition is not triggered, we use a selection operator in which individuals are selected based on their fitness function values. This means that individuals with higher fitness function values have a higher probability to be chosen. The selected individuals are passed to the next stage in which new individuals will be created using genetic operators such as *crossover* and *mutation* operators. Sometimes, it may happen that genetic operators create invalid individuals. In this

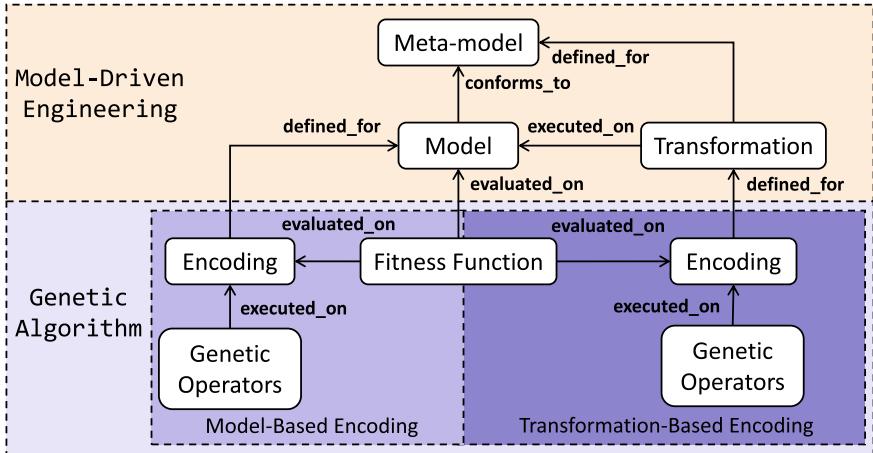


Fig. 4.5 Using GAs for MDE artefacts: encoding models versus transformations

case, we need to implement a repair function to convert those invalid individuals into valid ones.

Figure 4.5 shows the mapping between MDE components and GA ones. In terms of the encoding scheme, several encodings have been proposed in the past for solving model-based engineering tasks with GAs. First of all, let us recall the artefacts we have in model-based engineering. Models are the primary development artefacts that have to conform to their meta-models which gives a machine-readable and processable representation. Thus, the second main artefacts are transformations that manipulate the models with a certain intent.

The first encoding option is to directly encode models for AI-based techniques. Previously, GAs have been employed intensively. This means a dedicated encoding is required for the models and genetic operators for selection, recombination, and mutation. There exists a multitude of encoding schemes, such as binary, integer, and real-valued representations among others [46]. The encoding must be an actual representation of the models' information and the problem to be optimized. Therefore, choosing the encoding will depend on the problem we are trying to optimize. Consequently, a cautious decision with respect to the best-suited encoding needs to incorporate a comparison of alternative encodings based on the requirements and the problem at hand. Of course, a (set of) fitness function(s) is required to guide the search toward better solutions.

The second encoding option is to encode the transformations which are executed on the models. Again, transformations may be considered as models which have to conform to a particular meta-model, i.e., the transformation language. Thus, transformations can be transformed into particular encodings for AI-based techniques. Several approaches have been presented in the past that encode transformations with GAs. In these approaches, the transformation is considered as a sequence of transformation steps that are described by transformation rules. Thus, a transformation

step corresponds to the application of a transformation rule. The rule applications are computed by an MT engine and stored in vectors of transformation steps. Of course, as for the model-based encoding, dedicated genetic operators are required to evolve the found solutions with selection, recombination, and mutation. The fitness functions evaluate the transformed models and guide the evolution of the transformation sequences. However, additional fitness functions may be defined directly on the transformation sequences such as minimizing the length of the computed transformation sequences, i.e., a short sequence is considered better compared to a longer one that yields similar results. To sum up, the second encoding approach provides the indirect optimization of the models by searching for the best transformation sequences producing the best output models.

4.3.2 Model-based Approach

We now show how a model-based strategy is applied to modularize ER models. The description is structured according to and instantiates the generic constituents of GA approach. The entire approach, including a more comprehensive discussion of the rationale behind the chosen genetic operators and a twofold evaluation, can be found in the original publication that introduces the approach [26].

4.3.2.1 Objectives

We aim to optimize the modularization of an ER diagram by following five distinct objectives that have been initially proposed by Moody et al. [15]. Please note that similar objectives, subsets of the objectives, and even additional objectives have been proposed in other works such as [36]. Thus, a user may select the ones which are most preferred in a particular setting.

We summarize the five objectives we employ for this book chapter in the following.

- **Cohesion:** This objective is measured by the sum of links within a module. The GA should aim to maximize cohesion in order to group ModularizableElements that are related to each other in one module. Thus, we first need to define the measurement for links in an ER diagram as follows:

$$isLinked(e_i, r_j) = \begin{cases} 1 & \text{if entity } e_i \text{ has a Link to a relationship } r_j \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

where e represents an entity in module i and r a relationship r in module j . In order to obtain the sum of links, we define Eq. 4.2 as follows:

$$SumOfLinks(m_i, m_j) = \sum_{\substack{e_i \in E(m_i) \\ r_j \in R(m_j)}} isLinked(e_i, r_j) \quad (4.2)$$

Then, we can define the cohesion based on Eq. 4.2 as follows:

$$\text{Cohesion} = \sum_{m_i \in \text{Modules}} \text{SumOfLinks}(m_i, m_i) \quad (4.3)$$

where the links within the same module are calculated.

- **Coupling:** Coupling is defined as the sum of links between modules. Based also on Eq. 4.2, we can calculate coupling as follows:

$$\text{Coupling} = \sum_{\substack{m_i, m_j \in \text{Modules} \\ m_i \neq m_j}} \text{SumOfLinks}(m_i, m_j) \quad (4.4)$$

This measure can be seen as the opposite of cohesion. Consequently, coupling should be minimized, guiding the GA toward solutions where entities not linked to each other are assigned to different modules.

- **Number of modules:** The objective aims to minimize the overall number of modules in a modularization. In this sense, it is desirable to reward a solution that has fewer modules as this usually benefits the comprehensibility of a design.
- **Module size:** This objective aims to minimize the average number of ModularizableElements per module. Thus, the aim is to reward solutions that have smaller modules on average.
- **Module balance:** The goal of this objective is to produce a solution in which all modules contain a similar number of ModularizableElements. To accomplish this, we propose the use of the standard deviation. This means that the elements should be distributed among the modules as evenly as possible. Therefore, the standard deviation must be minimized.

The objectives introduced above combine structural objectives like cohesion and coupling with objectives solely aiming to account for human comprehensibility like the number and size of modules. Naturally, this leads to conflicts, i.e., between cohesion and coupling. Furthermore, the highest cohesion and coupling can be reached for solutions with only one module whereas this would maximize the average number of ModularizableElements per module. Likewise, minimizing the average number of ModularizableElements per module is inversely proportional to the number of modules.

4.3.2.2 Encoding

In order to represent the chromosome in the population, we represent ER models by the encoding proposed by Rizzi [47]. For the modularization of ER models, it is important to represent, alter, and evaluate the assignment of ModularizableElements to modules and the total number of modules in a solution. The proposed encoding by Rizzi [47] can represent all possible modular combinations. This encoding does not

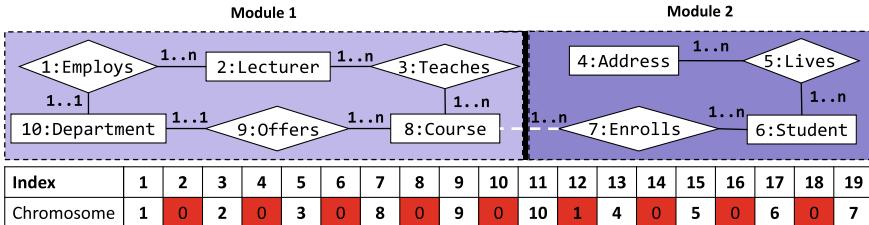


Fig. 4.6 Encoding scheme for ER models using the college database example

represent model elements directly but by their identifiers. We thus need to implement a bi-directional bridge between the model and the encoding.

The Rizzi chromosome representation composes two types of genes: *identifiers* and *separators* which makes it suitable for the requirements of our problem. The genes in odd positions represent the elements by using a unique integer value (i.e., the *identifiers*). The *separators* refer to the genes in even positions that may take a binary value of 0 or 1. If a separator gene at position n is equal to 0, it means that the two elements at positions $n - 1$ and $n + 1$ are in the same module. Otherwise, these two elements will be in different modules. The sum of the *separators* plus 1 is equal to the number of modules. The length of the chromosome is defined by the number of ModularizableElements multiplied by 2, minus 1.

Example. To exemplify the encoding, Fig. 4.6 shows the ER diagram and its corresponding chromosome representation. This ER diagram is an extension of the output college database with two modules depicted in Fig. 4.4. In this example, a Student has at least one, possibly many Addresses in the database, and an Address might be associated to one or many Students. A Lecturer Teaches one to many Courses and a Course is being taught by one to many Lecturers. A Lecturer is employed (Employ relationship) by exactly one Department whereas one Department employs one to many Lecturers. Each Department Offers one to many Courses and a Course is offered by exactly one Department. With respect to the chromosome shown at the bottom of Fig. 4.6, it can be seen that all separators are equal to 0 except the one at position 12. The integer values in even positions correspond to the *identifiers* of the ModularizableElement objects.

4.3.2.3 Genetic Operators: Overview

In the following, we introduce the genetic operators based on the encoding explained in Sect. 4.3.2.2. The approach comprises different kinds of genetic operators for the different types of genes in our chromosome. Specifically, we grouped the *identifiers* in an Enumerable Chromosome and the *separators* in a Bit Chromosome. Then, we can apply well-known crossovers and mutators used in GAs [48].

Figure 4.7 shows a general overview of the chosen genetic operators. At the top of this figure, the same chromosome as in Fig. 4.6 is illustrated. We can observe

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Chromosome	1	0	2	0	3	0	8	0	9	0	10	1	4	0	5	0	6	0	7

↓
Group by
type of genes

Enumerable Chromosome										Bit Chromosome									
Index	1	3	5	7	9	11	13	15	17	19	2	4	6	8	10	12	14	16	18
Chromosome	1	2	3	8	9	10	4	5	6	7	0	0	0	0	0	1	0	0	0
Crossover	Partially-matched Crossover (PMX)										Multi-point Crossover (MX)								
Mutator	Swap Mutator										Flip Mutator								

Fig. 4.7 Encoding and genetic operators grouped by type of genes

the conversion of this chromosome in the Enumerable Chromosome (composed by the genes in odd positions) and the Bit Chromosome (composed by the genes in even positions). By doing this, we can use as genetic operators on the Enumerable Chromosome the Partially Matched Crossover (PMX) and the Swap Mutator. With respect to the Bit Chromosome, we will use a Multi-point Crossover (MX) and Flip Mutator as genetic operators. With the use of these genetic operators, we avoid producing invalid chromosomes.

In order to exemplify the genetic operators, we will use the Enumerable Chromosome and the Bit Chromosome shown on Fig. 4.6 as a baseline. The description of each genetic operator will be given as follows.

4.3.2.4 Crossovers

Partially matched Crossover (PMX): This crossover was proposed by Goldberg et al. [49] as a genetic operator for the Traveling Salesman Problem. Generally, this crossover is used when there must not be duplicated genes in the chromosome. Notice that this is indeed the case for our Enumerable Chromosome (Fig. 4.7) [48].

This crossover takes as input two parents and randomly selects two positions usually called *cut-points*. The genetic information stored within these two cut-points will be exchanged among the parents. Then, it may be possible that the resulting offsprings are invalid, because they may contain duplicate genes. To convert the offspring to valid instances, a mapping table is built with the genes that were exchanged. By using this mapping table, the duplicate genes outside the cut-points can also be exchanged, thereby producing valid offsprings.

Figure 4.8 shows an example of the PMX. It can be seen that in the first step, we selected two chromosomes, in which parent 1 is the same chromosome shown in Fig. 4.7 and parent 2 was randomly created. Also in the first step, the positions 3 and 6 were randomly selected as cut-points. In the second step, the genes within the defined area were exchanged, producing offsprings that are not valid. The invalid genes in each child are highlighted in red to ease their identification. For instance,

Step 1: Select the range

Parent 1	1	2	3	8	9	10	4	5	6	7
Parent 2	1	10	9	8	2	6	5	4	3	7

Step 2: Exchanging the genes within the range

Child 1	1	2	9	8	2	6	4	5	6	7
Child 2	1	10	3	8	9	10	5	4	3	7

Mapping Table
3 \leftrightarrow 9 \leftrightarrow 2
10 \leftrightarrow 6

Step 3: Repair invalid offspring

Child 1	1	3	9	8	2	6	4	5	10	7
Child 2	1	6	3	8	9	10	5	4	2	7

Fig. 4.8 Partially matched crossover example**Step 1: Select the range**

	Seg. 1			Seg. 2			Seg. 3			
Parent 1	0	0	0	0	0	0	1	0	0	0
Parent 2	1	0	0	0	0	1	0	0	1	0

Step 2: Exchanging the genes within the range

Child 1	0	0	0	0	0	1	1	0	0	0
Child 2	1	0	0	0	0	0	0	0	1	0

Fig. 4.9 Two-point crossover example

the Child 1 has the 2 duplicated. Finally, in step three, the mapping table is used to exchange the duplicate genes outside the defined area to ensure the validity of the offsprings.

Multi-point Crossover (MX): This crossover, in contrast to PMX, does not guarantee that each gene has to be unique. Therefore, we use this crossover in our Bit Chromosome. The MX takes as an input two parents and n random positions that split the chromosome into segments. Then, two offsprings are produced by exchanging the genetic information between the segments in even positions. In this example, we will use a two-point crossover, which is an example shown in Fig. 4.9. The exchange of genetic information among the parents within the genes that belong to segment 2 can be easily observed.

Step 1: Select two random positions

1	2	3	8	9	10	4	5	6	7
---	---	---	---	---	----	---	---	---	---

Step 2: Swap the elements

1	10	3	8	9	2	4	5	6	7
---	----	---	---	---	---	---	---	---	---

Fig. 4.10 Swap mutator example**Step 1: Select one random position**

0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---

Step 2: Flip the gene

0	1	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---

Fig. 4.11 Flip mutator example

4.3.2.5 Mutators

Swap Mutator: As in the case of PMX, this mutator is used when the chromosome has unique genes [48]. This mutator randomly selects two genes within the chromosome and then swaps their positions. The appliance of this mutator to our Enumerable Chromosome may potentially move a ModularizableElement from its current module to a different one. Figure 4.10 shows an example of the swap mutator using our Enumerable Chromosome example. It can be observed that the gene 2 changes its position with the gene 10.

Flip Mutator: This mutator is also known as *Bit flip*. It is very common to apply this kind of mutator to Bit Chromosomes. Basically, this mutator randomly selects one gene and flips it. This means that if a gene is equal to 1, then the gene is converted to 0. If this happens in our defined Bit Chromosome, it will mean that two existing modules are merged into one. In our ER example, this mutator either adds or removes a module, thereby also affecting the assignment of ModularizableElements to modules. This can be observed in Fig. 4.11 using our Bit Chromosome. Notice that we replace a 0 with a 1.

4.3.2.6 Tool Support

The implementation of the presented approach is available in the ModulEER tool [26]. This tool is an Eclipse plugin that relies on the EMF framework, Sirius,³ and Jenetics.⁴ Sirius is a plugin that enables the creation of graphical model-

³ <https://www.eclipse.org/sirius>.

⁴ <https://jenetics.io>.

ing environments and has been used to create a graphical ER editor with modularization support. Jenetics is a Java library to implement evolutionary algorithms and has been used for realizing the presented encoding and genetic operators. The interested reader can use ModulEER to create ER diagrams or use the already provided college database model and execute the implementation of the presented model-based approach to get a Pareto set of modularization solutions. All implementation artefacts with installation instructions are provided on GitHub.⁵

4.3.3 Transformation-based Approach

In this subsection, we outline a transformation-based encoding presented in previous research on MOMoT⁶ [50]. As for the model-based encoding, we summarize how to define the objectives, encoding, and the genetic operators based on the ER diagram modularization problem.

4.3.3.1 Objectives

The objectives presented before in Sect. 4.3.2.1 are reused also for the transformation-based encoding. However, since we encode the transformation rules, the objectives are evaluated on the produced models after executing these rules on the initial model with the help of the Henshin transformation engine. Basically, these objectives are considered as problem-oriented objectives in order to produce an optimal output for a given problem. In addition to the problem-oriented objectives, MOMoT supports solution-oriented objectives which are also evaluated on the encoding, i.e., the transformation rules. Of course, finding a transformation that leads the model to its best possible state with respect to the earlier-mentioned quality criteria is paramount. At the same time, other criteria unrelated to the domain may benefit the search at exploring the design space. For instance, it is arguable to consider shorter transformation rule application sequences better compared to longer ones yielding the same results. This is because the chromosome will be easier to inspect and understand by humans. Besides, the search may become more efficient by having shorter sequences, given that the execution of the rules over the models may take less time. Regardless of the task, such additional objectives lead to multi-dimensional search spaces in which the demand to steer the optimization process emerges. To this end, in the chromosome, we may find units that do not modify the model itself. This type of unit is called *Placeholders* and will be explained in the next section.

It is possible that problem-specific and solution-specific goals are in contradiction. This may happen, e.g., when the solution length is not sufficient to reach solutions that are good with respect to the problem-oriented objectives.

⁵ <https://github.com/jku-win-se/ai-soft-bookchapter>.

⁶ MOMoT (Marrying Optimization and Model Transformations).

4.3.3.2 Encoding

In contrast to the application of Rizzi's encoding that represents the elements of the model to optimize, the encoding used in this subsection will represent the operations, i.e., the transformations, that can be executed on the model. Specifically, the chromosome will represent the maximum number of operations that can be executed on the initial model in order to produce the output model, e.g., in our case the monolithic model is transformed into a modularized version. In this context, the user must define an upper bound for the length of the chromosome, in order to initialize the population. Next, we discuss how we can represent variable lengths for the solutions.

MOMoT supports a unit that is called *Placeholder*. It will never actually be executed, hence does not have any effect on the output model. As explained in Sect. 4.3.3.1, this type of unit is used to get shorter sequences that may take less execution time and may be easier to understand. In addition, this special unit may be used to replace another element in the chromosome in the course of repair. This is useful when two operations enter into contradiction after applying genetic operations on the transformation sequences. In such situations, a repair process may replace an element with a placeholder in order to avoid an illegal sequence of transformations. Finally, as the maximum number of operations has to be defined beforehand, with the use of placeholders, the number of operations that are executed on the model may vary, thus, contributing to the diversity of solutions.

In summary, one solution consists of an ordered sequence of applicable transformation rules that creates a valid output model. Figure 4.12 shows an example of such a solution. The input model has all the elements within the same module (left bottom corner), then by applying the chromosome (top of the figure), the output model

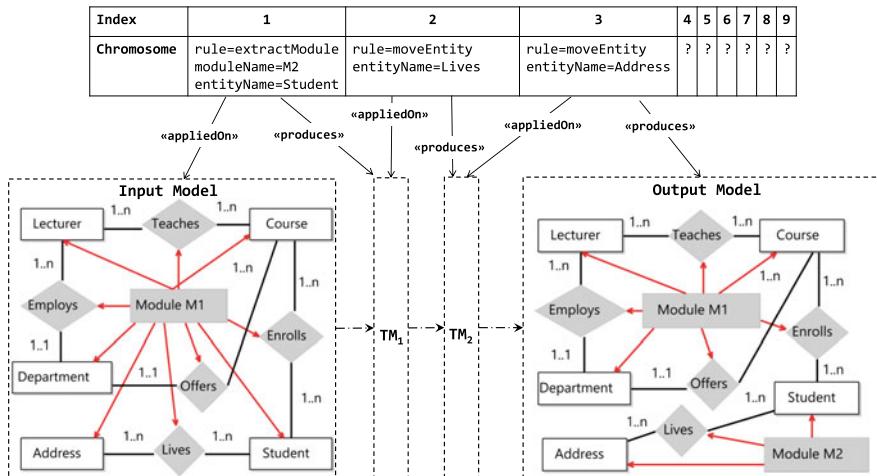


Fig. 4.12 Encoding scheme for transformation rule application sequences using the college database example

can be obtained (right bottom corner). The chromosome is 9 in length comprising 6 placeholder units and 3 rule applications that actually change the input model. Note that the same output model as previously presented in Sect. 4.3.2 is obtained, but now produced with rule applications instead of using genetic operators.

4.3.3.3 Genetic Operators

For the chromosome defined in Sect. 4.3.3.2, we may define several kinds of crossover and mutator operators. In particular, as crossover, we can use the one-point crossover. The offsprings are obtained by taking two chromosomes as input, then a random position is selected as a cut-point, and finally, the genetic information is exchanged. The exchange is done by merging the first part of the first parent with the second part of the second parent to get the first child. A second offspring is built by merging the first part of the second parent with the second part of the first parent. Notice that this is a variation of an MX crossover explained in Sect. 4.3.2.3, but in this case, we define only one cut-point. Of course, the MX crossover can be used on this chromosome as well. This can be accomplished by using the same algorithm explained in Fig. 4.9, but using the operation-based encoding described in Fig. 4.12.

There are also several mutators that we can define for the chromosome defined in Sect. 4.3.3.2. In particular, we can define the Placeholder mutator that selects a random position and replaces it with a placeholder unit. By this, we vary the length of the solution on our way to finding the optimal model. Another mutation is the Executable Rule mutator, which replaces a rule application with an alternative one.

4.3.3.4 Tool Support

The presented approach is implemented in MOMoT [50]. This tool was one of the first search-based MT frameworks based on EMF and combines the Henshin [22] graph transformation framework and the MOEA⁷ framework for performing the meta-heuristic search on graph transformation rule application sequences. All implementation artefacts with installation instructions are provided on GitHub.^{8,9}

⁷ <http://moeaframework.org>.

⁸ <https://github.com/jku-win-se/ai-soft-bookchapter>.

⁹ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

4.3.4 Synopsis

We have presented two distinct approaches for solving model-based engineering problems within GAs, namely the state-based encoding of models and the operation-based encoding of transformation sequences. While the first approach has to be specifically developed for a given problem, the second approach can be applied in a generic way as long as the solutions to the problem can be represented as transformation rule applications. Showing both approaches for solving the modularization problem for ER diagrams should give an orientation toward how other problems, even problems related to other modeling languages, may be encoded in GAs.

The provided solutions combine existing technologies from MDE and AI. The combinations are based on finding appropriate bridges between the technologies to exchange information such as models and their transformations as well as to find appropriate encodings of MDE artefacts in the AI world. While a lot of progress has been made in the last decade, there are still several major challenges remaining as we will discuss in the next section.

4.4 Conclusion and Outlook

In this chapter, we have presented two general strategies for solving MDE problems by the application of GAs which are especially of interest in the analysis and design phases as the early engineering phases are most often supported by models. Model-specific encodings allow to develop AI-driven solutions which deal with a certain type of problem. Transformation-specific encodings allow solving any problem which can be formulated as a transformation, but may require more runtime effort as the encoding and the corresponding genetic operators are not tweaked for a given problem. The Pareto set obtained by either method may be visualized in the graphical editor, in which the developer team can talk about the different alternatives in order to choose a solution and reason about the trade-offs.

The solutions discussed in this chapter have been used for several problems reported previously in the literature. In this chapter, we exemplify both approaches to ER diagrams, because of the widespread application of relational databases in the industry. However, even though these approaches may be used in industry, there are several interesting research lines that have to be tackled in the future.

First, the application of other meta-heuristic search algorithms beyond GAs has to be further explored. Especially, for the transformation-specific encoding, local search techniques [40] may be used to eliminate the challenging crossover operator for recombining transformation sequences. Nevertheless, such algorithms may make assumptions about the search space, e.g., under the notion of the neighborhood, local search algorithms often demand the discovery of solutions obtainable with one step.

Second, the application of learning-based approaches in addition to search-based approaches seems promising and has been already proposed for the transformation-

specific encoding of the MOMoT framework [51]. With reinforcement learning, also an objective-guided search can be performed for the MDE problems as we have presented in this paper. It takes a step-by-step approach to assemble transformations, thereby eliminating the need for expensive repair mechanisms to ensure workable solutions, and mitigating assumptions about the search space like a fixed length of chromosomes. Initial results indicate the feasibility of this idea which may be further researched in the near future on the scalability and stability of such solutions as well as on building hybrid algorithms which combine search and learning approaches.

Third, model encodings and genetic operators have yet to be developed for suitable use in a generic way. The large number of details possibly captured in models works against a one-size-fits-all solution. Depending on the used encoding scheme, the recombination of gene sequences may vastly change model parts and runs in danger of losing information and interfering with domain-related constraints [52]. Consequently, emerging solutions may turn out as malformed and become subject to repair operations, leading to increased runtime and harming overall performance.

Fourth, the collection of objectives for common problems is of interest to further evaluate the use of multi-objective search algorithms. Especially, in the early phases of software development, a lot of trade-offs have to be considered. With a plethora of quality indicators available nowadays, picking an appropriate evaluation method plays a significant role when comparing alternatives [53]. Thus, improving the selection of solutions from large Pareto sets for MDE problems is needed.

Finally, related to the previous point, user interaction is a must. Engineers, especially, in the early phases of the engineering process, will only accept solutions if they are understandable and adaptable [54]. Here, interactive algorithms as well as learning more about the real objectives engineers consider in their engineering tasks have to be further researched in experimental settings. In addition, also MDE tooling has to be improved to visualize alternatives, summarize changes, as well as allow to transmit feedback from engineers again to AI components in a continuous way.

Acknowledgements Work partially funded by the Austrian Science Fund (FWF) under grant number P 30525-N31, the Austrian Federal Ministry for Digital and Economic Affairs, and the National Foundation for Research, Technology, and Development (CDG).

References

1. A. Dennis, B.H. Wixom, R.M. Roth, *Systems Analysis and Design*, (John Wiley & Sons, 2008)
2. M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv. (CSUR)*. **45**(1), 1–61 (2012)
3. P.P.S. Chen, The entity-relationship model-toward a unified view of data. *ACM Trans. Database Syst. (TODS)*. **1**(1), 9–36 (1976)
4. G. Booch, J. Rumbaugh, I. Jacobson, *Unified Modeling Language User Guide.(The 2nd Edition)*. (2005)
5. D. Bork, D. Karagiannis, B. Pittl, A survey of modeling language specification techniques. *Inf. Syst.* **87**, 101425 (2020)

6. A. Wortmann, O. Barais, B. Combemale et al., Modeling languages in Industry 4.0: An extended systematic mapping study. *Softw. Syst. Model.* **19**(1), 67–94 (2020)
7. M. Harman, The role of artificial intelligence in software engineering, in *First International Workshop on Realizing AI Synergies in Software Engineering (RAISE 2012)*, (IEEE, 2012), pp. 1–6
8. I. Boussaïd, P. Siarry, M. Ahmed-Nacer, A survey on search-based modeldriven engineering. *Autom. Softw. Eng.* **24**(2), 233–294 (2017)
9. M. Harman, The current state and future of search based software engineering, in *Future of Software Engineering (FOSE'07)*, (IEEE, 2007), pp. 342–357
10. M. Brambilla, J. Cabot, M. Wimmer, *Model-Driven Software Engineering in Practice*, (Morgan & Claypool Publishers, 2017)
11. D. Schmidt, Guest editor's introduction: model-driven engineering. *Comput.* **39**(2), 25–31 (2006)
12. H. Ishibuchi, Y. Nojima, T. Doi, Comparison between single-objective and multi-objective genetic algorithms: performance comparison and performance measures, in *Proceedings of the IEEE International Conference on Evolutionary Computation*, (IEEE, 2006), pp. 1143–1150
13. A. Jaszkiewicz, On the computational efficiency of multiple objective metaheuristics. The knapsack problem case study. *Eur. J. Oper. Res.* **158**(2), 418–433 (2004)
14. D.L. Parnas, Software structures: a careful look. *IEEE Softw.* **35**(6), 68–71 (2018)
15. D.L. Moody, A. Flitman, A methodology for clustering entity relationship models—a human information processing approach, in *Proceedings of the International Conference on Conceptual Modeling*, (Springer, 1999), pp. 114–130
16. S. Sendall, W. Kozaczynski, Model transformation: the heart and soul of model-driven software development. *IEEE Softw.* **20**(5), 42–45 (2003)
17. K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches. *IBM Syst. J.* **45**(3), 621–646 (2006)
18. T. Mens, P.V. Gorp, A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.* **152**, 125–142 (2006)
19. R. Heckel, G. Taentzer, *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*, (Springer, 2020)
20. S. Kelly, J.P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, (John Wiley & Sons, 2008)
21. D. Steinberg, F. Budinsky, M. Paternostro et al., *EMF: Eclipse Modeling Framework*, 2nd edn. (Addison-Wesley Professional, 2008)
22. T. Arendt, E. Biermann, S. Jurack et al., Henshin: advanced concepts and tools for in-place EMF model transformations, in *International Conference on Model Driven Engineering Languages and Systems*, (Springer, 2010), pp. 121–135
23. N. Kahani, M. Bagherzadeh, J.R. Cordy et al., Survey and classification of model transformation tools. *Softw. Syst. Model.* **18**(4), 2361–2397 (2019)
24. D.L. Moody, Comparative evaluation of large data model representation methods: the analyst's perspective, in *Proceedings of the International Conference on Conceptual Modeling*, (Springer, 2002), pp. 214–231
25. D. Moody, A multi-level architecture for representing enterprise data models, in *Proceedings of the International Conference on Conceptual Modeling*, (Springer, 1997), pp. 184–197
26. D. Bork, A. Garmendia, M. Wimmer, Towards a Multi-Objective Modularization Approach for Entity-Relationship Models. *ER Forum, Demo and Poster 2020*. (CEUR, 2020), pp. 45–58
27. P. Jaeschke, A. Oberweis, W. Stucky, Extending ER model clustering by relationship clustering, in *International Conference on Conceptual Modeling*, (Springer, 1993), pp. 451–462
28. M. Tavana, P. Joglekar, M.A. Redmond, An automated entity-relationship clustering algorithm for conceptual database design. *Inf. Syst.* **32**(5), 773–792 (2007)
29. A. Burdusel, S. Zschaler, S. John, Automatic Generation of Atomic Consistency Preserving Search Operators for Search-Based Model Engineering, in *Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, (IEEE, 2019), pp. 106–116

30. P. Feldman, D. Miller, Entity model clustering: structuring a data model by abstraction. *Comput. J.* **29**(4), 348–360 (1986)
31. G. Simsion, A structured approach to data modelling. *Aust. Comput. J.* **21**(3), 108–117 (1989)
32. K.S. Komar, A. Santra, S. Bhowmick et al., EER->MLN: EER Approach for Modeling, Mapping, and Analyzing Complex Data Using Multilayer Networks (MLNs), in *International Conference on Conceptual Modeling*, (Springer, 2020), pp. 555–572
33. Y. Tzitzikas, J.L. Hainaut, How to tame a very large ER diagram (using link analysis and force-directed drawing algorithms), in *Proceedings of the International Conference on Conceptual Modeling*, (Springer, 2005), pp. 144–159
34. F. Pérez, J. Font, L. Arcega et al., Empowering the human as the fitness function in search-based model-driven engineering. *IEEE Trans. Softw. Eng.* **10**(10), 1–16 (2021)
35. A. Burdusel, S. Zschaler, S. John, Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. *Softw. Syst. Model.* **20**(6), 1857–1887 (2021)
36. W. Mkaouer, M. Kessentini, A. Shaout et al., Many-objective software remodularization using NSGA-III. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*. **24**(3), 1–45 (2015)
37. W. Kessentini, M. Wimmer, H. Sahraoui, Integrating the designer in-the-loop for meta-model/model co-evolution via interactive computational search, in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, (2018), pp. 101–111
38. W. Kessentini, V. Alizadeh, Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search, in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, (2020), pp. 68–78
39. C. Abid, V. Alizadeh, M. Kessentini et al., 30 years of software refactoring research: a systematic literature review (2020), arXiv preprint [arXiv:2007.02194](https://arxiv.org/abs/2007.02194)
40. R. Bill, M. Fleck, J. Troya et al., A local and global tour on MOMoT. *Softw. Syst. Model.* **18**(2), 1017–1046 (2019)
41. A. Burdusel, S. Zschaler, D. Strüber, MDEoptimiser: a search based model engineering tool, in *Companion Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, (ACM, 2018), pp. 12–16
42. H. Abdeen, D. Varró, H.A. Sahraoui et al., Multi-objective optimization in rule-based design space exploration, in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, (ACM, 2014), pp. 289–300
43. M. Kessentini, H. Sahraoui, M. Boukadoum et al., Search-based model transformation by example. *Softw. & Syst. Model.* **11**(2), 209–226 (2012)
44. A. García Domínguez, F. Krikava, and L. M. Rose, eds. Proc. of the 9th Transformation Tool Contest @ STAF. Vol. 1758. CEUR Workshop Proc. 2016
45. M. Fleck, J. Troya, M. Wimmer, Proc. of the 9th Transformation Tool Contest @ STAF. vol. 1758. CEUR Workshop Proc. 2016, pp. 1–8
46. F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms*, (Springer, 2006), pp. 9–32
47. S. Rizzi, Genetic operators for hierarchical graph clustering. *Pattern Recognit. Lett.* **19**(14), 1293–1300 (1998)
48. A.E. Eiben, J.E. Smith et al., *Introduction to Evolutionary Computing*, vol. 53, (Springer, 2003)
49. Goldberg DE, Lingle R, et al. “Alleles, loci, and the traveling salesman problem”. Proc. of the International Conference on Genetic Algorithms and their Applications. 1985, pp. 154–159
50. M. Fleck, J. Troya, and M. Wimmer, “Search-based model transformations”. *J. Softw. Evol. Process.* **28**.12 (2016), pp. 1081–1117
51. M Eisenberg, H Pichler, A Garmendia, et al. Towards Reinforcement Learning for In-Place Model Transformations. 24th International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE, 2021, pp. 82–88
52. S. Zschaler, L. Mandow, Towards Model-Based Optimisation: Using Domain Knowledge Explicitly. *Software Technologies: Applications and Foundations STAF, CollocatedWorkshops*. Vol. 9946. LNCS. Springer **2016**, 317–329 (2016)

53. M. Li, T. Chen, X. Yao, How to evaluate solutions in pareto-based search-based software engineering? a critical review and methodological guidance. *IEEE Trans. Softw. Eng.* (2020)
54. A. Ramirez, J.R. Romero, C.L. Simons, A systematic review of interaction in search-based software engineering. *IEEE Trans. Softw. Eng.* **45**(8), 760–781 (2018)

Part II

Development and Deployment

Chapter 5

Statistical Models and Machine Learning to Advance Code Completion: Are We There Yet?



Tien N. Nguyen

Abstract Code completion, commonly used by software developers, is the third most important feature in an IDE as code completion helps speed up the process of coding by filling in the desired code and reducing common mistakes. The early, traditional code completion approaches rely on program analysis to produce a long, alphabetically sorted list of potential suggested code elements. More advanced code completion approaches have leveraged statistical models as well as machine learning (including deep learning) in suggesting the code to be completed. This article will systematically review the advances of various code completion approaches and point out the limitations and future direction to further improve the state-of-the-art techniques in this area.

5.1 Introduction

The advances in Artificial Intelligence (AI) and Machine Learning (ML) have opened new directions in many aspects of our daily life. It has brought several intelligent techniques to improve the software development process including code-related tasks such as intelligent code assistants, program synthesis, automated software testing, and automated program repair. The AI/ML models have been leveraged to improve the programmers' productivity and effectiveness in the programming tasks within an integrated development environment (IDE). Within an IDE, a code completion (CC) tool helps a developer to fill in the code under editing. The granularity of the code to be completed could be at the token, statement, or code fragment level. Code completion, commonly used by software developers, is *the third most important feature in an IDE* (besides editing and compiling) [27]. CC helps speed up the process of coding by filling in the desired code and reducing common mistakes. Programmers use CC as often as using the copy and paste feature. In all major IDEs (e.g., Eclipse and IntelliJ IDEA), the CC feature has evolved from basic support to more advanced services.

T. N. Nguyen (✉)

University of Texas at Dallas, Dallas, USA

e-mail: tien.n.nguyen@utdallas.edu

Basic code completion helps a developer complete the names of classes, methods, fields, and keywords within the visibility scope. When code completion is invoked, the tool analyzes the context and suggests the choices that are reachable from the current position. If CC is applied to a part of a field, parameter, or variable declaration, the tool suggests a list of possible names depending on the entity's type. For example, if a variable `v` of `String` is typed, an IDE will recommend the list of possible fields and methods of the type `String`. In general, the list of those accessible elements is sorted in alphabetic order. Eclipse [11] also supports template-based completion for common constructs/APIs (`for/while`, `Iterator`).

More advanced code completion services, such as IntelliJ, include the completion of common program constructs: (1) the right part of assignment statements, (2) variable initializers, (3) return statements, (4) the list of arguments of a method call, (5) after the new keyword in an object declaration, (6) in chained expressions, (7) a method declaration, (8) a code construct, e.g., punctuation in a condition statement, and (9) wrapping a method call of an argument.

Those CC services are built mainly using *program analysis* (PA) on the currently edited code in the IDE. The key issues with the PA approaches that have been hindering the effectiveness of CC have two folds. The first issue with PA-based CC approaches is that there are a large amount of possibilities with equal likelihoods of code candidates that can be filled in at the cursor. Although the number of valid candidates for the next token is limited, the number of possible valid (complete) statements at the suggestion point might be combinatorially explosive or even infinite. For example, after `len =`, the valid next-token candidates include a limited set of code tokens including the appropriate prefix operators (e.g., `++` and `--`), the open parenthesis, field access, method call, and local variable (e.g., `children` and `f`). However, there is potentially an infinite number of valid statements at that suggestion location. The second issue is the long and unranked candidate list of candidates due to the inability of PA to evaluate their occurrence likelihoods. In brief, program analysis could produce a large number of candidates with equal occurrence likelihoods, despite that the candidates are syntactically or semantically valid.

To address those issues, several code completion approaches have leveraged *software mining* and *information retrieval* techniques to improve code completion [5, 31, 35]. The idea is that the CC tool could suggest and rank higher the more frequent code, called *a code pattern*. When the retrieved code has occurred frequently, it can be viewed as a code pattern. Such a model searches for the same or similar code that has been previously seen in a corpus. Such a pattern or a retrieved code can be used as the candidate for completion. However, the tokens that need to be filled for the current code might not be as frequent, leading to the ineffectiveness of those approaches. Moreover, while as single tokens, code is repetitive, as entire statements, they are quite specific for projects [40]. Thus, searching/mining for a similar code to be filled is not as effective across projects as within a project.

Recent advances in *deep learning* (DL) and *natural language processing* (NLP) have enabled several approaches for code completion using *statistical language models* (LM), which captures the regularity of source code [16]. Such NLP-based approaches leverage the fact that code is highly repetitive and predictable [16]. The

suggested tokens are based on the frequent sequences of tokens and the partial code. The intelligent code assistant tools [6, 7, 9] have leveraged the AI/ML models to learn from billions of lines of public code to recommend for developers in their programming tasks. Despite its popularity, recent studies [13, 24] have shown that the state-of-the-art CC tools are still far from achieving a satisfactory level.

In this article, we will systematically review the advances in various code completion approaches and point out the limitations and future direction to further improve the state-of-the-art techniques in this area.

5.2 Code Completion with Software Mining

In this section, we present the code completion techniques that are based on source code mining. The key idea driving the mining-based code completion approaches is that a tool can suggest the code belonging to *code patterns*. Several approaches use different representations of code patterns and depend on them for code completion.

5.2.1 Frequent Pairs or Sets as Code Patterns

Some code completion approaches detect the code patterns for code completion via mining frequent pairs or sets of method calls.

Bruch et al. [5] propose three code completion algorithms to suggest the method call for a single variable v under editing based on code examples in a database. The first one, FreqCCS, suggests the method that is most frequently used in the database. The second one, ArCCS, mines the associate rules $A \rightarrow B$ in which if method A is used, method B is often called, thus B will be suggested. The third algorithm, best-matching neighbors (BMN), adapts the k-nearest neighbor algorithm to recommend for a variable v . BMN encodes the current context and the examples in the database as binary feature-occurrence vectors [5]. The features for a context are the *un-ordered set* of method calls of v in the currently edited code and the names of the methods that use v . Each example in the database is also modeled as a binary feature-occurrence vector. To recommend the method call for v , BMN then uses the query vector to search against the vectors of the examples. The set of vectors of examples with the same smallest Hamming distance to the query vector is called the BMN set. Then, BMN ranks the methods based on their frequencies in the examples in the BMN set.

Several approaches have been proposed to mine the frequent pairs of method calls (pairs of calls [12, 22, 39] or partial orders of calls [2]). One can leverage those approaches to provide code suggestions and completion for a variable.

5.2.2 Graphs of Code Elements as Code Patterns

Some mining-based code completion approaches rely on graphs to represent code patterns. In those approaches, a code pattern is represented as a graph-based representation that captures the usages of multiple variables in different types, method calls in multiple libraries, control structures, and their data/control dependencies.

5.2.2.1 Graph-Based Code Pattern Representation

Figure 5.1 shows a portion of code using SWT, a graphical user interface (GUI) library, to create a window containing the “OK” button. According to SWT documentation, creating a window involves two types of objects. A `Display` object is responsible for managing the GUI events and related resources between SWT and the operating system. A `Shell` object represents a GUI window, which is associated with the `Display` object and acts as a container for other GUI elements such as buttons and text boxes. After a `Display` object (`display`) is created, a `Shell` object (`shell`) is created to form a top-level window (lines 1–2). It then starts to receive and process the events via `display` until object `shell` is disposed (lines 9–14). Finally, object `display` is disposed.

According to SWT documentation, this usage of those SWT elements (the classes `Display`, `Shell`, and their methods) is the correct way to perform the task of creating a top-level GUI window in SWT. Thus, this API usage (lines 1–2 and 9–15) occurs very frequently in Java code that uses the SWT library. This correct usage is referred to as *API usage pattern* (or a pattern for short).

GraPacc is a graph-based, pattern-oriented, context-sensitive code completion approach that is based on a database of graph-based patterns. GraPacc is based on Groum [33], a graph-based model, in which the nodes represent actions (i.e., method calls), data (i.e., objects/variables), and control points (i.e., branching points of control structures such as `if`, `while`, and `for`). The edges represent the control

```

1 Display display = new Display();
2 Shell shell = new Shell(display);
3 ...
4 Button button = new Button(shell, SWT.PUSH);
5 button.setText("OK");
6 button.setSize(new Point(40,20));
7 button.setLocation(new Point(200,20));
8 ...
9 shell.pack();
10 shell.open();
11 while (!shell.isDisposed()) {
12     if (!display.readAndDispatch())
13         display.sleep();
14 }
15 display.dispose();

```

Fig. 5.1 SWT usage example

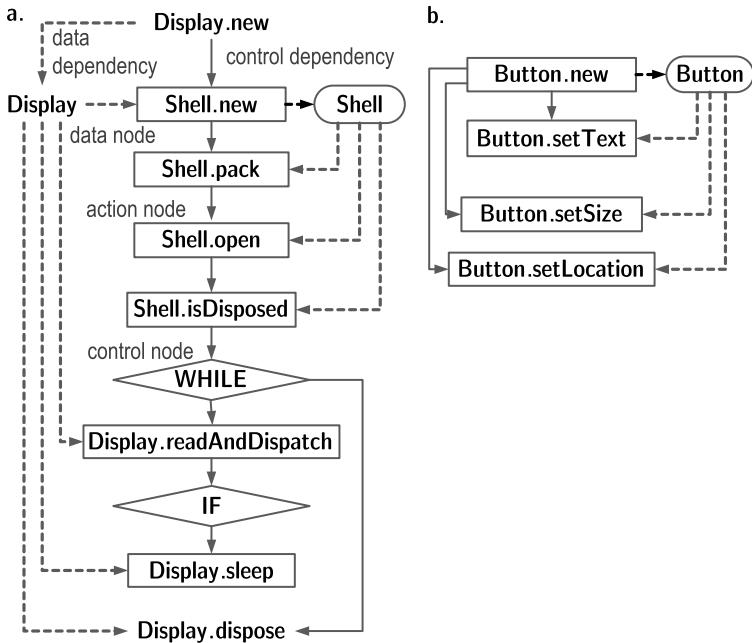


Fig. 5.2 SWT usage patterns

and data dependencies between the nodes. Labels of the nodes are built from the corresponding names of classes/methods/control structures.

Figure 5.2 illustrates the usage patterns shown in Fig. 5.1 as Groum models. For clarity purposes, the nodes are labeled with the simple names of classes and methods. In the actual implementation, nodes' labels have their fully qualified names. For example, the label of node `Display.close` is actually `org.eclipse.swt.widgets.Display.close`.

As seen, two data nodes labeled `Display` and `Shell` represent two variables `display` and `shell`. Action nodes such as `Display.new`, `Display.readAndDispatch`, and `Shell.open` represent the method calls. An edge from data node `Display` to action node `Shell.new` represents the data dependency, while an edge from `Display.new` to `Shell.new` represents their control flow order. Two control nodes `WHILE` and `IF` model a `while` loop and an `if` statement.

5.2.2.2 Code Completion with Graph-Based Patterns

A query is generally incomplete (in terms of the task that is intended to achieve) and might not be parsable. Figure 5.3 illustrates a code fragment as a query. The character `_` denotes the editing cursor where a developer invokes the code completion tool during programming.

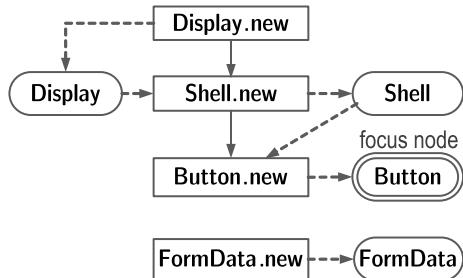
```

1 Display display = new Display();
2 Shell shell = new Shell(display);
3 ...
4 Button button = new Button(shell, SWT.PUSH);
5 FormData formData = new FormData();
6 button...

```

Fig. 5.3 SWT example for current code

Fig. 5.4 Graph-based usage model of Query in Fig. 5.3



GraPacc constructs the corresponding Groum from the AST provided by a partial program analysis tool [10] in the previous step using the constructing algorithm from our prior work [33]. Due to the incompleteness of the query code, the unresolved nodes in the AST are discarded. They are considered as tokens and used to extract token-based features. The data nodes corresponding to the variables of the data types that are not resolved to fully qualified names are kept with only simple names. Figure 5.4 shows the Groum built for the query example in Fig. 5.3. As seen, the objects `shell`, `button`, `bData`, and `display` are resolved to the data nodes labeled with their types `Shell`, `Button`, `FormData`, and `Display`, respectively. Node `Button` is denoted as the *focus node*, because the token closest to the editing cursor is `button`.

GraPacc extracts the context-sensitive features from the code under editing, e.g., the API elements being on focus or under modification and their relations to other code elements. The features are then used to search and rank the patterns that are best matched with the current code. When a pattern is chosen by a user, GraPacc will fill in the code based on that pattern with proper replacements of program elements in the current context of the program.

5.2.3 Leveraging Editing History for Code Completion

While the code patterns are based on the frequently repeated code, other code completion approaches leverage the patterns of code changes.

Hill and Rideout [15]'s code completion approach relies on editing histories. It matches the fragment under editing with small similar-structure code clones, and then performs transformations for code completion. Robbes and Lanza [35] introduce an

approach to improve code completion with change history during an editing session. A session is a sequence of dated changes separated by at most an hour [35]. Given a variable v with/without type T , to recommend the relevant methods, they propose 6 strategies to improve code completion using recent histories of modified/inserted code during an editing session: (1) search through all methods defined in the system that match the prefix of v , (2) search through all methods in the class hierarchy at T , (3) search first in the methods of the current class T , then in its package, and finally in the entire system, (4) order the methods based on their recently modified dates for their names and/or bodies, (5) order them based on their most recently inserted code, and (6) all methods that are changed during a session.

5.2.4 API Code Recommendation Using Statistical Learning from Fine-Grained Changes

APIRec [28] is an API recommendation approach that taps into the predictive power of repetitive code changes to provide relevant API recommendations for developers. APIRec is based on statistical learning from fine-grained code changes and from the context in which those changes were made. The intuition is that when developers make low-level changes, even non-contiguous changes are connected. These connections exist because the developer made the changes with a higher level intent in mind (e.g., adding a loop collector). Grouping fine-grained changes by higher level intent allows us to cut through the noise of unrelated tokens that may surround the recommendation point. To find these groups of fine-grained changes, statistical learning is used in APIRec on a large code change corpus. The changes that belong to higher level intents will co-occur more frequently than non-related changes.

APIRec computes the most likely API call to be inserted at the requested location where an API call would be valid. It works in three steps: (i) it builds a corpus of fine-grained code changes from a training set, (ii) it statistically learns which fine-grained changes co-occur, and (iii) it computes and then recommends a new API call at a given location based on the current context and previous changes.

5.3 Code Completion with Statistical Language Models

The key limitation of the code completion approaches using software mining techniques is that the code patterns must be explicitly represented and learned. In some cases, it is natural to ask the question of how likely a specific program element appears at a certain position. Thus, several researchers have applied machine learning techniques used to predict missing words in a sentence to code completion.

5.3.1 *N-Gram Language Model*

Statistical language models are used to capture the regularities/patterns in natural languages by assigning occurrence probabilities to linguistic units such as words, phrases, sentences, and documents [23]. Since a linguistic unit is represented as a sequence of one or more basic symbols, language modeling is performed by computing the probability of such sequences. To do that, a modeling approach assumes that a sequence is generated by an imaginary (often stochastic) process of the corresponding language model. With such a generative process, one could use a language model to predict the next symbol in a sentence.

Figure 5.5 illustrates a prediction process of the n -gram model. The larger a word, the high occurrence probability it has. For example, the phrase “*I have*” occurs more frequently than “*I am*” or “*I was*”. When a person types “*I do ...*”, the word that most likely follows that phrase is the word “*not*”.

Definition 1 (*Language Mode*) A language model L is a statistical, generative model defined via three components: a vocabulary V of basic units, a generative process G , and a likelihood function $P(\cdot|L)$. $P(s|L)$ is the probability that a sequence s of the elements in V is “generated” by the language model L following the process G .

When the context of discussion is clear regarding the language model L , we use $P(s)$ to denote $P(s|L)$ and call it *the generating probability of sequence s*. Thus, a language model could be simply considered to have a probability distribution of every possible sequence. It could be estimated (i.e., trained) from a given collection of sequences (called a *training corpus*).

Fig. 5.5 An illustration of N-gram language model [1]



5.3.2 Lexical Code Tokens and Sequences

Statistical language models have been applied to software engineering, such as in code suggestion/completion [17]. To apply such a model to source code, one first needs to define the vocabulary, i.e., the collection of basic units (also called *terms* or *words*) that are used to make up a sequence. A vocabulary can be constructed via performing lexical analysis on source code (as a sequence of characters), i.e., breaking it into *code tokens* based on the specification of the programming language. The lexemes (lexical values) of the tokens are then collected as the basic units in the vocabulary. The input source code is represented as a sequence of lexical code tokens, which is called *lexical code sequence*. Formally:

Definition 2 (Lexical Code Token) A *lexical code token* is a unit in the textual representation of source code and associated with a *lexical token type* including identifier, keyword, or symbol, specified by the programming language.

Definition 3 (Lexeme) The *lexeme* of a token is a sequence of characters representing its lexical value.

Definition 4 (Lexical Code Sequence) A *lexical code sequence* is a sequence of consecutive code tokens representing a portion of source code.

For example, after lexical analysis, the piece of code “`len = str.length();`” is represented by a lexical code sequence of eight tokens, with their token types and lexemes shown in Table 5.1. `len`, `str`, and `length` are three `Identifier` tokens, while the other tokens have different types of symbols. In lexical analysis, no semantic information (e.g., data type) is available. For example, `str` is not recognized as a `String` variable, and `length` is not recognized as the name of a method in the `String` class.

Table 5.1 Lexical code tokens from “`len = str.length();`”

Lexeme	Token type
<code>len</code>	Identifier
<code>=</code>	Equal (symbol)
<code>str</code>	Identifier
<code>.</code>	Period (symbol)
<code>length</code>	Identifier
<code>(</code>	Left parenthesis (symbol)
<code>)</code>	Right parenthesis (symbol)
<code>;</code>	Semicolon (symbol)

5.3.3 Lexical N-Gram Model for Source Code

A n -gram model is a language model with two assumptions. First, it assumes that a sequence could be generated from left to right. Second, the generating probability of a word in that sequence is dependent only on its *local context*, i.e., a *window of previously generated words* (a special case of Markov assumption). Such dependencies are modeled based on the occurrences of word sequences with limited lengths. A sequence of n words is called a n -gram. When n is fixed at 1, 2, or 3, the model is called a unigram, bigram, or trigram.

Definition 5 (*Lexical N -gram*) The lexeme of a sequence of n consecutive code tokens is called a lexical n -gram. It is defined as the sequence of the lexemes of those tokens.

Those assumptions are reasonable for source code. That is, the next code token could be predictable/dependent on the previously written code tokens [17]. For example, in a source file, the code sequence “`for (int i = 0; i < n;` ” is considered as the local context of the next token. This piece of code could be recognized as a `for` loop with i as the iterative variable, and thus, in many cases, the next code token is `i`.

With the assumption of generating tokens from left to right, the generating probability of code sequence $s = s_1s_2\dots s_m$ is computed as

$$P(s) = P(s_1).P(s_2|s_1).P(s_3|s_1s_2)\dots P(s_m|s_1\dots s_{m-1})$$

That is, the generating probability of a code sequence is computed via that of each of its tokens. Thus, a language model needs to compute all possible conditional probabilities $P(c|p)$ where c is a code token and p is a code sequence. With the Markov assumption, the conditional probability $P(c|p)$ is computed as $P(c|p) = P(c|l)$ in which l is a sub-sequence made of the last $n - 1$ code tokens of p . With this approximation, a model only needs to compute and store the conditional probabilities involving at most n consecutive code tokens. $P(c|l)$ is often estimated as

$$P(c|l) \simeq \frac{\text{count}(\text{lex}(l, c)) + \alpha}{\text{count}(\text{lex}(l)) + |V|\alpha}$$

`lex` is the function that builds the lexeme of a code sequence. For instance, the code sequence `i < n` might occur in several places, e.g., in a `for` or `if` statement. The same lexeme sequence (`i < n`) would be created for them, and they are all counted as the occurrences of the same code sequence. α is a smoothing value for the cases of small counting values. $|V|$ is the size of the vocabulary.

5.3.4 Semantic n-Gram Language Model for Source Code

5.3.4.1 Motivation

The lexical n -gram model has been shown to capture well code regularities/patterns at the lexical level to support code completion/suggestion [17]. Code patterns at higher levels of abstraction could be also useful for that task. However, such patterns with well-defined semantics could not be captured well with the lexical n -gram model.

Let us consider the following statement `"len = str.length();"`. The lexical n -gram model represents it as a lexical sequence (Table 5.1). However, an editor with semantic analysis capability will recognize it as an *assignment statement*, with the left-hand side being a *variable*, and the right-hand side being an *expression*, which in turn contains a *method call* to a method named `length`. If the code under editing is sufficiently complete, further semantic analysis such as typing and scoping will help identify the data types, e.g., `len` being of the type `int` and `str` being of the type `String`. Semantic analysis will also help verify the *applicability* of the method call to `length` on the variable `str`, and *type compatibility* of the assignment of the returned value from `length` to the variable `len`.

A language model could benefit from such semantic information to detect the pattern “*getting the length of a String object and assigning it to an int variable*”. Without semantic information, it is challenging for a language model to detect that pattern if the variable names are different in different places. Moreover, the pattern could then help in code suggestions. For example, assume that the statement was incomplete as `"len = str."` and code completion is requested. If the above semantic information is available, a model could determine that a method of a `String` object is sought and the returned value will be assigned to an `int` variable. Thus, the method `length` would be a candidate for the next suggested token. In brief, using semantic information, a language model would capture better the code patterns at higher abstraction levels, thus, producing better code suggestions.

5.3.4.2 SLAMC Model

SLAMC, a statistical Semantic Language Model, is designed for source Code. It encodes semantic information of code tokens into basic semantic units, and captures their regularities/patterns. It also combines local context with global concern information as well as the pair-wise association of tokens in the modeling process.

Overview and Design Strategies

Let us first explain our design strategies in selecting the kinds of semantic information to be incorporated into our model. The first semantic information is the **role** of a token in a program with respect to the written programming language, i.e., whether

it represents a variable, data type, operator, function call, field, and keyword. With that, SLAMC will be able to learn the syntactical regularities/patterns such as “*after a variable, there is often an assignment operator*”. Second, it is useful to include the **data types** of the tokens, especially the types of variables, fields, and literals. Data types would help us capture both syntactical patterns and the patterns at higher abstraction levels, e.g., “*the parameter of function System.out.println is often a string or Integer literal*”. For a method, the **return type**, the declared **parameter list**, and the **number of passing parameters** are important to identify and characterize the method. Thus, for a method, the following information is incorporated: its signature, including its name, class name, return type, and parameter list.

Those kinds of information are encoded as the *semantic values* of the code tokens, which are called **sememes** (will be formally defined later). The sememes are included in the vocabulary of our model and used to construct the n -gram sequences and associated pairs in the modeling process. To extract meaningful sequences and pairs, SLAMC uses the **scopes** and **dependencies** of code tokens. That is, it considers only the associated pairs of the code tokens that have dependencies and are in the proper scopes. For example, only the pairs of function calls having data dependencies are considered. Moreover, it considers only the sequences that are in appropriately structural scopes. For example, the sequences spanning across block, function, or class boundaries are excluded. Let us formally describe the concepts.

Semantic Code Tokens and Sequences

Definition 6 (*Semantic Code Token*) A semantic code token is a lexical code token with associated semantic information including its ID, role, data type, sememe, scope, and structural and data dependencies.

Definition 7 (*Role*) The *role* of a semantic code token refers to the role of the token in a program with respect to a programming language. The typical token roles include type, variable, literal, operator, keyword, function call, function declaration, field, and class.

In “`str.length()`”, after semantic analysis, `str` is recognized as a semantic code token with its role of a variable, while the role of `length` is a function call.

Definition 8 (*Sememe*) The *sememe* of a semantic code token is a structured annotation representing its semantic value, including its token role and data type.

Definition 9 (*Vocabulary*) A *vocabulary* is a collection of distinct sememes of all semantic code tokens.

Table 5.2 lists the construction rules to build the sememes for the popular types of semantic code tokens. For example, `length` in `str.length()` has the semantic role of a function call, its sememe consists of the annotation “CALL”, “[”, its class name `String`, its name `length`, the number of passing parameters(0), the returned type

Table 5.2 Construction rules for sememes of semantic code tokens

Token Role	Construction Rule	Example
Data type T	TYPE[T]	String → TYPE[String]
Variable x	VAR[typeof(x)]	str (String) → VAR[String]
Literal v	LIT[typeof(v)]	"Java" → LIT[String]
Function decl m	FUNC[type(m), lexeme(m), paralist(m), rettype(m)]	indexOf → FUNC[String, indexOf, PARA[String], Integer]
Function call m	CALL[type(m), lexeme(m), paracount(m), rettype(m)]	length → CALL[String, length, 0, Integer]
Parameter x	PARA[typeof(x)]	str (String) → PARA[String]
Field f	FIELD[type(f), lexeme(f)]	left → FIELD[Node, left]
Operator o	OP[name(o)]	= → OP[assign], . → OP[access]
Cast (T)	CAST[T]	(Integer) → CAST[Integer]
Keyword	To corresponding reserved token	if → IF, class → CLASS
Block open & close	To corresponding reserved token	{ (of a for loop) → FOREND
Special literal	To corresponding reserved token	"" → EMPTY, null → NULL
Unknown	To special lexical token LEX	abc → LEX[abc]

Integer, and "]" as shown in the fifth row. That sememe represents the semantic value of that semantic code token, i.e., a method call to `length`. The separator tokens, e.g., semicolons and parentheses, are not associated with semantic information, and thus are excluded.

Semantic information might be unavailable, e.g., when the current code is incomplete, leading to no typing information or un-deciding whether an identifier is a variable, data type, or a method name. In such cases, the lexical token is kept and annotated with the sememe of type `LEX` (the last row).

For a variable, its sememe does not include its name, e.g., the variable `str` is encoded as `VAR[String]` to denote it as a String variable. This allows us to capture more general code patterns involving variables because variables' names are often individuals' choices, and the naming convention might be even different across projects. For example, two statements "`len = str.length()`" and "`l = s.length()`" express the same code pattern when `l` and `len` are of type `int`, and `s` and `str` are of type `String`, although the variables' names are different (e.g., `len` versus `l`, and `str` versus `s`). To capture that pattern and improve predictability, SLAMC represents those statements by two code sequences with the same sememe sequence:

```
VAR[Integer] OP[assign] VAR[String] OP[access] CALL[String, length, 0, Integer]
```

Table 5.3 Associated information of semantic code tokens for `len=str.length()`

ID	Role	Sememe	Lexeme	Scope	Depend
T1	Variable	VAR[int]	len	C1.M2.B3	[T3,T5]
T2	Operator	OP[assign]	=	C1.M2.B3	NA
T3	Variable	VAR[String]	str	C1.M2.B3	[T1,T5]
T4	Operator	OP[access]	.	C1.M2.B3	NA
T5	Method	CALL[String,...]	length	C1.M2.B3	[T1,T3]

Similarly, the concrete literals' values could vary in concrete usages. For example, the pattern of printing a string could be instantiated with different string literals in different usages (e.g., `System.out.println("Hello World!")`, or `System.out.println("File not found!")`). To capture code patterns with higher abstraction levels and enhance predictability for code suggestion, SLAMC annotates the sememe of a literal with its data type rather than its lexeme. Thus, those two printing statements will have the same sememe sequence.

In other cases, programming patterns could involve special literal values. For example, many functions use a 0 (`zero`) as the returned value indicating successful execution. Objects are frequently checked for `nullity` before being processed, for example “`if (node != null)`”. Thus, SLAMC has also special sememes representing such values (including `null`, `zero`, and `empty string`). For instance, the expression “`if (node != null)`” is captured as the sequence `IF VAR[Node] OP[neq] NULL`.

Definition 10 (Scope) A *scope* associated with a semantic code token identifies the block containing that token.

For a program, a scope is modeled by a sequence of blocks' identifiers in its abstract syntax tree (AST). For example, the scope C1.M2.B3 identifies the third block in the second method of the first class in the current source file.

Definition 11 (Dependency) The *dependency* set of a semantic code token t is a set of IDs of the other code tokens that have structural or data dependencies with t .

Structural dependencies are defined as child-parent relations in an AST. Data dependencies are defined among program elements and currently computed via data analysis on variables. Table 5.3 illustrates the semantic code tokens for the example `"len = str.length();"`. The variable `len` (with the ID of T1) depends on the `String` variable `str` (T3), and the method call `length` (T5), thus its dependency set is `[T3, T5]`. The lexemes of semantic code tokens are used in code suggestion. The patterns with sememes only suggest the token role and data type of the next token (e.g., a variable of type `String`). SLAMC needs to find the most suitable semantic token and use the lexeme to fill in the code.

Definition 12 (Semantic Code Sequence) A *semantic code sequence* is a sequence of semantic code tokens.

Definition 13 (*Semantic n -gram*) The sememe of a semantic code sequence of size n , called a *semantic n-gram*, is the sequence of the sememes of the corresponding tokens.

For example, SLAMC represents the piece of code “`if (node != null)`” as the semantic code sequence of four semantic tokens (keyword `if`, variable `node`, operator `!=`, and special literal `null`). The sememe of this sequence, computed from those of its tokens, is a semantic 4-gram `IF VAR[Node] OP[neg] NULL`. For brevity, we will use the terms *code token*, *code sequence*, and *n-gram* to refer to the semantic counterparts.

5.3.5 Code Suggestion with Semantic N-Gram Language Model

5.3.5.1 Semantics and Context-Sensitive Suggestion

Overview. Instead of suggesting code in a template, SLAMC suggests a sequence of tokens that is *best-fit to the context* of the current code and *most likely to appear next*. It provides a *ranked list of such sequences*. SLAMC is semantic-based, thus it defines a set of suggestion rules that are based on the current context and aim to complete a meaningful code sequence (Table 5.4). The idea is that such a suggested sequence would *complete the code at the current position to form a meaningful code unit and likely appear next*. Currently, the rules are implemented to define a meaningful code unit in terms of a member access, a method call, an infix expression, or a condition expression. For example, if the code context is recognized as an incomplete binary expression such as in “`x +`”, the suggestions will be an expression for the remaining operand with a data type compatible with `x` in the addition. If the context is an incomplete method call, a suggestion will be an expression with a compatible type

Table 5.4 Rules of context-sensitive suggestion

Context	Example	Suggestion
Member access	<code>node.</code>	a token for method or field name, e.g., <code>size</code> or <code>value</code>
Method call	<code>map.get(</code> or <code>Math.max(x,</code>	a type-compatible expression for the next argument, e.g., <code>y</code>
Infix expression	<code>x +</code>	a type-compatible expression for the other operand, e.g., <code>y</code>
Condition expression	<code>if (</code> or <code>while (</code>	a Boolean expression e.g., <code>x != y</code> or <code>!set.isEmpty()</code>
Other	<code>for (int i = 0;</code>	a next token, e.g., <code>i</code>

Table 5.5 Semantic and context-sensitive completion

	Current code	Suggestions	
Lexical tokens	(1) if (node	!= null == null .isRoot()	(4)
Semantic tokens	(2) IF VAR[Node]	OP[neq] NULL OP[equal] NULL OP[access] CALL[Node, isRoot,0,Boolean]	(3)

for the next argument. If it does not match with any pre-defined context, the token with the highest probability is suggested.

To illustrate SLAMC code suggestion algorithm, let us consider an example (Table 5.5). Assume that a developer is writing a statement “if (node” and requests a suggestion (see (1) in Table 5.5). SLAMC engine first converts the code into a semantic code sequence p (see (2)). Analyzing this sequence, the engine recognizes that it matches the rule for an incomplete condition statement. Then, it searches for potential code sequences q that connect with the current code to form a Boolean expression. Such sequences are ranked based on the score $P(q|p)$. Assume that the search returns a ranked list of three semantic code sequences as in (3). Those sequences are transformed back to lexical forms and presented to the user as in (4).

Detailed SLAMC Algorithm. SLAMC code suggestion algorithm has three main steps (Fig. 5.6). In the first step (lines 2–3), it analyzes the code in the current method and produces its semantic code sequence. Since the current code might not be complete or syntactically correct, it uses Partial Program Analysis (PPA) [10] for code analysis and then recognizes the matched code context. PPA parses the code into an AST, which is then analyzed by SLAMC to produce the semantic code tokens and other associated semantic information. If PPA cannot parse some tokens, it marks them as Unknown nodes and SLAMC creates the semantic tokens of type LEX for them. It also estimates the topics using the n -gram topic model (line 3). In step 2 (lines 4–5, 11–22), it predicts the next code sequences that connect with the current code to form a type-compatible code unit as described in the rule of the matched context. All such sequences are ranked based on their scores using a search-based method. In step 3 (lines 6–9), those sequences are transformed to lexical forms and presented to users for selection and filling up. Let us detail steps 2–3.

5.3.5.2 Predicting Relevant Code

Let us use s to denote the semantic code sequence for the entire source file under editing, and θ for its estimated proportion of the topics in the source file. Since the current editing position $edpos$ might not be at the end of s , the engine starts the

```

1 function Recommend(CurrentCode F, NGramTopicModel  $\phi$ )
2    $s = \text{BuildSequence}(F)$  //sequence of semantic code tokens
3    $\theta = \text{EstimateNGramTopic}(s, \phi)$  //topic proportion of  $F$ 
4    $p = \text{GetCodePriorEditingPoint}(s, \text{edpos})$ 
5    $L = \text{Search}(p, \theta)$ 
6   foreach  $q \in L$ 
7      $\text{lex}[q] = \text{Unparse}(q)$ 
8      $u = \text{UserSelect}(\text{lex})$ 
9      $\text{Fillin}(u)$ 
10
11 function Search( $p, \theta$ )
12    $L = \text{new sorted list of size topSize}$ ,  $Q = \text{new queue}$ 
13    $Q.\text{add}("", 1)$  //empty sequence, score = 1
14   repeat
15      $q = Q.\text{next}()$ 
16     if  $\text{length}(q) \geq \text{maxDepth}$  then continue
17      $C(q) = \text{ExpandableTokens}(p, q)$ 
18     for each  $c \in C(q)$   $Q.\text{add}(qc)$ 
19     if  $\text{ContextFit}(p, q)$  then  $L.\text{add}(q, \text{Score}(q, p, \theta, \phi))$ 
20   until  $Q$  is empty
21   if  $L$  is empty then add the top relevant tokens to  $L$ 
22   return  $L$ 

```

Fig. 5.6 SLAMC code suggestion algorithm

search from a sub-sequence p of s , containing all tokens prior to `edpos`. It looks for sequence(s) $q = c_1c_2\dots c_t$. The relevant score of q is

$$P(q|p) = P(c_1|p, \theta).P(c_2|pc_1, \theta)\dots P(c_t|pc_1c_2\dots c_{t-1}, \theta) \quad (*)$$

This suggests that one could expand the sequences token-by-token and compute the score of a newly explored sequence from the previously explored ones. Thus, the SLAMC engine generates relevant next sequences by searching on a lattice of tokens of which each path is a potential suggestion using a depth-limited strategy. That is, it keeps a queue Q of exploring paths and chooses to expand a path q if it has not reached the maximum depth (`maxDepth`), which is a pre-defined maximum length for q (lines 15-18). If q satisfies a context rule, its score will be computed and it will be added to the ranked list L of suggested sequences (line 19). If no sequences satisfy the context, the top relevant tokens are added (line 21).

Expanding Relevant Tokens

Theoretically, at each search step, every token should be considered. However, to reduce the search space, SLAMC chooses only the tokens “expandable” for the current search path q (function `ExpandableTokens` at line 17). To do that, SLAMC uses the trained n -gram topic model ϕ to infer the possible sememes $V(q)$ for the next token of q , and then chooses semantic tokens matching those sememes. Assume that the current search path is $q = c_1c_2\dots c_i$. To find the set of possible sememes $V(q)$ of the next token c , SLAMC connects p and q and extracts any possible n -grams l ending at c_i (l might have tokens in both p and q). Then, SLAMC looks for l on the

prefix tree of n -grams (see Sect. 3.3.1). If l exists, all sememes of its child nodes are added to $V(q)$.

For each sememe $v \in V(q)$, SLAMC creates a corresponding semantic code token and puts it into the set of expandable tokens $C(q)$. SLAMC uses the rules in Table 5.2 to infer necessary information, e.g., role or lexeme. For instance, if the sememe is `CALL[Node, isRoot, 0, Boolean]`, the semantic code token has the role of *function call* and the lexeme of `isRoot`. It has the same scope as the previous token c_i in q and no dependency.

The sememes of variables and literals in the semantic language model do not have lexemes. Thus, SLAMC infers the lexemes for sememes of variables using a **caching** technique. If v is a sememe for a variable, SLAMC selects all existing semantic code tokens in the sequence s that represent variables. Then, all tokens for variables that belong to the same or contain the scope of the last code token c_i of the search path and have the same type as specified in the sememe v will be added to $C(q)$. For example, if c_i has the scope `C1.M2.B3` and v is a `VAR[String]`, all `String` variables in the scopes `C1.M2.B3`, `C1.M2`, and `C1` are considered. For a literal sememe, SLAMC creates a semantic token with the default value for its type (e.g., `0, null`).

Checking of Context

SLAMC code completion engine uses the rules in Table 5.4 to check if a recommended sequence q produced by the above process fits with the context of the current code sequence p (function `ContextFit`, line 19). For example, from analyzing the current code via PPA to build semantic tokens, SLAMC knows that the last method call in the current code p has less number of arguments than that of parameters specified in its sememe; the context is then detected as an incomplete method call.

Then, based on the type of context of p , SLAMC checks if q fits with p as they are connected. If an expression is expected, SLAMC will check if q is a syntactically correct expression and has the expected type in the context p . If the context is a method call, it will check if q contains the expression that has the correct type of the next parameter for the method in p . If the context is an infix expression, then the result statement of connecting p and q must have the form of $X \diamond Y$, where X and Y are two valid expressions and have data types compatible with operator \diamond . A similar treatment is used for a condition statement in which a Boolean expression is expected to be formed. If a context cannot be recognized due to incomplete code, `ContextFit` returns false.

Computing Relevance Scores

The relevance score of a new path qc is computed incrementally by (*) as $P(qc|p) = R(c).P(q|p)$, in which $R(c)$ is the relevance score of the token c to the current search path. Initially, $R(c)$ is computed as $P(c|pc_1c_2\dots c_i)$ using the SLAMC model ϕ . Since ϕ models only local context, $R(c)$ is adjusted for other factors. First, if c is a token for

a control keyword, or a method call, the maximal pair-wise association probability $P(c|b)$ for every $b \in pc_1c_2...c_i$ is selected for adjusting. Otherwise, if c is a token for a variable, $R(c)$ is adjusted based on the distance r , in terms of tokens, from the position of its declaration to the cursor. $R(c)$ is multiplied by $\lambda = 1/\log(r + 1)$: the more distant the declaration of a variable, the lower its relevance to the current position.

5.3.5.3 Transforming to Lexical Forms

The transformation of a sequence q is done by creating the sequence of lexemes for the tokens in q . This task is straightforward since the lexeme is available in a token. However, SLAMC also adds the syntactic sugars for correctness (line 7). For instance, in `CALL[String, length, 0, Integer]`, the lexeme is `length`, and the method call has no argument. Thus, the lexical form `length()` is created with added parentheses. Finally, the lexical forms will be suggested in the original ranking.

5.3.6 Graph-Based Statistical Language Model

Let us start with an example that explains the challenges when using the sequence-based models (e.g., n -gram model) for code suggestion. Figure 5.7 shows the code in which the book data is read from a text file via the classes `File` and `Scanner`, and then stored into another text file via `File` and `FileWriter` of the JDK library. After the metadata is read (lines 7–8), a `while` loop (lines 10–15) is used to iterate over all the lines in the first text file to retrieve the authors' data, and write to the second file.

From the example, we see that to achieve a programming task, developers use the Application Programming Interface elements (**API elements**, APIs for short), which

```

1 File bookFile = new File("books.txt");
2 Scanner bookSc = new Scanner(bookFile);
3
4 File authorFile = new File("authors.txt");
5 FileWriter authorFW = new FileWriter(authorFile);
6
7 BookMetaData metaData = getMetaData("bookMetaData.txt");
8 metaData.printData();
9
10 while (bookSc.hasNextLine()) {
11     String bookInfo = bookSc.nextLine();
12     if (isValid(bookInfo)) continue;
13     String authorInfo = getAuthorInfo(bookInfo);
14     authorFW.append(authorInfo + System.lineSeparator());
15 }
16
17 authorFW.close();
18 bookSc.close();

```

Fig. 5.7 An API usage example illustrating issues with sequence-based code completion models

are the *classes*, *methods*, and *fields* provided by a framework or a library. The API elements are used in some order with specific data and/or control flow dependencies, and control structures (i.e., condition/repetition) according to the API specification. A usage of API elements is used to achieve a programming task and is called an **API usage**. An example of a usage is for reading data from a file, involving `File.new`, `Scanner.new`, `Scanner.hasNextLine`, `Scanner.nextLine`, `Scanner.close`, and the control unit `while`.

Using a sequence (e.g., *n*-gram) to capture common usages (called *patterns*) and give code suggestions would face the following challenges:

1. Total order. The API elements in a usage do not always have a specific required order, e.g., the instantiations of the `Scanner` and `FileWriter` objects. However, *n*-gram requires a total order among APIs. Thus, it would not consider two sequences `FileWriter.new-Scanner.new` and `Scanner.new-FileWriter.new` as two instances of the same pattern. Using a graph representation can help to represent the partial order among APIs.

2. Interleaving between the code of patterns and project-specific code. The code for the pattern “reading from a file and writing to another” is interleaved with the code for the project-specific logic, e.g., reading metadata (lines 7–8), checking information validity (line 12), or retrieving authors’ information (line 13). An *n*-gram could incorrectly include the nearby tokens of the project-specific usage (e.g., `getAuthorInfo` or `isValid`) into a pattern, leading to incorrect suggestion.

3. API elements of the same pattern can be far apart in the code. Reading from a file involves `File.new`, `Scanner.new` (lines 1–2), the `while` loop with `Scanner.hasNextLine` (line 10), `Scanner.nextLine` (line 11), and `Scanner.close` (line 18). Such patterns cannot be captured well with *n*-grams with limited lengths. However, considering dependency graphs among API elements can help to connect them together.

4. Non-sequential order of editing. For example, a developer could write the body of the `while` loop (line 10) before writing its condition. If (s)he requests for code suggestion at the condition, *n*-gram will use the code prior to line 10 for suggestion. However, the code after that, such as `bookSc.nextLine`, could suggest the use of `bookSc.hasNextLine` at the condition of the `while` loop, since they often go together. This also suggests us to expand the context for code suggestion to include the code after the requested location.

To address those issues, Nguyen et al. [30] propose GRALAN, a graph-based statistical language model. Let us present it in the context of its application of API code suggestion where it is applied to the graphs representing API usages. However, GRALAN is general for any graphs extracted from code. For the concepts specifically applicable to API suggestion, we will explicitly state so.

5.3.6.1 API Usage Representation

Definition 14 (API Usage) An API usage is a set of related API elements (i.e., classes, method calls, field accesses, and operators) in use in the client code, together with control units (i.e., condition and repetition) in a specific order, and with the control and data flow dependencies among API elements [31].

GRALAN is based on a graph-based representation model, called *Groum* to represent API usages.

Definition 15 (Groum [33]) A Groum is a graph in which the nodes represent actions (i.e., method calls, overloaded operators, and field accesses) and control points (i.e., branching points of control units `if`, `while`, and `for`). The edges represent the control and data flow dependencies between nodes. The nodes' labels are from the names of classes, methods, or control units.

An API usage can be represented by a connected (sub)graph in a Groum. In Fig. 5.8, $P_2(g)$ illustrates the pattern on `FileWriter` as a Groum. The action nodes such as `File.new` and `FileWriter.new` represent API calls, field accesses, or operators. The nodes' labels have fully qualified names, and an action node for a method call also has its parameters' types (not shown). An edge connects two action nodes if there exist control and data flow dependencies between them. For example, `FileWriter.new` must be executed before `FileWriter.append` and the object created by the former is used in the latter call, thus, there is an edge from the former to the latter. If a usage involves a `while` loop (e.g., in Fig. 5.5), a control node named `WHILE` is created after the node for the condition and is connected to the first node in the body of `while`. If a method call is an argument of another call, e.g., `m(n())`, the node for the call in the argument will be created before the node for the outside method call (i.e., the node for `n` comes before that of `m`). The rationale is that `n` is evaluated before `m`.

5.3.6.2 Generation Process

A graph can be constructed from one of its subgraphs by adding nodes and edges. Thus, the graph generation process can be modeled by the addition of nodes and edges to already-constructed subgraphs. The following concepts are defined as follows:

Definition 16 (Parent and Child Graphs) A connected graph $P(g)$ is a parent graph of a graph g if adding a new node N and inducing edges from N to $P(g)$ will create g . g is a child graph of $P(g)$. A child graph of g is denoted as $C(g)$. A graph can have multiple parents and multiple children.

This relation is general for any graph. However, let us illustrate it via Fig. 5.8 for API usage graphs (Groums). The graph $P_1(g)$ is a parent graph of g because adding the node `File.new` and the edge `File.new-FileWriter.new` to $P_1(g)$ will create g . g also

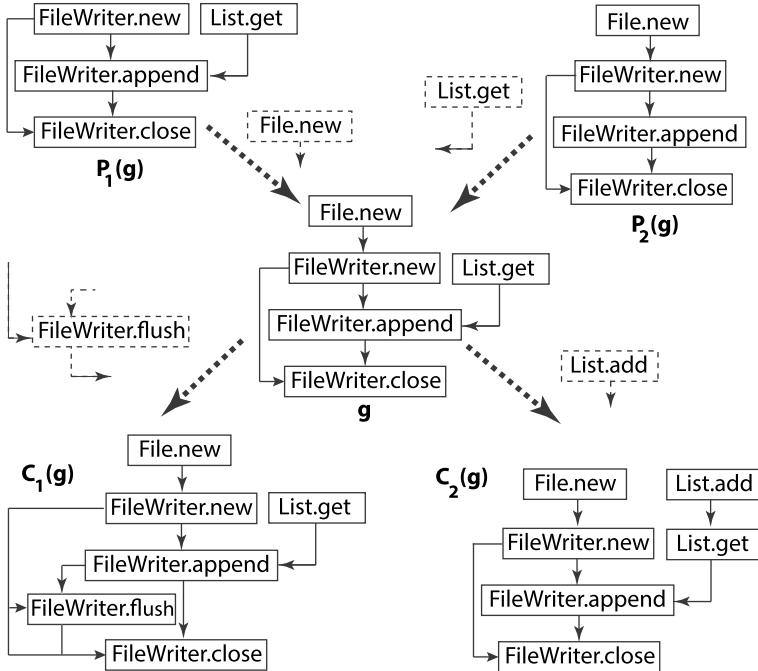


Fig. 5.8 Parent and child graphs

has its children $C_1(g)$ and $C_2(g)$. The suggestion of a new API given an already-observed Groum g can be done by considering all of its children $C(g)$'s. The concept of parents is extended to ancestors and that of children is extended to descendants.

Definition 17 (Context) The context of a generation process of a new graph $C(g)$ from a graph g is a set of graphs including g that are used to generate $C(g)$.

$Pr(C(g)|Ctxt)=Pr((g, N^+, E^+)|Ctxt)$ is used to denote such generation probability. N^+ is the additional node and E^+ is the list of additional edges connecting g and N^+ to build $C(g)$. All the graphs in $Ctxt$ including g affect the generation of $C(g)$. For the API suggestion application, the context contains the subgraphs g_1, \dots, g_n (of the Groum G built from the code) that surround the current editing location. Those subgraphs represent the potential usages that are useful in the prediction. For each child graph generated from a subgraph g_i , the corresponding additional nodes N_j 's will be collected and ranked. Each new node will be added to G to produce a candidate graph G' as a suggestion (see details in Sect. 5.3.6.3).

5.3.6.3 GRALAN in API Element Suggestion

Let us explain how GRALAN is used to build an engine for suggesting the next API element for the current code. The suggestion task for API elements is to recommend an API element upon request at a location in the current code under editing (not necessarily at the end). An example of partially edited code is shown in Fig. 5.9a. A developer requests the engine to suggest an API call at the `while` loop (line 11).

Algorithm

Overview. The key idea of the API suggestion algorithm is to extract from the currently edited code the usage subgraphs (Groups) surrounding the current location, and use them as the context. Then, the algorithm utilizes GRALAN to compute the probabilities of the child graphs given those usage subgraphs as the context. Each child graph has a corresponding additional node, which is collected and ranked as a candidate of API element for suggestion. Those probabilities are used to compute the scores for ranking the candidates.

Detailed Algorithm. Figure 5.10 shows the pseudo-code of the code completion algorithm. The input includes the current code C , the current location L , and the trained model with graph database GD . First, Eclipse's Java parser is used to create the AST for the current code. If the incomplete code under editing is not parsable by the parser, the Partial Program Analysis (PPA) tool [10] is run. The PPA tool accepts

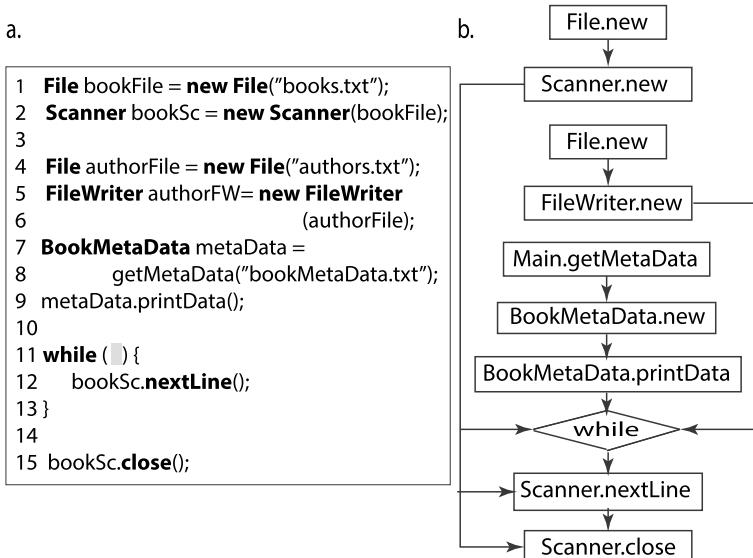


Fig. 5.9 An API suggestion example and API usage graph

```

1 function APISuggestion(Code C, Location L, GraphDatabase GD)
2   G = BuildGroum(C)
3   Ctxt = GetContextGraphs(G, L)
4   NL = ∅ // a ranked list of recommended nodes
5   foreach g ∈ Ctxt
6     {C(g)} = GetChildrenGraphs(g, GD)
7     foreach C(g) ∈ {C(g)}
8       score = log(Pr(C(g)|Ctxt))
9       NM = GetAddedNode(C(g))
10      NL = UpdateRankedNodeList(NL, NM, score)
11  return NL
12 end

```

Fig. 5.10 API suggestion algorithm

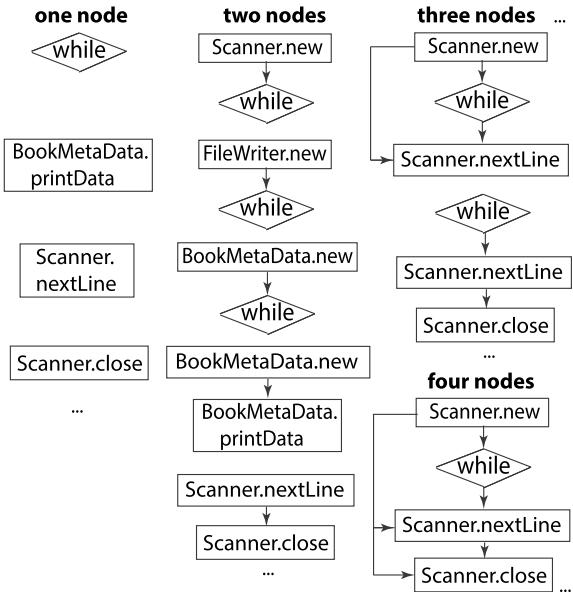
a portion of code and returns an AST with all available type binding information. However, in some cases, there might exist some unresolved nodes, for example, their syntactic or data types are undetermined. Thus, they are assigned with an `unknown` type. Then, the Groum is built from the AST using the Groum building algorithm [33] (line 2). Due to the possible incompleteness of the current code, the unresolved nodes in the AST (if any) are considered as single-node graphs. Their labels are the lexemes. The Groum of the code in Fig. 5.9a is shown in Fig. 5.9b.

Next, `APISuggestion` determines the list of context graphs from the Groum G and the current location L (line 3). The graphs that contain the APIs surrounding L are considered as the context. One or more of those context graphs are potentially the graphs that “generate” the child graphs in which the corresponding additional nodes are the candidates to be filled in at L . They represent the usages with a high impact on the selection of the API to be filled. Details on context graphs are in Sect. 5.3.6.3. Figure 5.11 shows the context graphs for the code in Fig. 5.9.

Then, for each graph g in the context, the algorithm searches in the graph database GD of GRALAN to determine all feasible child graphs $C(g)$ s (line 6). The score that each child graph $C(g)$ would be generated is computed (line 8). The respective additional nodes for those child graphs are collected (line 9) and ranked based on the computed probabilities as the candidate APIs for suggestion (line 10).

Table 5.6 shows a few examples of the context graphs and their corresponding child graphs for the example in Fig. 5.9. In the interest of space, we show the graphs as a sequence of the nodes’ labels. The respective additional nodes of the child graphs are written in bold, e.g., `Scanner.hasNextLine` is from the child graph #4 in Table 5.6. Moreover, an additional node N^+ from a child graph $C(g)$ will assume the location L in the code. The relative order between N^+ and other nodes in $C(g)$ must be consistent with their corresponding order in the graph G . For example, the API `Scanner.nextLine` is after both the current location L and `WHILE`. Thus, the child graphs $C(g)$ s with `Scanner.nextLine` appearing before N^+ or before `WHILE` are not considered. Any graph $C(g)$ with its additional node N^+ violating that condition will not be used. The graphs 6–8 in Table 5.6 conform to that condition. Such checking is part of `UpdateRankedNodeList` in line 10 of Fig. 5.10. Note that in a Groum, the node

Fig. 5.11 Context subgraphs



for the condition of a `while` loop appears before the `WHILE` node. In Table 5.6, the child graphs $C(g)$ s with N^+ (in bold) connecting to `WHILE` are still valid.

The probability that a node is added to G is estimated by the probability that the respective child graph is generated given its context. Table 5.7 shows the examples of candidate APIs. Each candidate might be generated by more than one parent graph. Thus, its highest score is used for ranking. For example, the additional node `Scanner.hasNextLine` appears in the two child graphs 4 and 6. Finally, the node with the highest score could be used to be filled in the requested location L . The additional edges E^+ s are determined from the corresponding $C(g)$ s, but they are not needed for API code suggestion. A user just uses the suggested API with their chosen arguments.

Building Database GD of Parent and Child Graphs

For GRALAN to work on API code suggestion/completion, one needs to build a database of Groums. GrouMiner [33] can be used to do so. To identify the parent and child (sub)Groums, an algorithm can traverse a Groum in a depth-first order and expand from a smaller parent graph by adding a new node and inducing edges to get a child graph. The process is repeated until all nodes/edges are visited. Nguyen et al. [30] have used 1,000 GitHub projects with 100K classes and +600K methods to build a database of almost 800M graphs. Among them, 55M are distinctive. This size of a graph database has given an excellent level of performance [30].

Table 5.6 Context graphs and their child graphs

g_i	$C(g_i)$	score
WHILE	1. Scanner.hasNext WHILE	0.010
	2. StringTokenizer.hasMoreElements WHILE	0.015
	...	
	3. Scanner.new Scanner.hasNext WHILE	0.200
Scanner.new WHILE	4. Scanner.new Scanner.hasNextLine WHILE	0.150
	5. Scanner.new Scanner.hasNextChar WHILE	0.050
	...	
	null, i.e., no child graph in GD (project-specific)	0.000
WHILE Scanner.nextLine	6. Scanner.hasNextLine WHILE Scanner.nextLine	0.700
	7. Scanner.hasNext WHILE Scanner.nextLine	0.050
	8. Scanner.hasNextChar WHILE Scanner.nextLine	0.000

Table 5.7 Ranked candidate nodes

Node	Scores	Highest score
Scanner.hasNextLine	0.15, 0.7	0.7
Scanner.hasNext	0.01, 0.2, 0.05	0.2
Scanner.hasNextChar	0.05, 0.0	0.05
String Tokenizer.hasMoreElements	0.015	0.015

Determining Context Subgraphs

To determine the context graphs, at the current location L , the algorithm collects the surrounding API calls. A threshold θ is used to limit the number of such calls. The closer to L an API call is in the code, the higher priority it has. In Fig. 5.9a, if $\theta = 4$, the surrounding API elements are `metaData.printData()`, `while`, `bookSC.nextLine()`, and `bookSC.close()`. Thus, we collect into a set S the nodes `BookMetaData.printData`, `WHILE`, `Scanner.nextLine`, and `Scanner.close`. From those nodes, the algorithm expands them to all the subgraphs in G that satisfy the following: (1) containing at least one API in S , and (2) having sizes smaller than a threshold δ . δ is also used to limit the

number of context graphs, which can increase exponentially. For example, given the set S of `BookMetaData.printData`, `WHILE`, `Scanner.nextLine`, `Scanner.close`, and $\delta = 5$, the context graphs are partially listed in Fig. 5.11.

5.4 Deep Learning for Code Completion

5.4.1 Deep Neural Network language Model

Neural Network Language Models (NNLMs) have been explored in NLP [4, 25, 26, 36–38]. NNLM has been shown to perform better than n -gram model due to a key advanced mechanism called *word embedding*, in which each word is projected into a continuous-valued feature space.

Recent research in deep learning brings many advantages to the use of neural network. Researchers have conducted experiments to show that DNN layers can be trained with proper model configurations to capture high abstraction levels of the inputs [3, 20, 25, 26, 38].

Figure 5.12 displays the general architecture of a DNN language model to predict the next token. Each token in a dictionary is represented by a 1-hot vector in which the position corresponding to the index of the token is 1 and other positions are set to 0. The model typically has three layers. The input of the model is the concatenation of $n - 1$ vectors in a sequence. The role of the first layer is to project the input vectors into a continuous space that could produce better representations for the input tokens. At the output layer, for prediction, we have different outputs in which each of them represents the probability of a token with index i being the next token. For training, the actual n th token has its $P(\text{lex}_n = j | h_n) = 1$, and other output values are 0.

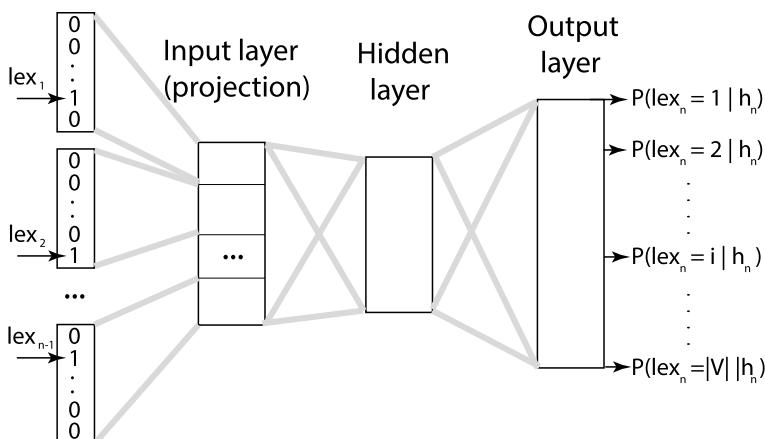


Fig. 5.12 Deep neural network language model (DNN LM) [3]

5.4.2 Recurrent Neural Network Language Model

The *recurrent neural network LM* (RNN LM) [14, 26, 34] was also applied to lexical code tokens. RNN LM represents the word history via learning from data with back-propagation through time. The input at the iteration t , $x(t)$, is formed by concatenating the vector representing the current word w , and the output from the context/history layer s at the previous iteration $t - 1$. The vector for the new context is computed from the input $x(t)$, and the output vector is computed from that new context with an activation function. By using recurrent connections, information can cycle back for longer than $n - 1$ prior words [26].

5.4.3 Deep Neural Network Language Model with Syntactic and Type Information

DNN4C [29] is a DNN-based LM for source code that incorporates syntactic and type contexts (Fig. 5.13).

Incorporating syntactic and type information as contexts. While the traditional DNN LM uses only $n - 1$ prior lexical tokens (words), DNN4C attempts to parse the current file and derive the syntactical types and the data types for the code tokens (if possible), and use the sequences of syntactic units and types as contexts. The idea from DNN4C is that with more precise information on the current syntactic unit and on data/token types, the model will be able to *capture patterns at higher abstraction levels*, thus, leading to more correct suggestion of the next token. For example, in Fig. 5.13, with the token `hasNext` and the sememe `CALL [Scanner, hasNext, 0, boolean]` being in the contexts, GRALAN could rank the token `next` of `Scanner` higher in the output since `hasNext` of a `Scanner` object is often followed by a call to `next`.

Multiple-prototype model. Instead of using only one DNN for all sequences at three levels, DNN4C has as input each code lexeme and its syntax and type contexts into two additional DNNs, each of which is dedicated to incorporating one type of

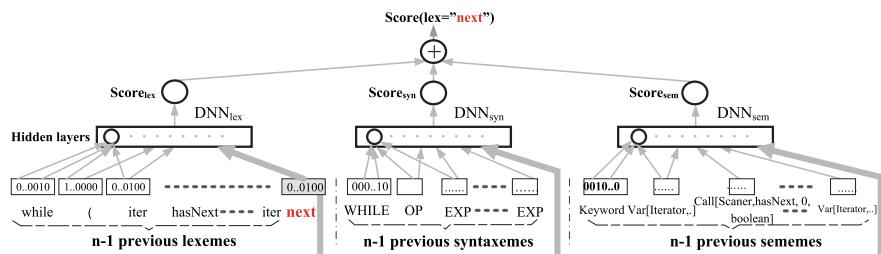


Fig. 5.13 Context-aware DNN: incorporating syntactic and type contexts

context (see Fig. 5.13 for the model). As shown in Huang et al. [18], using a single DNN would not capture well different meanings of the same lexeme (e.g., `next`) in different contexts (e.g., `LinkedList.next` or `Scanner.next()`) as the model is influenced by all of them. They showed that using multiple DNNs for multiple representations in different contexts captures well a word in different usages. When word meaning is still ambiguous given the local context, it is expected that information in other contexts can help disambiguation.

Training objectives. There are the following objectives in training in DNN4C: (1) to train the first DNN to learn to determine the potential next code token based on the $n - 1$ previous lexemes, and (2) to train the two additional DNNs for contexts to discriminate each correct next token c from other tokens in the vocabulary *given the window of $n - 1$ previous lexemes and the syntactic/type contexts of that token c* . That is, the score should be large for the actual next token, compared to the score for other tokens. Specifically, lex is used to denote the current sequence of $n - 1$ prior lexemes. The training goal is to discriminate the actual next token c (appearing after lex) from the other tokens in the vocabulary. Let $Score_{syn}$ and $Score_{sem}$ be the scoring functions for two DNNs modeling syntactic and type contexts. The goal is that with the input lex , they give the scores $Score_{syn}(c, syn)$ and $Score_{sem}(c, sem)$ for the correct token c much higher than the scores $Score_{syn}(c', syn)$ and $Score_{sem}(c', sem)$ for any other token c' in the vocabulary. syn and sem are the sequences of $n - 1$ prior syntactic units and $n - 1$ prior type units representing the contexts for c and lex . In general, one can use any sub-sequences of the syntactic units and type units for the tokens from the beginning of a file to c as contexts. However, performance will be an issue when the sequences are long. Thus, the same length ($n - 1$) for the sequences of lexical and syntactical units is used.

As an example, the training aims to have the scores $Score_{syn}$ and $Score_{sem}$ for the token `next` of `Scanner` to be higher than the scores for other tokens. Mathematically, as in [8, 18], a training objective $O(c, syn)$ is used that minimizes the ranking loss for each pair of tokens c and sequence syn in a file, and gives the margin of 1 between two such scores. Thus, for the sequence lex ending with c , it is computed as

$$O(c, syn) = \sum_{c' \in V} \max(0, 1 - (Score_{syn}(c, syn) - Score_{syn}(c', syn)))$$

If the margin between the two scores for c and c' is smaller than 1, the 2nd argument in the \max function is greater than 0. If the margin is greater than 1, the \max function returns 0, helping the objective O reach its minimum. Thus, via the \max function, one can minimize that ranking loss for (c, syn) . The same training objectives $O(c, lex)$ and $O(c, sem)$ are used for the sequences of lexical and syntactical units.

5.4.4 Other Neural Models for Code Completion

Li et al. [21] propose a pointer mixture network, which can predict the next word by either generating it from the global vocabulary or copying a word from the local context. The intuition is that instead of replacing out-of-vocabulary words with a special word, they observe that when coding, the variables tend to be repeated locally. A variable’s name may repeat several times and has a relatively high frequency. When predicting such unknown words, their model learns to choose one location in the local context and copy the word at that location as a prediction.

Their tool can predict the next word by either generating one from the global vocabulary or copying a word from the local context. The first component for the former task is a standard RNN with attention, which is called the global RNN component. For the latter task, they use a pointer network that shares the same RNN architecture and attention scores.

A recent work by Kim et al. [19] proposes a code completion approach by making the Transformer architecture aware of the syntactic structure of source code. The intuition of the work has two folds. First, the authors leverage the advances in Transformers in NLP community which have been shown to achieve better results than RNNs for a variety of NLP tasks such as language modeling, question answering, and sentence entailment. Second, inspired by several works from the software engineering community, to improve the accuracy, they enable the machine learning system to encounter code structures. They explore two models that represent two ways to capture the (partial) structure of an AST: one, based on decomposing the tree into paths (PathTrans), and the other, based on a tree traversal order (TravTrans) [19].

5.5 Conclusion

As the third most important feature in an IDE, code completion helps speed up the process of coding by filling in the desired code and reducing common mistakes. In the early IDEs, traditional code completion tools supported a simple completion of a method called for a currently editing variable. The traditional code completion tools have leveraged program analysis, however, they produce a large number of candidates with equal occurrence likelihoods, despite that the candidates are syntactically or semantically valid. The long, alphabetically sorted list of possible suggested code is not exactly helpful for developers.

In the evolution of the field, to improve the ranking of possible candidates, other code completion approaches have leveraged *software mining* and *information retrieval* techniques. Via mining, a CC tool could suggest and rank higher the more frequent code (i.e., *a code pattern*). The issues with software mining-based code completion approaches include (1) hard thresholds on frequent code, and (2) rare code not to be completed. Therefore, statistical approaches have been proposed to improve those two aspects of code completion. Those approaches could implicitly

derive the code patterns and suggest the code to be completed. Finally, the advances in machine learning and deep learning have opened more opportunities for researchers in further learning to suggest the next program units.

As a natural step moving forward, there are promising future directions. First, the combination of program analysis and data-driven approaches needs to be further explored. A few first steps in this direction has been explored [32]. The promising second direction is to explore the approaches to support the completion of more volume of code in the current editing session. Some approaches have supported the completion of entire statements [32] or entire code patterns [31]. However, integration between different lines of approaches is still almost in non-existence. Third, most CC approaches assume the editing order from left to right. In reality, a developer can edit in arbitrary order. Fourth, the evaluation of most CC approaches is still limited in a simulation fashion. More human studies should be conducted to evaluate the usefulness of those CC tools. Finally, an approach that can take advantage of other sources of information during the development process is desirable. Currently, only the source code is considered for a code completion process.

References

1. Using google N-Gram corpus. <http://googlesystem.blogspot.com/2008/05/using-googles-n-gram-corpus.html>
2. M. Acharya, T. Xie, J. Pei, J. Xu, Mining api patterns as partial orders from source code: from usage scenarios to specifications, in *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (ACM, 2007), pp. 25–34
3. E. Arisoy, T.N. Sainath, B. Kingsbury, B. Ramabhadran, Deep neural network language models, in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, WLM '12, (Association for Computational Linguistics, 2012), pp. 20–28
4. Y. Bengio, R. Ducharme, P. Vincent, C. Janvin, A neural probabilistic language model. *J. Mach. Learn. Res.* **3**, 1137–1155 (2003)
5. M. Bruch, M. Monperrus, M. Mezini, Learning from examples to improve code completion systems, in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09*, (ACM, 2009), pp. 213–222
6. M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F.P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W.H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A.N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba, Evaluating large language models trained on code (2021)
7. M. Ciniselli, N. Cooper, L. Pasarella, A. Mastropaoletti, E. Aghajani, D. Poshyvanyk, M.D. Penta, G. Bavota, An empirical study on the usage of transformer models for code completion. *IEEE Trans. Softw. Eng.* **01**, 4818–4837 (5555). <https://doi.org/10.1109/TSE.2021.3128234>

8. R. Collobert, J. Weston, A unified architecture for natural language processing: deep neural networks with multitask learning, in *Proceedings of the 25th International Conference on Machine Learning, ICML '08*, (ACM, 2008), pp. 160–167
9. Copilot. <https://copilot.github.com/>
10. B. Dagenais, L. Hendren, Enabling static analysis for partial java programs, in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA '08*, (ACM, 2008), pp. 313–328
11. Eclipse. www.eclipse.org
12. D. Engler, D.Y. Chen, S. Hallem, A. Chou, B. Chelf, Bugs as deviant behavior: a general approach to inferring errors in systems code, in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, (ACM, 2001), pp. 57–72
13. V.J. Hellendoorn, S. Proksch, H.C. Gall, A. Bacchelli, When code completion fails: a case study on real-world completions, in *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, (IEEE Press, Piscataway, NJ, USA, 2019), pp. 960–970 <https://doi.org/10.1109/ICSE.2019.00101>
14. M. Hermans, B. Schrauwen, Training and analysing deep recurrent neural networks, in *Advances in Neural Information Processing Systems 26*, (2013), pp. 190–198
15. R. Hill, J. Rideout, Automatic method completion, in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE '04*, (IEEE Computer Society, 2004), pp. 228–235
16. A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, (IEEE Press, 2012), pp. 837–847
17. A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, (IEEE Press, 2012), pp. 837–847
18. E.H. Huang, R. Socher, C.D. Manning, A.Y. Ng, Improving word representations via global context and multiple word prototypes, in *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers - Volume 1, ACL '12*, (Association for Computational Linguistics, 2012), pp. 873–882
19. S. Kim, J. Zhao, Y. Tian, S. Chandra, Code prediction by feeding trees to transformers, in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, (2021), pp. 150–162 <https://doi.org/10.1109/ICSE43902.2021.00026>
20. H. Le, I. Oparin, A. Allauzen, J.L. Gauvain, F. Yvon, Structured output layer neural network language models for speech recognition. *IEEE Trans. Audio Speech Lang. Process.* **21**(1), 197–206 (2013)
21. J. Li, Y. Wang, M.R. Lyu, I. King, Code completion with neural attention and pointer networks, in *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI'18*, (AAAI Press, 2018), pp. 4159–4165
22. B. Livshits, T. Zimmermann, Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes* **30**(5), 296–305 (2005)
23. C.D. Manning, H. Schütze, *Foundations of Statistical Natural Language Processing*, (MIT Press, Cambridge, MA, USA, 1999)
24. M. Marasoiu, L. Church, A.F. Blackwell, An empirical investigation of code completion usage by professional software developers, in *PPIG* (2015)
25. T. Mikolov, A. Deoras, D. Povey, L. Burget, Cernocky, J.: Strategies for training large scale neural network language models, in *Proceedings of the IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), ASRU'11*, (IEEE, 2011)
26. T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, S. Khudanpur, Recurrent neural network based language model, in *Proceedings of International Conference on Acoustics Speech and Signal Processing (ICASSP), ICASSP'10*, (IEEE, 2010), pp. 1045–1048
27. G.C. Murphy, M. Kersten, L. Findlater, How are java software developers using the eclipse ide? *IEEE Softw.* **23**(4), 76–83 (2006). <https://doi.org/10.1109/MS.2006.105>

28. A.T. Nguyen, M. Hilton, M. Codoban, H.A. Nguyen, L. Mast, E. Rademacher, T.N. Nguyen, D. Dig, Api code recommendation using statistical learning from fine-grained changes, in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, (Association for Computing Machinery, New York, NY, USA, 2016), pp. 511–522 <https://doi.org/10.1145/2950290.2950333>
29. A.T. Nguyen, T.D. Nguyen, H.D. Phan, T.N. Nguyen, A deep neural network language model with contexts for source code, in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (2018), pp. 323–334. <https://doi.org/10.1109/SANER.2018.8330220>
30. A.T. Nguyen, T.N. Nguyen, Graph-based statistical language model for code, in *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, (IEEE Press, 2015), pp. 858–868
31. A.T. Nguyen, T.T. Nguyen, H.A. Nguyen, A. Tamrawi, H.V. Nguyen, J. Al-Kofahi, T.N. Nguyen, Graph-based pattern-oriented, context-sensitive source code completion, in *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, (IEEE Press, Piscataway, NJ, USA, 2012), pp. 69–79. <http://dl.acm.org/citation.cfm?id=2337223.2337232>
32. S. Nguyen, T.N. Nguyen, Y. Li, S. Wang, *Combining Program Analysis and Statistical Language Model for Code Statement Completion*, (IEEE Press, 2019), pp. 710–721. <https://doi.org/10.1109/ASE.2019.00072>
33. T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen, Graph-based mining of multiple object usage patterns, in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09*, (ACM, 2009), pp. 383–392
34. R. Pascanu, Ç. Gülcöhre, K. Cho, Y. Bengio, How to construct deep recurrent neural networks. CoRR <abs/1312.6026> (2013)
35. R. Robbes, M. Lanza, How program history can improve code completion, in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, (IEEE Computer Society, 2008), pp. 317–326
36. R. Sarikaya, M. Afify, B. Kingsbury, Tied-mixture language modeling in continuous space, in *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, NAACL '09*, (Association for Computational Linguistics, 2009), pp. 459–467
37. H. Schwenk, Continuous space language models. Comput. Speech Lang. **21**(3), 492–518 (2007)
38. H. Schwenk, J.L. Gauvain, Training neural network language models on very large corpora, in *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing, HLT '05*, (Association for Computational Linguistics, 2005), pp. 201–208
39. C.C. Williams, J.K. Hollingsworth, Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans. Softw. Eng. **31**(6), 466–480 (2005)
40. Y. Yang, Y. Jiang, M. Gu, J. Sun, J. Gao, H. Liu, A language model for statements of software code, in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, (IEEE Press, 2017), pp. 682–687

Chapter 6

Cloud Development and Deployment



José Antonio Parejo and Ana Belén Sánchez

Abstract This chapter describes the challenges related to the software deployment and configuration activities, which could be addressed through AI techniques. Additionally, we illustrate the application of such techniques with two specific problems: the QoS-aware binding of web services, and the selection of the optimal configuration in highly-configurable systems. The AI techniques applied to solve such problems are *GRASP with Path Relinking* and *Multi-Objective Evolutionary Algorithms*, respectively. These techniques will be applied to deployment scenarios based on empirically obtained data from real-world web services and highly-configurable software systems. These examples are implemented in Java and C, respectively, providing the source code, build scripts and data sets for the sake of reproducibility.

6.1 Introduction

In this section, we introduce the activities of deployment and configuration of the software development lifecycle, such as the late-binding of dependencies, and the deployment of the components into an infrastructure which could be hosted on premise or in the cloud.

According to the Official Scrum Guide [26], once the software components have been implemented, tested and accepted in the context of an iteration/sprint, a new increment is generated that could be deployed and delivered to the end users. This process of deployment/delivery could vary significantly depending on the architecture (monolith, based on micro-services, based on a highly-configurable system, etc.) and the type of application (web application/API, desktop, mobile, etc.). According to ISO 12207-2017 [1], those activities are structured into two technical processes,

J. Antonio Parejo (✉) · A. B. Sánchez

Smart Computer Systems Research and Engineering Lab (SCORE), Research Institute of Computer Engineering (I3US), Universidad de Sevilla, Seville, Spain
e-mail: japarejo@us.es

A. B. Sánchez
e-mail: anabsanchez@us.es

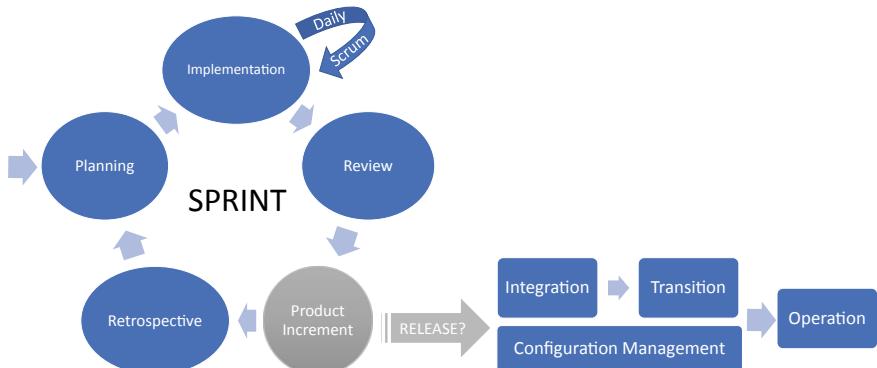


Fig. 6.1 Software process activities in a Scrum team including release and deployment of increments

namely, *integration* and *transition*, that are complemented by some of the activities of the *configuration management* process, such as *release control* and *configuration change management* (Fig. 6.1).

The purpose of the integration process is to synthesise a set of components into a realised system that can be deployed. The output of such a process is a new deployable artefact containing the added functionality developed in the sprint. The transition process moves the artefact in an orderly and planned manner into the operational status, possibly installing/delivering the new version of the system under development and handling/configuring its enabling systems, leading to the start of the operation process for the just deployed system.

In the context of Highly-Configurable Systems (HCSs) and Software product Lines (SPLs) [7, 22], the integration and configuration management for release activities are specially complex, since the combination of possible components to integrate, and the set of configuration options available, are extensive. Furthermore, the constraints between component compatibility and configuration values can be quite complex in those scenarios, which leads to the use of AI as a logical way to assist in the execution of those activities.

In the context of Service-Based Applications (SBAs) deployed on the cloud, the set of available components and their configurations to be taken into account in the integration, transition and management configuration phases can change dynamically. Specifically, a service can become unavailable and other functionally equivalent alternatives can appear in the service registry. Moreover, even with a constant set of available services, new Service Level Agreements (SLAs) with different pricing or Quality of Service (QoS) guarantees can be offered by those same providers, leading to possible improvements in terms of operating cost reductions and superior QoS. In those scenarios, AI techniques can modify the configuration of consumed services/SLAs (named binding) on runtime to better exploit such a changing landscape of possible improvements.

In Sect. 6.2, we review the challenges that can be addressed through AI techniques in those processes, providing references to the well-known problems and suitable approaches that have been identified. Additionally, we further describe the two specific problems that will be addressed in depth in this chapter. In Sect. 6.3, we focus on solving the QoS-aware web service binding problem using a search-based technique named GRASP with Path Relinking. In Sect. 6.4, we focus on solving the optimal configuration selection problem in HCSs, using a multi-objective evolutionary algorithm. Furthermore, we present a running example for both problems of different sizes, ranging from a small service-based application to a real-world production-grade highly-configurable system with billions of configurations representing the Drupal framework. Finally, we draw the conclusions of this work and provide insights into the future applications of AI in the later phases of the software development lifecycle in Sect. 6.5.

6.2 Open Problems in the Context of Software Configuration, Integration and Transition

In this section, we delve into how current software development and architecture trends involve decision-making on the deployment and configuration activities of the software development lifecycle that are suitable for aid and automation using AI-based techniques.

To perform such a prospective study, we applied the methodology described in [16]. The sources used to search for the papers were Scopus, Google Scholar and Web of Science. The search strings used to conform the set of papers reported were ‘software deployment Artificial Intelligence’, ‘devops Artificial Intelligence’, ‘software configuration Artificial Intelligence’, ‘Software product line artificial intelligence’ and ‘highly-configurable system artificial intelligence’. We followed up to 5 pages of results in each search engine, performing a first filtering based on the title of the references. A second filtering phase was performed based on the reading of the abstract leading to a final set of papers to perform a full read. In the next subsection, we report the results using narrative synthesis.

Harman stated in [11] that the first step for the successful application of any AI technique to any Software Engineering problem is to find a suitable formulation of the software engineering problem so that AI techniques become applicable. Hence, apart from a wide scope look-out into the problems and applied AI techniques in the activities of software deployment and configuration, in this section we will introduce two problems in detail: (i) the QoS-aware web service binding problem, and (ii) the optimal configuration selection problem. After their introduction in this section, such problems will be formulated and addressed through search-based techniques in Sects. 6.3 and 6.4, respectively.

6.2.1 Open Problems and Suitable AI Techniques

The need of properly configuring software systems adapting them to their deployment environment has been there since the dawn of software engineering. However, the ever-increasing number of components to be integrated, the extreme levels of configurability of some of those components, the disparate plethora of possible deployment environments, and the increased frequency of updates and re-deployments associated with modern software development lifecycles, has increased the complexity and costs of such tasks to the point in which the use of AI techniques to aid and support them is not only suitable, but even essential.

Highly-Configurable Systems (HCSs) provide a common core functionality and a set of optional features. These systems appeared in response to the growing customer demand to customise their systems [7, 22]. An example of HCSs is the *Amazon Elastic Compute Cloud* service which offers 1,758 different possible configurations [10]. In this context, the scenario becomes more complicated, involving thousands or even millions of different possible system configurations. AI techniques have been used extensively in HCSs, e.g. logic-based representation and reasoning, ontological modelling and reasoning, constraint satisfaction techniques, and optimization with evolutionary algorithms [3]. Optimization is the selection of the best candidate from a set of possible candidates, and evolutionary algorithms can help to select the optimal solution from the set of candidates by applying the mechanics of natural evolution of species. From the perspective of HCSs, the ultimate configuration derivation objective is to find the feature set that better fulfils the customer's requirements. Furthermore, optimization algorithms have been used to exploit the customer preferences such as low cost and minimum number of defects to search an optimal selection of features in a given HCS. We describe this problem in detail in Sect. 6.2.3. In [13], the authors proposed a new approach to enhance the search of a multi-objective evolutionary algorithm to solve the optimal product selection problem. Their approach named $1 + n$ first optimised the number of constraints that fail, and then the rest of objectives. Thus, they invalidated products that did not meet the constraints of the HCS since they had no value to the software engineers. In Sect. 6.4, the optimal configuration selection problem is further formalised along with the adaptions required to apply a multi-objective evolutionary algorithm, and a real-world instance of the problem based on the Drupal content management system is solved through such an algorithm.

AI techniques have been used not only to configure the software system itself but also to configure its deployment environment. Modern deployment environments, such as Kubernetes and Apache Mesos, provide some extent of automated support for the dynamic deployment and retirement of docker containers and Virtual Machines (VMs) based on rules and user preferences. Those mechanisms can be improved using AI techniques that could improve elasticity [29] and reduce lost requests [30].

Regarding the deployment of software systems, AI techniques have been used for several purposes. For instance, to generate a plan of deployments and reconfigurations, that either optimises the usage of the underlying infrastructure in distributed

software systems [4], or that achieves desirable properties such as scalable rollback [25] or non-stop deployment [18]. Those approaches become even more essential when we transition from component-based software systems to cloud-based or service-based software systems, where AI-based techniques are applied [12] to (i) create predictive models of client demands, costs, QoS properties and behaviours in cloud deployments; (ii) identify provision and deployment plans of resources for the VMs and docker containers, in order to improve the QoS, minimise SLA violations, or infrastructure utilisation; (iii) identify the service providers to be invoked and the plans or SLAs to be used [19]. Such identification of the service providers and plans or SLAs to be used, when driven by the preferences of the users on the QoS properties and guarantees offered by the providers, is named the QoS-aware web services binding problem. This problem is further described next, and it is formalised and solved using a search algorithm in Sect. 6.3.

Regarding future directions in this area, new deployment models like serverless solutions are consolidating; AWS Lambda, Azure Functions or Google Cloud Functions are some of the providers that offer this kind of solutions. Such new deployment models can be used in a combination with traditional public, on premise or hybrid cloud deployments, generating a complex landscape of deployment options. Furthermore, novel architectural styles like micro-services architectures promote the development of a plethora of small components that must interoperate to implement business logic, for which the corresponding deployment option should be chosen. Consequently, practitioners will need to optimise even more complex deployment models, and bigger problem instances in the future, which makes the need to apply artificial intelligence techniques more pressing. Apart from that, the application of modern software development methodologies and practices such as continuous deployment and agile methodologies involves that re-deployments will be more frequent. Such re-deployments could involve the re-evaluation of the deployment option of choice, making the use of a manual approach to solve these problems even more impracticable.

6.2.2 *The QoS-aware Web Service Binding Problem*

Web services may include information about non-functional properties that affect their quality, so-called quality of service (QoS) attributes, e.g. cost, availability, etc. When several providers expose web services that are functionally equivalent through compatible interfaces, or a provider supports different endpoints with different QoS guarantees, QoS properties can be used to drive the selection of the candidate provider or endpoint to invoke. For instance, one may choose the most reliable and expensive endpoint, the cheapest one, or a third alternative that provides a balance among those properties.

The QoS-aware binding of web services enables the creation of context-aware and auto-configurable applications that can adapt itself depending on the available services/endpoints and user preferences. For instance, consumers could specify

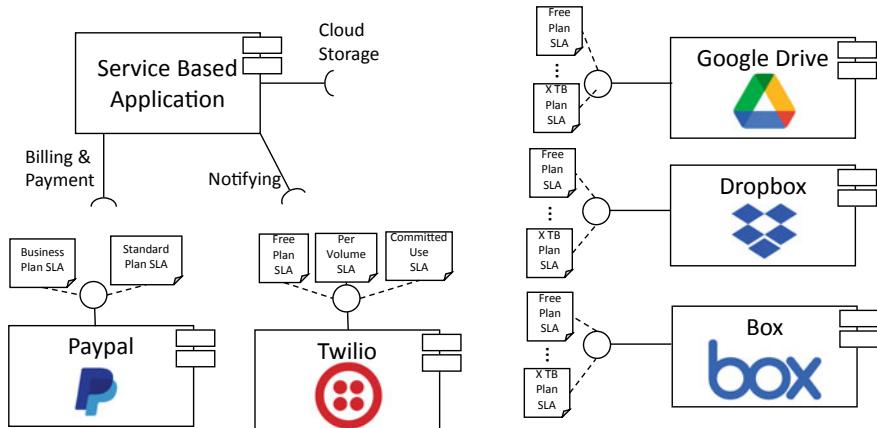


Fig. 6.2 Service based application without binding with multiple SLAs per provider

constraints like ‘The total cost per invocation must be lower than 1\$’ and QoS criteria such as ‘choose faster providers instead of cheaper ones’.

The QoS-aware web service binding problem consists of determining the optimal binding; i.e. the set of service providers (or endpoints with their associated SLAs) to invoke that meet user constraints and optimises user satisfaction according to some QoS criteria. Figure 6.2 shows a decorated UML component diagram that depicts an scenario where a Service-Based Application consumes several web services through the *Cloud Storage*, *Billing and Payment* and *Notifying* interfaces.

The *Cloud Storage* interface has three service providers, namely, GDrive, Dropbox and Box, where each provider offers several SLAs, with different maximum storage and rate of invocation, and different availability guarantees. We assume that the providers and SLAs are mutually interchangeable since they are functionally equivalent, and we could develop clients for all those providers with a common interface. The *Billing and Payment* and *Notifying* interfaces have one single provider, but each provider offers different SLAs, with different billing limitations and fees in the case of Paypal, and different volume of notifications per month and cost per invocation in the case of Twilio—this service provides different alternatives for client notification such as SMS, e-mail and even programmable phone calls.

Even in such a simple scenario, assuming only two alternative SLAs/plans per Cloud Storage provider, we would have up to 36 different bindings available, each one with its specific cost, global availability guarantees of the services, rate limitations, etc. Thus, the use of AI techniques to choose the most appropriate binding is a sensible approach for more complex scenarios or for scenarios with a high frequency of changes in the SLAs or available providers.

6.2.3 Optimal Configuration Selection Problem in HCSs

A Highly-Configurable System (HCS) provides a common core functionality and a set of optional features, where a feature represents an increment in system functionality [7, 22]. Examples of HCSs are operating systems such as *Debian Wheezy*, which offers more than 37,000 available packages [2] or content management systems like *Prestashop* which supports more than 3,500 modules and visual templates [28].

HCSs are usually represented in terms of features. A *feature* depicts a choice to include a certain functionality in a system configuration [22]. It is common that not all combinations of features are allowed or meaningful. In this case, additional constraints are defined between them, normally using a variability model, such as a feature model. A *feature model* represents all the possible configurations of the HCS in terms of features and constraints among them [15]. A *configuration* is a valid composition of features satisfying all the constraints. Figure 6.3 depicts a simplified feature model representing the variability of an HCS of online shopping systems. The hierarchical relationship among features can be divided into

- Mandatory. If a feature has a mandatory relationship with its parent feature, it must be included in all the configurations in which its parent feature appears.
- Optional. If a feature has an optional relationship with its parent feature, it can be optionally included in all the configurations including its parent feature.
- Alternative. A set of child features has an alternative relationship with their parent feature when only one of them can be selected when its parent feature is part of the configuration.
- Or. A set of child features has an or-relationship with their parent when one or more of them can be included in the configurations in which its parent feature appears.

In addition, a feature model can also contain cross-tree constraints among features. These are usually of the form:

- Requires. If a feature A requires a feature B, the inclusion of A in a configuration implies the inclusion of B in such configuration.
- Excludes. If a feature A excludes a feature B, both features cannot appear in the same configuration.

The feature model in Fig. 6.3 illustrates that *Catalogue*, *Payment* and *Security* are mandatory features that must appear in all the configurations, while *Search* feature is optional. The model also indicates that online shops can provide payment support for *Bank Transfer*, *Credit Card* or the combination of both at the same time. On the other hand, an online shop may implement a *High* or an *Standard* security policy but not both in the same configuration. Also, the inclusion of *Credit Card* payment requires support for *High* security policy. Finally, an online shop rules out the possibility of offering support for *High* security policy and *Public Report* in a search in the same configuration. For instance, the following set of features represents a valid

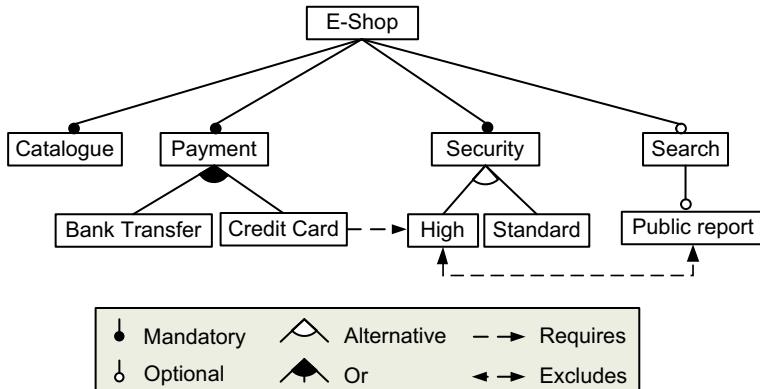


Fig. 6.3 E-Shop feature model

Table 6.1 E-Shop feature attributes

Feature	Size	CC	Tests	Installations	Developers	Changes
Catalogue	650	0.16	127	10,000	22	30
Payment	479	0.43	432	10,000	18	82
Bank transfer	258	0.32	511	4,367	8	68
Credit card	389	0.24	267	9,222	6	33
Security	803	0.61	914	10,000	31	51
High	501	0.42	432	5,009	28	40
Standard	288	0.29	803	3,991	17	19
Search	466	0.58	805	6,045	15	73
Public report	321	0.29	623	4,246	9	25

configuration of the model: {E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}.

Feature models can also be extended with additional information by means of feature attributes, these are called attributed or extended feature models [5]. Feature attributes are often defined as tuples $\langle name, value \rangle$ specifying non-functional information of features such as cost or memory consumption. As an example, Table 6.1 depicts six different feature attributes inspired on a previous work of the authors [24] and their values on the features of the model in Fig. 6.3: Size (size of features in terms of their lines of code), CC (Cyclomatic complexity of features' code), Tests (number of test assertions related to the features), Installations (number of reported installations of each feature) Developers (number of developers involved in the development of each feature) and Changes (number of commits made in each feature).

We consider the problem of automatically choosing the optimal configurations from a feature model based on a set of user preferences, being represented as a many-objective optimisation problem. The result of a search may be used, for example, to determine which configurations to release first. In order to decide whether a configuration is optimal or not, we can use the attributes of the features in a feature model. For example, one might prefer configurations that contain many features since these could satisfy the demands of more customers, or those configurations containing features with a low cost or complexity.

6.3 Solving the QoS-aware Web Service Binding Problem

We address the problem of QoS-aware web service binding using a hybrid algorithm that combines two search-based algorithms, GRASP and Path Relinking (PR). Specifically, we use GRASP to generate a set of bindings that is then used as input in PR to generate the optimal solution. This algorithm has provided good results for the QoS-aware web service binding problem in the past [19], and its hybrid nature provides diversity in the set of techniques applied to solve SE problems in this book. Since PR is not described in the specific chapter devoted to algorithms of this book, we provide a brief introduction to such algorithm next.

6.3.1 Path Relinking

Path Relinking (PR) is a metaheuristic optimization technique that generates new solutions by exploring trajectories connecting promising solutions. The basic hypothesis is that by exploring the region of the search space between promising solutions we will find more promising solutions. The working scheme of PR is shown in Fig. 6.4.

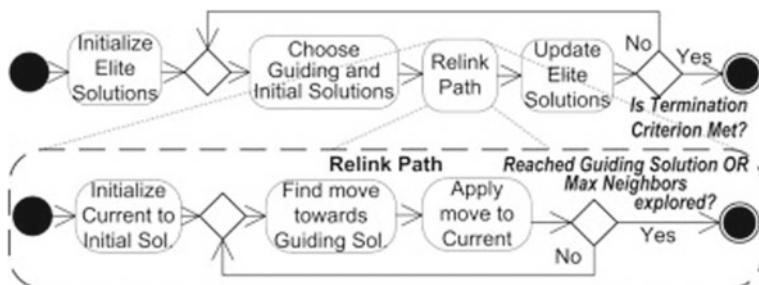


Fig. 6.4 Path Relinking working scheme

PR manages a set of promising solutions named the ‘elite set’. In each iteration, until the meeting of a termination criterion, PR randomly chooses two solutions from the elite set, named the initiating and guiding solution. Then, PR generates a sequence of successive solutions from the initiating to the guiding solution [17]. Each step is generated by replacing elements of the initial solution with the corresponding elements of the guiding solution. In the QoS-aware binding of services, the elements are the specific provider and plan chosen for a task. For instance, in the example problem depicted in Fig. 6.2, we would replace the provider and plan chosen for the *Cloud Storage* in the initial solution—that could be for instance the free Dropbox plan—with the provider and plan chosen in the guiding solution—for instance the Google Drive paid plan.

After reaching the guiding solution the elite set is optionally updated. For instance, the best solution found could be added, the initiating and/or guiding solutions could be removed, etc. The key parameters of PR are the number of paths explored N_{paths} between each pair of initiating and guiding solutions, and the maximum number of steps explored per path N_{steps} .

6.3.2 Objective Function

Objective functions guide search algorithms towards a specific solution whose properties maximise or minimise such function. For the problem of the QoS-aware binding of services, such function should encode the user preferences regarding the different QoS properties and their balance, such as cost, availability and rate limitations. Although this problem can be formulated and solved in a natural way as a multi-objective problem, we adopted a mono-objective approach to address it, using a weighted sum of the expected utility perceived by the user for the value of each QoS property.

For instance, in Table 6.2 the QoS guarantees and values provided for each available SLA and provider are shown. The values shown in the table were collected by analysing the actual SLAs provided by the services, or using real monitoring data published in different sources¹ for those services which do not provide guarantees on some of the properties. The cost is measured on a monthly basis per service (in the case of Twilio, we show the cost per invocation, but we have assumed an average value for 500 notifications per month, to get uniform units for the property in cost per month). The availability is measured in percentage of invocations, and the storage is measured in gigabytes.

The calculation of the global QoS value for a binding χ is crucial for solving this problem. For some properties such as the cost, the individual QoS value is present for all the available providers and plans, thus the global cost is computed as the sum of the cost of the corresponding chosen services. In other cases, such as the cloud

¹ For instance the availability of Dropbox was taken form <https://digitalfirst.com/dropbox-availability-finally-revealed/>.

Table 6.2 QoS values per service provider and plan

Service	Provider	Plan SLA	Qos properties		
			Cost	Availability	Storage
Billing & Payment	Paypal	Business card	0.420	Unspecified	–
		Business phone	12.250	Unspecified	–
Notifying	Twilio	Pay as you go	0.075	99.95	–
		Per volume	0.050	99.99	–
		Committed use	0.030	99.99	–
Cloud storage	GDrive	Free	0.000	99.95	15
		Paid	1.990	99.95	100
	Dropbox	Free	0.000	99.63	2
		Paid	9.990	99.63	2048
	Box	Free	0.000	99.00	10
		Paid	9.000	99.00	100

storage capacity, the property is only specified for one of the services to bind (and its corresponding providers), thus the global value would be that of the chosen provider and plan for such service. Finally, there are other properties such as the availability, where the global availability of a binding would be the product of the individual availabilities guaranteed by the chosen providers and plans. Interested readers can find taxonomies of possible aggregation functions for the usual QoS properties in the literature depending on the usage workflow, but in general this aggregation is defined by the users [6].

Given that cost is a negative property, i.e. the lower the cost the better, we could use as utility function for the cost $U_1(\text{cost}) = 1/\text{cost}$. However, such a formulation would be a source of problems for free services, where the value of cost is 0. Thus, we would prefer as utility function for the cost $U_2(\text{cost}) = -\text{cost}$. Moreover, in order to ease the creation of a balance between the different QoS properties to be taken into account, the value of the utilities for each QoS property will be normalised, providing a value between 0 and 1, where 0 is the minimum utility and 1 is the maximum. This means that the actual formulation of the utility for the cost is

$$U_{\text{cost}}(\chi) = \frac{\text{Max}_{\text{cost}} - \text{cost}(\chi)}{\text{Max}_{\text{cost}} - \text{Min}_{\text{cost}}}$$

where Max_{cost} and Min_{cost} are the maximum and minimum possible costs of any feasible binding, respectively. For instance, in our example, Max_{cost} is $9.99 + 0.075 * 500 + 12.25 = 59.74$ choosing the most expensive provider available for each service, Min_{cost} is $0 + 0.03 * 500 + 0.42 = 15.42$. Hence, the utility for the cost property of the binding $\chi = \{\text{Paypal-business-card}, \text{Twilio-pay-as-you-go}, \text{Gdrive-Paid}\}$, whose total cost is 17.41, would be

$$U_{cost}(\chi) = \frac{Max_{cost} - cost(\chi)}{Max_{cost} - Min_{cost}} = \frac{59.74 - 17.41}{59.74 - 15.42} = (42.33/47.32) = 0.8945.$$

Conversely, for positive properties such as availability (or maximum cloud storage capacity), where the higher the value, the better, the normalised utility function is

$$U_{availability}(\chi) = \frac{availability(\chi) - Min_{availability}}{Max_{availability} - Min_{availability}}.$$

Such a balance between the utilities of the QoS properties is expressed through a weight factor per property w , leading to the formulation of the objective function for the QoS-aware Services Binding problem:

$$\begin{aligned} U(\chi) = \sum_{q \in Q} w_q * U_q(\chi) &= w_{cost} * U_{cost}(\chi) + w_{availability} * U_{availability}(\chi) \\ &\quad + w_{storage} * U_{storage}(\chi) \end{aligned}$$

where Q is the set of quality properties taken into account, w_q is the weight associated to the property q and U_q is the utility function defined for the property q . The weights define the relative importance of each property. For instance, $w_{Cost} = 0.2$ and $w_{availability} = 0.1$ means cost is twice as important as availability for the user. In this chapter, we will take into account only three QoS Properties: $Q = \{\text{cost}, \text{availability}, \text{storage}\}$ with the values shown in Table 6.2.

6.3.3 Solution Encoding

In order to apply GRASP and Path Relinking to solve the problem, a suitable encoding of solutions is needed. An encoding is the mechanism used for expressing the characteristics of solutions in a form that facilitates its manipulation by the algorithm. Vector-based encoding structures are suitable for this problem, and has been extensively used in the literature [6]. Specifically, solutions are encoded as a vector of integer values, with a size equal to the number of services to bind. Thus, value j at position i of this vector encodes the choice of provider SLA j for service i .

For instance, in our motivating example, the binding [Paypal-business-card, Twilio-pay-as-you-go, Dropbox-Free] is encoded as the vector [1, 1, 3]. Note that the index of each provider is determined by the order of appearance in Table 6.2; e.g. for cloud storage, Dropbox-free would be the third option available, encoded as 3.

6.3.4 Constraints Handling

GRASP and PR do not directly support the optimization of constrained problems. Thus, to overcome this drawback, an alternative objective function is used. This variant adds a penalization term in a similar way as [6] using a weight w_{unf} , and a function Df that measures the distance of a binding χ from a full constraint satisfaction. Thus, our final function to be maximised is $F(\chi) = \sum_{q \in Q} w_q * U_q(\chi) - w_{unf} * Df(\chi)$.

In the case of simple constraints where we impose a minimum or maximum value for the global value of a QoS property, such as for instance ‘*the cost of the binding should be lower or equal to 20\$*’, the function of distance to satisfaction—if the constraint is not met—could be the absolute value of the difference between the threshold and the actual QoS of the binding, divided by the global range of the values of the property, to provide a value between 0 and 1. If the constraint is met, the value would be zero.

For instance, for the binding $\chi = \{\text{Cloud Storage} \rightarrow \text{Dropbox-Free}, \text{Payment \& Billng} \rightarrow \text{Paypal-Business Phone}, \text{Notification} \rightarrow \text{Twilio-Committed Use}\}$, has a total cost of $cost(\chi) = 0 + 12.25 + 0.03 * 500 = 27.25$, and distance to satisfaction for the above mentioned constraint of $(27.25 - 20)/44.32 = 7.25/44.32 = 0.163$.

6.3.5 GRASP Building Phase

The most important decision regarding the solution building phase required to adapt GRASP to a specific problem is the strategy for building the Restricted Candidate List (RCL). To solve the QoS-aware web service composition problem, we use the classical threshold $\tau = g_{min}^f + \alpha * (g_{max}^f - g_{min}^f)$ strategy [21], which is based on a greediness parameter (named α) and a greedy function g , where g_{max}^f and g_{min}^f are the maximum and minimum values of the greedy function for all the available features. Thus, a feature p will be part of the RCL if $g(p) \geq \tau$. The greediness parameter α is a value between 0 and 1 that determines the elitism applied to the elements of the RCL. For instance, if $\alpha = 0$, all the elements will be included in the RCL, and the building phase of GRASP becomes a random generation procedure. Conversely, if $\alpha = 1$ only the best available feature (in terms of the greedy function g) is included in the RCL, and the building phase of GRASP becomes an hill climbing algorithm with g as the objective function. In [19] up to six greedy functions $G_{1\dots 6}$ are formulated for this problem, we use $G_1(f) = \sum_{q \in Q} w_q * U_q^f(f)$ since it is the most simple, it provides good results, and it is based on the utility of the service and the weights defined by the user. For positive properties, where the larger QoS value provided the better, the local utility function would be $U_q^f(f) = (f_q - q_{min})/(q_{max} - q_{min})$, and for negative properties, it would be $U_q^f(f) = (q_{max} - f_q)/(q_{max} - q_{min})$. For instance, for the *cost* QoS property, $q_{min} \equiv cost_{min}$ is 0, since some plans are free; $q_{max} \equiv cost_{max}$ is 12.25 which is the cost of the most expensive service plan for any task; and the

local utility of the paid plan of Google Drive would be $U_{cost}^f(GDrive - Paid) = (12.25 - 1.99)/(12.25 - 0) = 0.837$ since the cost is a negative property and the cost of the paid plan of Google Drive is 1.99 monetary units.

6.3.6 GRASP Improvement Phase

In the improvement phase of the GRASP algorithm, we used a simple hill climbing algorithm, with a neighbourhood structure consisting of all the possible bindings which share all the service provider plans with the current solution except for one; i.e., we searched for the service plan exchange on a single task that improves the largest possible improvement.

6.3.7 Adaption of Path Relinking

The set of elite solutions in Path Relinking was created by repeatedly executing the GRASP algorithm, generating a hybrid search algorithm [21]. Moreover, the guiding and initial solutions of the path to be linked were chosen randomly among the set of elite solutions.

6.3.8 Running Example

In order to illustrate the application of GRASP with Path Relinking, we applied it to the running example depicted in Fig. 6.2 with the QoS values shown in Table 6.2. Regarding the weights of the objective function, we used the values shown in Table 6.3. Please note that the choice of the weights will guide the search process, and consequently, they should be chosen carefully. Interested readers can delve deeper on this topic and on how to use alternative approaches in [27]. Regarding the parameters of the algorithm, the number of iterations of GRASP used for generation was 5, and we linked up to 3 paths, exploring a maximum of 2 neighbours on each path.²

If we run the algorithm without any constraint, the binding provided by the algorithm is: $\chi_{unconstrained}^* = \{\text{Cloud Storage} \rightarrow \text{GDrive-Free}, \text{Payment \& Billng} \rightarrow \text{Paypal-Business card}, \text{Notification} \rightarrow \text{Twilio-Committed Use}\}$ which is the global maximum with a value of 0.791. We mapped the indexes into the corresponding service provider plans for the sake of readability.

If a constraint specifying a minimum storage capacity of 20 GB is added, the previous binding becomes unfeasible, and the binding provided by the algorithm

² Note that it makes no sense to explore more neighbours since the maximum size of the paths is 3 (the number of different services to bind).

Table 6.3 Weights used for the optimization

Weight	w_{cost}	$w_{availability}$	$w_{storage}$	$w_{unfeasibility}$
Value	0.5	0.3	0.2	0.6

is $\chi_{storage \geq 20}^* = \{\text{Cloud Storage} \rightarrow \text{Dropbox-Paid}, \text{Payment \& Billing} \rightarrow \text{Paypal-Business card}, \text{Notification} \rightarrow \text{Twilio-Committed Use}\}$ which is again the global optimum among the feasible solutions with a value of 0.674. The total execution times in both cases were less than 1 s. The execution infrastructure for all the examples and experiments in this chapter was a Surface Pro 4 laptop with 8 GBs of RAM and an i5 Intel processor using Windows 10. As execution environments, the Oracle JDK 11 virtual machine was used for Java, and Visual Studio 2019 Community was used for C.

6.4 Solving the Optimal Configuration Selection Problem in Highly-Configurable Systems

We addressed the problem of selecting the optimal configuration in HCSs using Multi-Objective Evolutionary Algorithms (MOEA). Specifically, we used the ‘1+n’ approach presented by Hierons et al. [13] that considers one first objective as being more important than the others. The first objective (number of cross-tree constraint violations) is viewed as the main objective to be considered first and then the remaining objectives as secondary objectives to be optimised equally using a multi-objective paradigm. In particular, we focused on the NSGA-II algorithm [9] to solve the problem. However, for solving large problems in production with four objectives or more, the use of many-objective algorithms such as NSGA-III [8], SPEA2 [32] or MOEA/D [31] is advised. In this case, we use NSGA-II since it works well with the N+1 approach and it has provided good results for this problem in the past even with such a number of objectives [14]. In the following, we will refer to the approach adopted as ‘NSGA-II 1+n’.

We next define the problem of the optimal configuration selection in HCSs.

6.4.1 Solution Encoding

Given the set of configurations of an HCS represented by a feature model, we present the following definition in our approach:

Configuration. A configuration is a set of features of the feature model. A configuration is valid if its features satisfy the constraints of the model. The natural representation is to define a configuration as a binary string with N bits, being N the

number of features in the feature model. The order of each feature in each configuration corresponds to the depth-first traversal order of the tree. For a feature F_i there is a corresponding gene, with value 1 if the configuration contains F_i and 0 otherwise. Note that for efficiency reasons, mandatory features are safely removed from input feature models.

If the search returns a set $C0$ of features then we generate the corresponding candidate configuration by applying the following steps: (1) Add all core features, (2) Add any feature F_i not in $C0$ such that $C0$ contains one or more children of F_i , and apply this step repeatedly until no more features can be added.

6.4.2 Constraints Handling

The ‘NSGA-II 1+n’ approach uses two simple rules to compare individuals: (1) prefer the individual with fewer cross-tree constraint violations and (2) choose the individual with better fitness (determined by the remaining objectives functions) when the number of violated cross-tree constraints is equal using a multi-objective perspective; i.e. pareto-dominance is applied, generating a set of non-dominated solutions according to the remaining objectives, with the same number of violated constraints. The aim of this proposal is to produce more valid configurations, providing the software engineer with a wider range of configurations from which to choose.

6.4.3 Objective Functions

We propose to use eight objective functions for the problem of the optimal configuration selection. These functions were successfully applied by Hierons et al. to select the optimal product [13]. Most of these functions were defined by the authors of this work in [20, 24]. All the objective functions receive an attributed feature model, representing the HCS under test, and a configuration as inputs and return an integer value measuring the quality of the configuration with respect to the optimization. The eight functions are described below.

1. **Correctness function.** This function calculates the number of constraints that were not satisfied by the configuration [13]. This value should be minimised. As an example, consider the following two configurations from the feature model in Fig. 6.3 and their features’ attributes in Table 6.1: $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, Standard}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$. This objective function would return 1 and 0 from $C1$ and $C2$ configurations, respectively. That is, the configuration $C2$ does not violate any constraint in the feature model in Fig. 6.3, hence, the algorithm would prefer that configuration.

2. **Richness of features.** This function calculates how many features were included in the configuration [13]. This measure is based on the idea that one might prefer the configurations that contain many features since these could satisfy the demands of more customers. Hence, this value should be maximised. As an example, consider the following two valid configurations from the feature model in Fig. 6.3 and their features' attributes in Table 6.1: $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$. This objective function would return 7 and 6 from $C1$ and $C2$ configurations, respectively. Hence, the algorithm would give preference to configuration $C1$ based on this function.
3. **Feature size.** The size of a feature, in terms of its Number of Lines of Code (LoC), has been shown to provide a rough idea of the complexity of the feature and its error proneness [24]. This function should be minimised to give preference to configurations with smaller features. As an example, consider the following two valid configurations from the feature model in Fig. 6.3 and their features' attributes in Table 6.1: $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$. This objective function would return 3,080 and 2,478 from $C1$ and $C2$ configurations, respectively. Hence, the algorithm would give preference to configuration $C2$ based on this function.
4. **Cyclomatic Complexity.** This metric reflects the total number of independent logic paths used in a program and provides a quantitative measure of its complexity. In a previous work, the authors showed that the cyclomatic complexity could be a helpful indicator to estimate the fault-proneness of features [24]. Hence, this function should be minimised to give preference to configurations with lower cyclomatic complexity. For instance, consider the following two valid configurations from the feature model in Fig. 6.3 and their features' attributes in Table 6.1: $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$. This objective function would return 2.18 and 1.81 from $C1$ and $C2$ configurations, respectively. Hence, the algorithm would give preference to configuration $C2$, which presents a lower value.
5. **Test Assertions.** This metric computes the total number of test cases and test assertions of each feature in a feature model [24]. The function should be maximised based on the intuition that the configurations with features with a higher number of test assertions should be less error-prone. As an example considering the configurations used in previous functions: $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$, this function would return 2,683 and 2,787 from $C1$ and $C2$ configurations, respectively. Hence, the algorithm would give preference to configuration $C2$.
6. **Number of installations.** This metric depicts the number of times that the features in a configuration have been installed. This data could be used as an indicator of

the popularity or impact of the features in a configuration [24]. Therefore, this objective function should be maximised. For instance, from the two configurations $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$, this function would return the values 48,598 and 38,358, respectively. Thus, the algorithm would give preference to configuration $C1$, which presents the higher value.

7. **Number of developers.** This measure could give us information about the scale of the features in a configuration as well as their fault-proneness related to the number of people working on them [24]. This function should be minimised. As an example, from the two configurations $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$, this objective function would return 113 and 96, respectively. Thus, the algorithm would give preference to configuration $C2$, which presents a lower number of developers working on it.
8. **Number of changes.** This metric measures the number of changes made in the features of a configuration. Changes in the code are likely to introduce defects, therefore the number of changes in a feature may be a good indicator of the fault-proneness in configurations including it [24]. Thus, this function should be minimised. For instance, considering the following configurations $C1 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Credit Card, Security, High}\}$ and $C2 = \{\text{E-Shop, Catalogue, Payment, Bank Transfer, Security, Standard}\}$, this objective function would return 304 and 250, respectively. Hence, the algorithm would give preference to configuration $C2$ based on this function.

6.4.4 Operators

Standard mutation, selection and crossover operators for the NSGA-II algorithm were used with their recommended values. In particular, we used the configuration defined by Hierons et. al in their work [13]. Specifically, bit-flip mutation with a mutation probability $pm = 1/n$ was used (where n denotes the number of decision variables, in our case the non-core features). The uniform crossover operator was used, with an overall crossover probability of $pc = 1.0$. As mating selection operator, a tournament between two randomly chosen individuals of the population was used. All those parameter values are in accordance with the recommendations in [9]. As a result of using the recommended default parameter values from the literature, we did not require a tuning phase.

6.4.5 Running Example

We carried out a running example to solve the problem of the optimal configuration selection using as input the real-world feature model of Drupal defined by the authors in [23, 24]. Drupal is a highly modular open source web content management framework written in PHP. This tool can be used to build a variety of websites including e-commerce applications or online newspapers. Furthermore, some providers offer Drupal instances using a software-as-a-service model on the cloud, such as for instance Pantheon.³

According to the work in [24], the Drupal feature model has 48 features, 21 non-redundant cross-tree constraints and it represents $2.09E9$ different valid configurations of the system. The Drupal feature model also includes attributes and their values for all its features [24]. To summarise, we used as inputs the Drupal feature model and its attributes related with the objective functions described in Sect. 6.4.3. The values of the attributes were obtained by analysing the commits available at the Drupal git repository,⁴ and the Drupal issue tracking system.⁵ The details on how the data were obtained and the derived attributes were computed are described in [23].

Since this is a considerably larger problem instance than those used to exemplify the QoS-aware service binding problem, we execute the algorithm for 50,000 objective function evaluations as the termination criterion, with a population size of 100 individuals. The algorithm was executed 30 times to reduce the impact of their stochastic nature.

The application of a multi-objective optimization paradigm involves more than one objective function being optimised simultaneously. Except in trivial systems, there rarely exists a single solution that simultaneously optimises all the objectives. In that case, the objectives are said to be conflicting, and there exists a (possibly infinite) number of so-called Pareto optimal solutions. A solution is said to be Pareto optimal (a.k.a. non-dominated) if none of the objectives can be improved without degrading some of the other objectives. Analogously, the solutions where the objectives can be improved without degrading the rest are referred to as dominated solutions.

Table 6.4 shows the average values and descriptive statistics for each objective function of the solution found in the Pareto fronts generated for the 30 runs. The total execution time of the 30 runs was 134 s using the same laptop as described for the previous problem. It is worth noting that all the provided configurations were valid, since the maximum and minimum value of constraint violations (column three) was 0. The average number of solutions in the Pareto front generated was 100 (due to the population size specified for running the algorithm).

Since the execution of the algorithm does not return a single optimal solution but a set of solutions in the Pareto front, it is up to the software engineers to choose among them the configuration that provides a better trade-off of the objective functions according to their specific preferences and needs.

³ <https://pantheon.io/blog/drupal-service>.

⁴ <https://git.drupalcode.org/project/drupal>.

⁵ <https://www.drupal.org/project/issues>.

Table 6.4 Descriptive statistics of the solutions provided in the 30 runs of the algorithms

Metric	Constraint violations	Missing features	LoC	CC	Test assertions	Installations	Developers	Changes
Median	0	21.00	203,398	9.700	16,860	66,722,484	1760	301.0
Mean	0	20.85	198,783	9.592	16,516	67,005,815	1,785	301.9
Min.	0	0.00	54,695	2.000	7,250	36,816,675	658	45.0
Max.	0	40.00	336,025	17.400	24,152	97,342,266	3,060	555.0

6.5 Conclusions and Future Trends

In this chapter, some of the most important problems of the later stages of the software development lifecycle were presented along with their most preeminent solution approaches. Moreover, two specific problems prevalent in those phases—the QoS-aware service binding problem and the optimal configuration search in Highly Configuration Systems—were developed and solved using different search-based algorithms, for problem instances of a different size and complexity. Thus, in this chapter, we have proved that AI techniques and specifically search-based optimisation algorithms can contribute to improve and partially automate the later phases of the software development lifecycle. Regarding future trends and problems, emerging approaches such as micro-services architectures and domain-driven development are leading the later phases of the development lifecycle to pose problem instances of even larger size with complex constraints, which contributes to make AI-based techniques even more suitable to support the development of large projects and aid in the work of software engineers in the future.

Reproducibility

For the sake of verifiability and reproducibility, the source code of the algorithm implementations as well as all artefacts used in the experiments and examples described in this chapter are available.⁶ The repository <https://github.com/isa-group/GRASPPRQoS AwareBinding-AISoftDevBookChapter> contains the Java source code of the GRASP and Path Relinking algorithm used to solve the QoS-aware web service binding problem, along with the data of the running example used to exemplify it along this chapter. In the repository <https://github.com/isa-group/OptimalProduct-AISoftDevBookChapter>, readers can find the C implementation of NSGA-II algorithm and the data of the Drupal feature model.

⁶ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

Acknowledgements Authors of this chapter are especially thankful to Miqing Li, which provided the C implementation of the NSGA-II algorithm applied to solve the Optimal Configuration Problem. Additionally, the authors are thankful to Sergio Segura for their support during the elaboration of this chapter. This work has been partially supported by FEDER/Ministerio de Ciencia e Innovación—Agencia Estatal de Investigación under projects HORATIO (RTI2018101204-B-C21) and OPHELIA (RTI2018101204-B-C22); and by FEDER/Junta de Andalucía under the project APOLO (US-1264651). PID2021-126227NB-C22 funded by MCIN/AEI/10.13039/501100011033/FEDER, UE and by grant TED2021-131023B-C21 funded by MCIN/AEI/10.13039/501100011033 and by NextGeneration EU.

References

1. International standard—systems and software engineering—software life cycle processes. ISO/IEC/IEEE 12207:2017(E) First edition 2017-11 pp. 1–157 (2017). <https://doi.org/10.1109/IEEESTD.2017.8100771>
2. Debian Wheezy. <http://www.debian.org/releases/wheezy/>. Accesed July 2021
3. U. Afzal, T. Mahmood, Z. Shaikh, Intelligent software product line configurations: a literature review. *Comput. Stand. Interfaces* **48**, 30–48 (2016). Special Issue on Information System in Distributed Environment
4. N. Arshad, D. Heimbigner, A. Wolf, Deployment and dynamic reconfiguration planning for distributed software systems, in *Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence* (2003), pp. 39–46
5. D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analyses of feature models 20 years later: a literature review. *Inf. Syst.* **35**(6), 615–636 (2010)
6. G. Canfora, M. Di Penta, R. Esposito, M.L. Villani, Qos-aware replanning of composite web services, in *IEEE International Conference on Web Services (ICWS'05)* (IEEE, 2005), pp. 121–129
7. M.B. Cohen, M.B. Dwyer, J. Shi, Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *Trans. Softw. Eng.* **34**(5), 633–650 (2008)
8. K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE Trans. Evol. Comput.* **18**(4), 577–601 (2013)
9. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm. *IEEE Trans. Evol. Comput.* **6**(2), 182–197 (2002)
10. J. García-Galán, O. Rana, P. Trinidad, A. Ruiz-Cortés, Migrating to the cloud: a software product line based analysis, in *3rd International Conference on Cloud Computing and Services Science* (2013), pp. 416–426
11. M. Harman, The role of artificial intelligence in software engineering, in *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)* (2012), pp. 1–6
12. M. Harman, K. Lakhotia, J. Singer, D.R. White, S. Yoo, Cloud engineering is search based software engineering too. *J. Syst. Softw.* **86**(9), 2225–2241 (2013)
13. R.M. Hierons, M. Li, X. Liu, S. Segura, W. Zheng, Sip: optimal product selection from feature models using many-objective evolutionary optimization. *ACM Trans. Softw. Eng. Methodol.* **25**(2) (2016)
14. R.M. Hierons, M. Li, X. Liu, S. Segura, W. Zheng, Sip: optimal product selection from feature models using many-objective evolutionary optimization. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **25**(2), 1–39 (2016)
15. K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis (foda) feasibility study, in *SEI* (1990)

16. B.A. Kitchenham, S.L. Pfleeger, Principles of survey research part 2: designing a survey. *SIGSOFT Softw. Eng. Notes* **27**(1), 18–20 (2002)
17. M. Laguna, R. Martí, Grasp and path relinking for 2-layer straight line crossing minimization. *INFORMS J. Comput.* **11**(1), 44–52 (1999)
18. W. Li, Z. Zhao, Automating dynamic reconfiguration for non-stop dataflow systems, in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1 (2007), pp. 260–267
19. J.A. Parejo, S. Segura, P. Fernandez, A. Ruiz-Cortés, Qos-aware web services composition using grasp with path relinking. *Expert Syst. Appl.* **41**(9), 4211–4223 (2014)
20. J.A., Parejo, A.B. Sánchez, S. Segura, A. Ruiz-Cortés, R.E. Lopez-Herrejon, A. Egyed, Multi-objective test case prioritization in highly configurable systems: a case study. *J. Syst. Softw.* **122**, 287–310 (2016)
21. M.G., Resende, C.C. Ribeiro, Greedy randomized adaptive search procedures: advances, hybridizations, and applications, in *Handbook of Metaheuristics* (Springer, Berlin, 2010), pp. 283–319
22. A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, T. Berger, Presence-condition simplification in highly configurable systems, in *International Conference on Software Engineering* (2015)
23. A.B. Sánchez, S. Segura, A. Ruiz-Cortés, The drupal framework: a case study to evaluate variability testing techniques, in *Workshop on Variability Modelling of Software-Intensive Systems* (ACM, 2014), pp. 11:1–11:8
24. A.B. Sánchez, S. Segura, J.A. Parejo, A. Ruiz-Cortés, Variability testing in the wild: the drupal case study. *Softw. Syst. Model.* **J.** 1–22 (2015)
25. S. Satyal, I. Weber, L. Bass, M. Fu, Scalable rollback for cloud operations using AI planning, in *2015 24th Australasian Software Engineering Conference* (2015), pp. 195–202
26. K. Schwaber, J. Sutherland, The scrum guide. Technical report, Scrum.org (2020). <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf>
27. M.J. Scott, E.K. Antonsson, Compensation and weights for trade-offs in engineering design: beyond the weighted sum. *J. Mech. Des.* **127**(6), 1045–1055 (2005)
28. S. Segura, A.B. Sánchez, A. Ruiz-Cortés, Automated variability analysis and testing of an e-commerce site: an experience report, in *International Conference on Automated Software Engineering* (ACM, 2014), pp. 139–150
29. B. Thurgood, R.G. Lennon, Cloud computing with kubernetes cluster elastic scaling, in *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems (ICFNDS '19)*, Association for Computing Machinery, New York, NY, USA, 2019
30. L. Toka, G. Dobreff, B. Fodor, B. Sonkoly, Machine learning-based scaling management for kubernetes edge clusters. *IEEE Trans. Netw. Serv. Manage.* **18**(1), 958–972 (2021)
31. Q. Zhang, H. Li, MOEA/D: a multiobjective evolutionary algorithm based on decomposition. *IEEE Trans. Evol. Comput.* **11**(6), 712–731 (2007)
32. E. Zitzler, M. Laumanns, L. Thiele, Spea2: improving the strength pareto evolutionary algorithm. *TIK-Report* **103** (2001)

Part III

Testing and Maintenance

Chapter 7

Automated Support for Unit Test Generation



Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveira Neto,
and Robert Feldt

Abstract Unit testing is a stage of testing where the smallest segment of code that can be tested in isolation from the rest of the system—often a class—is tested. Unit tests are typically written as executable code, often in a format provided by a unit testing framework such as `pytest` for Python. Creating unit tests is a time and effort-intensive process with many repetitive, manual elements. To illustrate how AI can support unit testing, this chapter introduces the concept of *search-based unit test generation*. This technique frames the selection of test input as an optimization problem—we seek a set of test cases that meet some measurable goal of a tester—and unleashes powerful *metaheuristic* search algorithms to identify the best possible test cases within a restricted timeframe. This chapter introduces two algorithms that can generate `pytest`-formatted unit tests, tuned towards coverage of source code statements. The chapter concludes by discussing more advanced concepts and gives pointers to further reading for how artificial intelligence can support developers and testers when unit testing software.

From “Optimising the Software Development Process with Artificial Intelligence” (Springer, 2022)

A. Fontes · G. Gay (✉) · F. G. de Oliveira Neto · R. Feldt (✉)
Chalmers and the University of Gothenburg, Gothenburg, Sweden
e-mail: ggay@chalmers.se

R. Feldt
e-mail: robert.feldt@chalmers.se

A. Fontes
e-mail: afonso.fontes@chalmers.se

F. G. de Oliveira Neto
e-mail: francisco.gomes@cse.gu.se

7.1 Introduction

Unit testing is a stage of testing where the smallest segment of code that can be tested in isolation from the rest of the system—often a class—is tested. Unit tests are typically written as executable code, often in a format provided by a unit testing framework such as `pytest` for Python. Unit testing is a popular practice as it enables test-driven development—where tests are written before the code for a class, and because the tests are often simple, fast to execute, and effective at verifying low-level system functionality. By being executable, they can also be re-run repeatedly as the source code is developed and extended.

However, creating unit tests is a time and effort-intensive process with many repetitive, manual elements. If elements of unit test creation could be automated, the effort and cost of testing could be significantly reduced. Effective automated test generation could also complement manually written test cases and help ensure test suite quality. Artificial intelligence (AI) techniques, including optimization, machine learning, natural language processing, and others, can be used to perform such automation.

To illustrate how AI can support unit testing, we introduce in this chapter the concept of *search-based unit test input generation*. This technique frames the selection of test input as an optimization problem—we seek a set of test cases that meet some measurable goal of a tester—and unleashes powerful *metaheuristic* search algorithms to identify the best possible test input within a restricted timeframe. To be concrete, we use metaheuristic search to produce `pytest`-formatted unit tests for Python programs.

This chapter is laid out as follows:

- In Sect. 7.2, we introduce our running example, a Body Mass Index (BMI) calculator written in Python.
- In Sect. 7.3, we give an overview of unit testing and test design principles. Even if you have prior experience with unit testing, this section provides an overview of the terminology we use.
- In Sect. 7.4, we introduce and explain the elements of search-based test generation, including solution representation, fitness (scoring) functions, search algorithms, and the resulting test suites.
- In Sect. 7.5, we present advanced concepts that build on the foundation laid in this chapter.

To support our explanations, we created a Python project composed of (i) the class that we aim to test, (ii) a set of test cases created manually for that class following good practices in unit test design, and (iii) a simple framework including two search-based techniques that can generate new unit tests for the class.

The code examples are written in Python 3, therefore, you must have Python 3 installed on your local machine in order to execute or extend the code examples. We target the `pytest` unit testing framework for Python.¹ We also make use of the

¹ For more information, see <https://pytest.org>.

`pytest-cov` plug-in for measuring code coverage of Python programs,² as well as a small number of additional dependencies. All external dependencies that we rely on in this chapter can be installed using the `pip3` package installer included in the standard Python installation. Instructions on how to download and execute the code examples on your local machine are available in our code repository³ at <https://github.com/Greg4cr/PythonUnitTestGeneration>.

7.2 Example System—BMI Calculator

We illustrate concepts related to automated unit test generation using a class that implements different variants of a body mass index (BMI) classification.⁴ BMI is a value obtained from a person’s weight and height to classify them as underweight, normal weight, overweight, or obese. There are two core parts to the BMI implementation: (i) the BMI value, and (ii) the BMI classification. The BMI value is calculated according to Eq. 7.1 using height in metres (m) and weight in kilos (kg).

$$BMI = \frac{weight}{(height)^2} \quad (7.1)$$

The formula can be adapted to be used with different measurement systems (e.g., pounds and inches). In turn, the BMI classification uses the BMI value to classify individuals based on different threshold values that vary based on the person’s age and gender.⁵

The BMI thresholds for children and teenagers vary across different age ranges (e.g., from 4 to 19 years old). As a result, the branching options quickly expand. In this example, we focus on the World Health Organization (WHO) BMI thresholds for cisgender⁶ women, who are adults older than 19 years old,⁷ and children/teenagers between 4 and 19 years old.⁸ In Fig. 7.1, we show an excerpt of the `BMICalc` class, and in Fig. 7.2, we show the method that calculates the BMI value for adults. The complete code for the `BMICalc` class can be found at https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/example/bmi_calculator.py.

² See <https://pypi.org/project/pytest-cov/> for more information.

³ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

⁴ This is a relatively simple program compared to what is typically developed and tested in the software industry. However, it allows clear presentation of the core concepts of this chapter. After reading this chapter, you should be able to apply these concepts to a more complex testing reality.

⁵ Threshold values can also vary depending on different continents or regions.

⁶ An individual whose personal identity and gender corresponds with their birth sex.

⁷ See <https://www.euro.who.int/en/health-topics/disease-prevention/nutrition/a-healthy-lifestyle/body-mass-index-bmi>.

⁸ See <https://www.who.int/tools/growth-reference-data-for-5to19-years/indicators/bmi-for-age>.

```

1  class BMICalc:
2      def __init__(self, height, weight, age):
3          self.height = height
4          self.weight = weight
5          self.age = age
6
7      def bmi_value(self):
8          # The height is stored as an integer in cm. Here we convert it to
9          # meters (m).
10         bmi_value = self.weight / ((self.height / 100.0) ** 2)
11
12         return bmi_value
13
14     def classify_bmi_adults(self):
15         if self.age > 19:
16             bmi_value = self.bmi_value()
17             if bmi_value < 18.5:
18                 return "Underweight"
19             elif bmi_value < 25.0:
20                 return "Normal weight"
21             elif bmi_value < 30.0:
22                 return "Overweight"
23             elif bmi_value < 40.0:
24                 return "Obese"
25             else:
26                 return "Severely Obese"
27         else:
28             raise ValueError(
29                 "Invalid age. The adult BMI classification requires an age "
30                 "older than 19.")

```

Fig. 7.1 An excerpt of the BMICalc class. The snippet includes the constructor for the BMICalc class, the method that calculates the BMI value according to Eq. 7.1, and a method that returns the BMI classification for adults

```

1  def classify_bmi_teens_and_children(self):
2      if self.age < 2 or self.age > 19:
3          raise ValueError(
4              'Invalid age. The children and teen BMI classification '
5              'only works for ages between 2 and 19.')
6
7      bmi_value = self.bmi_value()
8      if self.age <= 4: ...
9      elif self.age <= 7: ...
10     elif self.age <= 10: ...
11     elif self.age <= 13: ...
12     elif self.age <= 16: ...
13     elif self.age <= 19: ...

```

Fig. 7.2 Method for the BMI classification of several age brackets that, in turn, expand further into the branches of each BMI classification. For readability, the actual thresholds were omitted from the excerpt above

The BMI classification is particularly interesting case for testing because, (i), it has *numerous branching* statements based on multiple input arguments (age, height, weight, etc.), and (ii), it requires testers to think of specific *combinations of all arguments* to yield BMI values able to cover all possible classifications. Table 7.1 shows all of the different thresholds for the BMI classification used in the BMICalc class.

Table 7.1 Threshold values used between different BMI classifications across the various age brackets. The children and teens reference values are for young girls

Classification	[2, 4]	(4, 7]	(7, 10]	(10, 13]	(13, 16]	(16, 19]	>19
Underweight	≤ 14	≤ 13.5	≤ 14	≤ 15	≤ 16.5	≤ 17.5	< 18.5
Normal weight	≤ 17.5	≤ 14	≤ 20	≤ 22	≤ 24.5	≤ 26.5	< 25
Overweight	≤ 18.5	≤ 20	≤ 22	≤ 26.5	≤ 29	≤ 31	< 30
Obese	> 18.5	> 20	> 22	> 26.5	> 29	> 31	< 40
Severely obese	—	—	—	—	—	—	≥ 40

While the numerous branches add complexity to writing unit tests for our case example, the use of only integer input simplifies the problem. Modern software requires complex inputs of varying types (e.g., DOM files, arrays, abstract data types) which often need contextual knowledge from different domains such as automotive, web or cloud systems, or embedded applications to create. In unit testing, the goal is to test small, isolated units of functionality that are often implemented as a collection of methods that receive primitive types as input. Next, we will discuss the scope of unit testing in detail, along with examples of good unit testing design practices, as applied to our BMI example.

7.3 Unit Testing

Testing can be performed at various levels of granularity, based on how we interact with the system under test (SUT) and the type of code structure we focus on. As illustrated in Fig. 7.3, a system is often architected as a set of one or more cooperating or standalone subsystems, each responsible for a portion of the functionality of the overall system. Each subsystem, then, is made up of one or more “units”—small, largely self-contained pieces of the system that contain a small portion of the overall

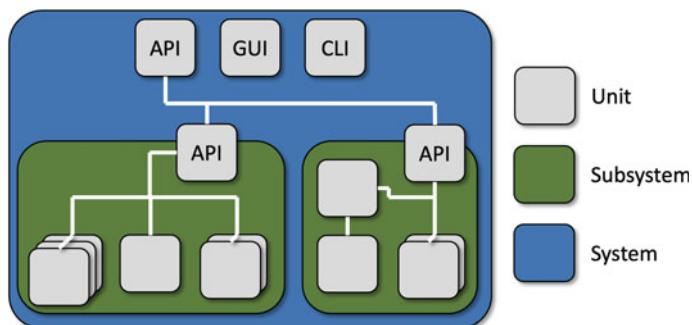


Fig. 7.3 Illustration of common levels of granularity in testing. A system is made up of one or more largely-independent subsystems. A subsystem is made up of one or more low-level “units” that can be tested in isolation

system functionality. Generally, a unit is a single class when using object-oriented programming languages like Java and Python.

Unit testing is the stage of testing where we focus on each of those individual units and test their functionality in *isolation* from the rest of the system. The goal of this stage is to ensure that these low-level pieces of the system are trustworthy before they are integrated to produce more complex functionality in cooperation. If individual units seem to function correctly in isolation, then failures that emerge at higher levels of granularity are likely to be due to errors in their *integration* rather than faults in the underlying units.

Unit tests are typically written as executable code in the language of the unit-under-test (UUT). Unit testing frameworks exist for many programming languages, such as JUnit for Java, and are integrated into most development environments. Using the structures of the language and functionality offered by the unit testing framework, developers construct *test suites*—collections of test cases—by writing test case code in special test classes within the source code. When the code of the UUT changes, developers can re-execute the test suite to make sure the code still works as expected. One can even write test cases before writing the unit code. Before the unit code is complete, the test cases will fail. Once the code is written, passing test cases can be seen as a sign of successful unit completion.

In our BMI example, the UUT is the `BMICalc` class outlined in the previous section. This example is written in Python. There are multiple unit testing frameworks for Python, with `pytest` being one of the most popular. We will focus on `pytest`-formatted test cases for both our manually written examples and our automated generation example. Example test cases for the BMI example can be found at https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/example/test_bmi_calculator_manual.py and will be explained below.

Unit tests are typically the majority of tests written for a project. For example, Google recommends that approximately 70% of test cases for Android projects be unit tests [1]. The exact percentage may vary, but this is a reasonable starting point for establishing your expectations. This split is partially, of course, due to the fact that there are more units than subsystem or system-level interfaces in a system and almost all classes of any importance will be targeted for unit testing. In addition, unit tests carry the following advantages:

- **Useful Early in Development:** Unit testing can take place before the development of a “full” version of a system is complete. A single class can typically be executed on its own, although a developer may need to *mock* (fake the results of) its dependencies.
- **Simplicity:** The functionality of a single unit is typically more limited than a subsystem or the system as a whole. Unit tests often require less setup and the results require less interpretation than other levels of testing. Unit tests also often require little maintenance as the system as a whole evolves, as they are focused on small portions of the system.
- **Execute Quickly:** Unit tests typically require few method calls and limited communication between elements of a system. They often can be executed on the developer’s computer, even if the system as a whole runs on a specialised device

(e.g., in mobile development, system-level tests must run on an emulator or mobile device, while unit tests can be executed directly on the local computer). As a result, unit tests can be executed quickly and can be re-executed on a regular basis as the code evolves.

When we design unit tests, we typically want to *test all “responsibilities” associated with the unit*. We examine the functionality that the unit is expected to offer and ensure that it works as expected. If our unit is a single class, each “responsibility” is typically a method call or a short series of method calls. Each broad outcome of performing that responsibility should be tested—e.g., alternative paths through the code that lead to different normal or exceptional outcomes. If a method sequence could be performed out-of-order, this should be attempted as well. We also want to *examine how the “state” of class variables can influence the outcome of method calls*. Classes often have a set of variables where information can be stored. The values of those variables can be considered as the current state of the class. That state can often influence the outcome of calling a method. Tests should place the class in various states and ensure that the proper method outcome is achieved. When designing an individual unit test, there are typically five elements that must be covered in that test case:

- **Initialisation (Arrange):** This includes any steps that must be taken before the core body of the test case is executed. This typically includes initialising the UUT, setting its initial state, and performing any other actions needed to execute the tested functionality (e.g., logging into a system or setting up a database connection).
- **Test Input (Act):** The UUT must be forced to take actions through method calls or assignments to class variables. The test input consists of values provided to the parameters of those method calls or assignments.
- **Test Oracle (Assert):** A test oracle, also known as an expected output, is used to validate the output of the called methods and the class variables against a set of encoded expectations in order to issue a verdict—pass or fail—on the test case. In a unit test, the oracle is typically formulated as a series of *assertions* about method output and class attributes. An assertion is a Boolean predicate that acts as a check for correct behaviour of the unit. The evaluation of the predicate determines the verdict (outcome) of the test case.
- **Tear Down (Cleanup):** Any steps that must be taken after executing the core body of the test case in order to prepare for the next test. This might include cleaning up temporary files, rolling back changes to a database, or logging out of a system.
- **Test Steps (Test Sequence, Procedure):** Code written to apply input to the methods, collect output, and compare the output to the expectations embedded in the oracle.

Unit tests are generally written as methods in dedicated classes grouping the unit tests for a particular UUT. The unit test classes are often grouped in a separate folder structure, mirroring the source code folder structure. For instance, the `utils.BMICalc` class stored in the `src` folder may be tested by a `utils.TestBMICalc` test class stored in the `tests` folder. The test methods

```

1 def test_bmi_value_valid():
2     bmi_calc = bmi_calculator.BMICalc(150, 41, 18)      # Arrange
3     bmi_value = bmi_calc.bmi_value()                      # Act
4     # Here, the approx method allows 0.01
5     # differences in floating point errors.
6     assert pytest.approx(bmi_value, abs=0.1) == 18.2 # Assert.
7
8 # Cases expected to throw exception
9 def test_invalid_height():
10    # 'with' blocks expect exceptions to be thrown, hence
11    # the assertion is checked *after* the constructor call
12    with pytest.raises(ValueError) as context:           # Assert
13        bmi_calc = bmi_calculator.BMICalc(-150, 41, 18) # Act
14
15    with pytest.raises(ValueError) as context:           # Assert
16        bmi_calc.height = 0                             # Act
17
18 def test_bmi_adult():
19    bmi_calc = bmi_calculator.BMICalc(160, 65, 21)      # Arrange
20    bmi_class = bmi_calc.classify_bmi_adults()          # Act
21    assert bmi_class == "Overweight"                     # Assert
22
23 def test_bmi_children_4y():
24    bmi_calc = bmi_calculator.BMICalc(100, 13, 4)       # Arrange
25    bmi_class = bmi_calc.classify_bmi_teens_and_children() # Act
26    assert bmi_class == "Underweight"

```

Fig. 7.4 Examples of test methods for the BMICalc class using the pytest framework

are then executed by invoking the appropriate unit testing framework through the IDE or the command line (e.g., as called by a continuous integration framework). Figure 7.4 shows four examples of test methods for the BMICalc class. Each test method checks a different scenario and covers different aspects of good practices in unit test design, as will be detailed below. The test methods and scenarios are:

- `test_bmi_value_valid()` : verifies the correct calculation of the BMI value for valid and typical (“normal”) inputs.
- `test_invalid_height()` : checks robustness for invalid values of height using exceptions.
- `test_bmi_adult()` : verifies the correct BMI classification for adults.
- `test_bmi_children_4y()` : checks the correct BMI classification for children up to 4 years old.

Due to the challenges in representing real numbers in binary computing systems, a good practice in unit test design is to allow for an error range when assessing the correct calculations of floating point arithmetic. We use the `approx` method from the pytest framework to automatically verify whether the returned value lies within the 0.1 range of our test oracle. For instance, our first test case would pass if the returned BMI value would be 18.22 or 18.25, however, it would fail for 18.3. Most unit testing frameworks provide a method to assert floating points within specific ranges. Testers should be careful when asserting results from floating point arithmetic because failures in those assertions can represent precision or range limitations in the programming language instead of faults in the source code, such as incorrect

calculations. For instance, neglecting to check for float precision is a “test smell” that can lead to flaky test executions [2, 3].⁹ If care is not taken some tests might fail when running them on a different computer or when the operating system has been updated.

In addition to asserting the valid behaviour of the UUT (also referred informally to as “happy paths”), unit tests should check the robustness of the implementation. For example, testers should examine how the class handles exceptional behaviour. There are different ways to design unit tests to handle exceptional behaviour, each with its trade-offs. One example is to use exception handling blocks and include *failing assertions* (e.g., `assert false`) in points past the code that triggers an exception. However, those methods are not effective in checking whether specific types of exceptions have been thrown, such as distinguishing between input/output exceptions for “file not found” or database connection errors versus exceptions thrown due to division by zero or accessing null variables. Those different types of exceptions represent distinct types of error handling situations that testers may choose to cover in their test suites. Therefore, many unit test frameworks have methods to assert whether the UUT raises specific types of exception. Here we use the `pytest.raises(...)` context manager to capture the exceptions thrown when trying to specify invalid values for height and check whether they are the exceptions that we expected, or whether there are unexpected exceptions. Additionally, testers can include assertions to verify whether the exception includes an expected message.

One of the challenges in writing good unit tests is deciding on the maximum size and scope of a single test case. For instance, in our `BMICalc` class, the `classifyBMI_teensAndChildren()` method has numerous branches to handle the various BMI thresholds for different age ranges. Creating a single test method that exercises all branches for all age ranges would lead to a very long test method with dozens of assertions. This test case would be hard to read and understand. Moreover, such a test case would hinder debugging efforts because the tester would need to narrow down which specific assertion detected a fault. Therefore, in order to keep our test methods small, we recommend breaking down the test coverage of the method (`classifyBMI_teensAndChildren()`) into a series of small test cases—with each test covering a different age range. In turn, for improved coverage, each of those test cases should assert all BMI classifications for the corresponding age bracket.

Testers should avoid creating redundant test cases in order to improve the cost-effectiveness of the unit testing process. Redundant tests exercise the same behaviour and do not bring any value (e.g., increased coverage) to the test suite. For instance, checking invalid height values in the `test_bmi_adult()` test case would introduce redundancy because those cases are already covered by the `test_invalid_height()` test case. On the other hand, the (`test_bmi_adult()`) test case currently does not attempt to invoke BMI for ages below 19. Therefore, we can improve our unit tests by adding this invocation to the existing test case, or—even better—creating a new method with that invocation (e.g., `test_bmi_adult_invalid()`).

⁹ Tests are considered flaky if their verdict (pass or fail) changes when no code changes are made. In other words, the tests seem to show random behaviour.

7.3.1 Supporting Unit Testing with AI

Conducting rigorous unit testing can be an expensive, effort-intensive task. The effort required to create a single unit test may be negligible over the full life of a project, but this effort adds up as the number of classes increases. If one wants to test thoroughly, they may end up creating hundreds to thousands of tests for a large-scale project. Selecting effective test input and creating detailed assertions for each of those test cases is not a trivial task either. The problem is not simply one of scale. Even if developers and testers have a lot of knowledge and good intentions, they might forget or not have the time needed to think of all important cases. They may also cover some cases more than others, e.g., they might focus on valid inputs, but miss important invalid or boundary cases. The effort spent by developers does not end with test creation. Maintaining test cases as the SUT evolves and deciding how to allocate test execution resources effectively—deciding *which* tests to execute—also require care, attention, and time from human testers.

Ultimately, developers often make compromises if they want to release their product on time and under a reasonable budget. This can be problematic, as insufficient testing can lead to critical failures in the field after the product is released. Automation has a critical role in controlling this cost, and ensuring that both sufficient quality and quantity of testing are achieved. AI techniques—including optimization, machine learning, natural language processing, and other approaches—can be used to partially automate and support aspects of unit test creation, maintenance, and execution. For example,

- Optimization and reinforcement learning can *select test input* suited to meeting measurable testing goals. This can be used to create either new test cases or to amplify the effectiveness of human-created test cases.
- The use of supervised and semi-supervised machine learning approaches has been investigated in order to *infer test oracles* from labelled executions of a system for use in judging the correctness of new system executions.
- Three different families of techniques, powered by optimization, supervised learning, and clustering techniques, are used to make effective use of computing resources when executing test cases:
 - *Test suite minimization* techniques suggest redundant test suites that could be removed or ignored during test execution.
 - *Test case prioritisation* techniques order test cases such that the potential for early fault detection or code coverage is maximised.
 - *Test case selection* techniques identify the subset of test cases that relate in some way to recent changes to the code, ignoring test cases with little connection to the changes being tested.

If aspects of unit testing—such as test creation or selection of a subset for execution—can be even partially automated, the benefit to developers could be immense. AI has been used to support these, and other, aspects of unit testing. In the

remainder of this chapter, we will focus on **test input generation**. In Sect. 7.5, we will also provide pointers to other areas of unit testing that can be partially automated using AI.

Exhaustively applying all possible input is infeasible due to an enormous number of possibilities for most real-world programs and units we need to test. Therefore, deciding *which* inputs to try becomes an important decision. Test generation techniques can create partial unit tests covering the initialisation, input, and tear down stages. The developer can then supply a test oracle or simply execute the generated tests and capture any crashes that occur or exceptions that are thrown. One of the more effective methods of automatically selecting effective test input is *search-based test generation*. We will explain this approach in the following sections.

A word of caution, before we continue—it is our firm stance that AI *cannot* replace human testers. The points above showcase a set of good practices for unit test design. Some of these practices may be more easily achieved by either a human or an intelligent algorithm. For instance, properties such as readability mainly depend on human comprehension. Choosing readable names or defining the ideal size and scope for test cases may be infeasible or difficult to achieve via automation. On the other hand, choosing inputs (values or method calls) that mitigate redundancy can be easily achieved through automation through instrumentation, e.g., the use of code coverage tools.

AI can make unit testing more cost-effective and productive when used to support human efforts. However, there are trade-offs involved when deciding how much to rely on AI versus the potential effort savings involved. AI cannot replace human effort and creativity. However, it can reduce human effort on repetitive tasks and can focus human testers towards elements of unit testing where their creativity can have the most impact. And over time, as AI-based methods become better and stronger, there is likely to be more areas of unit testing they can support or automate.

7.4 Search-Based Test Generation

Test input selection can naturally be seen as a search problem. When you create test cases, you often have one or more goals. Perhaps that goal is to find violations of a specification, to assess performance, to look for security vulnerabilities, to detect excessive battery usage, to achieve code coverage, or any number of other things that we may have in mind when we design test cases. We cannot try all input—any real-world piece of software with value has a near-infinite number of possible inputs we could try. However, somewhere in that space of possibilities lies a subset of inputs that best meets the goals we have in mind. Out of all of the test cases that could be generated for a UUT, we want to identify—systematically and at a reasonable cost—those that best meet those goals. Search-based test generation is an intuitive AI technique for locating those test cases that map to the same process we might use ourselves to find a solution to a problem.

Let us consider a situation where you are asked a question. If you do not know the answer, you might make a guess—either be an educated guess or one made

completely at random. In either case, you would then get some feedback. *How close were you to reaching the “correct” answer?* If your answer was not correct, you could then make a second guess. Your second guess, if nothing else, should be *closer* to being correct based on the knowledge gained from the feedback on that initial guess. If you are still not correct, you might then make a third, fourth, etc. guess—each time incorporating feedback on the previous guess.

Test input generation can be mapped to the same process. We start with a problem we want to solve. We have some goal that we want to achieve through the creation of unit tests. *If that goal can be measured*, then we can automate input generation. Fortunately, many testing goals can be measured.

- If we are interested in exploring the exceptions that the UUT can throw, then we want the inputs that *trigger the most exceptions*.
- If we are interested in covering all outcomes of a function, then we can divide the output into representative values and identify the inputs that *cover all representative output values*.
- If we are interested in executing all lines of code, then we are searching for the inputs that *cover more of the code structure*.
- If we are interested in executing a wide variety of input, then we want to find a set of inputs with *the highest diversity in their values*.

Attainment of many goals can be measured, whether as a percentage of a known checklist or just a count that we want to maximise. Even if we have a higher level goal in mind that cannot be directly measured, there may be measurable sub-goals that correlate with that higher level goal. For example, “find faults” cannot be measured—we do not know what faults are in our goal—but maximising code coverage or covering diverse outputs may increase the likelihood of detecting a fault.

Once we have a measurable **goal**, we can automate the guess-and-check process outlined above via a metaheuristic optimization algorithm. **Metaheuristics** are strategies to sample and evaluate values during our search. Given a measurable goal, a metaheuristic optimization algorithm can systematically sample the space of possible test input, guided by feedback from one or more **fitness functions**—numeric scoring functions that judge the optimality of the chosen input based on its attainment of our goals. The exact process taken to sample test inputs from that space varies from one metaheuristic to another. However, the core process can be generically described as:

1. Generate one or more initial **solutions** (test suites containing one or more unit tests).
2. While time remains:
 - (a) Evaluate each solution using the **fitness functions**.
 - (b) Use feedback from the fitness functions and the sampling strategy employed by the **metaheuristic** to improve the solutions.
3. Return the best solution seen during this process.

In other words, we have an optimization problem. We make a guess, get feedback, and then use that additional knowledge to make a *smarter* guess. We keep going until we run out of time, then we work with the best solution we found during that process.

The choice of both metaheuristic and fitness functions is crucial to successfully deploying search-based test generation. Given the existence of a near-infinite space of possible input choices, the order that solutions are tried from that space is the key to efficiently finding a solution. The metaheuristic—guided by feedback from the fitness functions—overcomes the shortcomings of a purely random input selection process by using a deliberate strategy to sample from the input space, gravitating towards “good” input and discarding input sharing properties with previously-seen “bad” solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic impacts the quality and efficiency of the search process. Metaheuristics are often inspired by natural phenomena, such as swarm behaviour or evolution within an ecosystem.

In search-based test generation, the fitness functions represent our goals and guide the search. They are responsible for evaluating the quality of a solution and offering feedback on how to improve the proposed solutions. Through this guidance, the fitness functions shape the resulting solutions and have a major impact on the quality of those solutions. Functions must be efficient to execute, as they will be calculated thousands of times over a search. Yet, they also must provide enough detail to differentiate candidate solutions and guide the selection of optimal candidates.

Search-based test generation is a powerful approach because it is *scalable* and *flexible*. Metaheuristic search—by strategically sampling from the input space—can scale to larger problems than many other generation algorithms. Even if the “best” solution cannot be found within the time limit, search-based approaches typically can return a “good enough” solution. Many goals can be mapped to fitness functions, and search-based approaches have been applied to a wide variety of testing goals and scenarios. Search-based generation often can even achieve higher goal attainment than developer-created tests.

In the following sections, we will explain the highlighted concepts in more detail and explore how they can be applied to generate partial unit tests for Python programs. In Sect. 7.4.1, we will explain how to represent solutions. Then, in Sect. 7.4.2, we will explore how to represent two common goals as fitness functions. In Sect. 7.4.3, we will explain how to use the solution representation and fitness functions as part of two common metaheuristic algorithms. Finally, in Sect. 7.4.4, we will illustrate the application of this process on our BMI example.

7.4.1 *Solution Representation*

When solving any problem, we first must define the form the solution to the problem must take. What, exactly, does a solution to a problem “look” like? What are its contents? How can it be manipulated? Answering these questions is crucial before we can define how to identify the “best” solution.

In this case, we are interested in identifying a set of unit tests that maximise attainment of a testing goal. This means that a solution is a **test suite**—a collection of test cases. We can start from this decision, and break it down into the composite elements relevant to our problem.

- A solution is a **test suite**.
- A test suite contains one or more **test cases**, expressed as individual methods of a single test class.
- The solution interacts with a **unit-under-test (UUT)** which is a single, identified Python class with a constructor (optional) and one or more methods.
- Each test case contains an **initialisation** of the UUT which is a call to its constructor, if it has one.
- Each test case then contains one or more **actions**, i.e., calls to one of the methods of the UUT or assignments to a class variable.
- The initialisation and each action have zero or more **parameters** (input) supplied to that action.

This means that we can think of a test suite as a collection of test cases, and each test case as a single initialisation and a collection of actions, with associated parameters. When we generate a solution, we choose a number of test cases to create. For each of those test cases, we choose a number of actions to generate. Different solutions can differ in size—they can have differing numbers of test cases—and each test case can differ in size—each can contain a differing number of actions.

In search-based test generation, we represent two solutions in two different forms:

- **Phenotype (External) Representation:** The phenotype is the version of the solution that will be presented to an external audience. This is typically in a human-readable form or a form needed for further processing.
- **Genotype (Internal) Representation:** The genotype is a representation used internally, within the metaheuristic algorithm. This version includes the properties of the solution that are relevant to the search algorithm, e.g., the elements that can be manipulated directly. It is generally a minimal representation that can be easily manipulated by a program.

Figure 7.5 illustrates the two representations of a solution that we have employed for unit test generation in Python. The phenotype representation takes the form of an executable pytest test class. In turn, each test case is a method containing an initialisation, followed by a series of method calls or assignments to class variables. This solution contains a single test case, `test_0()`. It begins with a call to the constructor of the UUT, `BMICalc`, supplying a height of 246, a weight of 680, and an age of 2. It then applies a series of actions on the UUT: setting the age to 18, getting a BMI classification from `classify_bmi_teens_and_children()`, setting the weight to 466, getting further classifications from each method, setting the weight to 26, then getting one last classification from `classify_bmi_adults()`.

This is our desired external representation because it can be executed at will by a human tester, and it is in a format that a tester can read. However, this representation is not ideal for use by the metaheuristic search algorithm as it cannot be easily

```

1   import pytest
2   import bmi_calculator
3
4   def test_0():
5       cut = bmi_calculator.BMICalc(246, 680, 2)
6       cut.age = 18
7       cut.classify_bmi_teens_and_children()
8       cut.weight = 466
9       cut.classify_bmi_adults()
10      cut.classify_bmi_teens_and_children()
11      cut.weight = 26
12      cut.classify_bmi_adults()
13

```

Fig. 7.5 The genotype (internal, left) and phenotype (external, right) representations of a solution containing a single test case. Each identifier in the genotype is mapped to a function with a corresponding list of parameters. For instance, 1 maps to setting the weight, and 5 maps to calling the method `classify_bmi_adults()`

manipulated. If we wanted to change one method call to another, we would have to identify which methods were being called. If we wanted to change the value assigned to a variable, we would have to identify (a) which variable was being assigned a value, (b) identify the portion of the line that represents the value, and (c), change that value to another. Internally, we require a representation that can be manipulated quickly and easily.

This is where the genotype representation is required. In this representation, a test suite is a `list` of test cases. If we want to add a test case, we can simply append it to the list. If we want to access or delete an existing test case, we can simply select an index from the list. Each test case is a `list` of actions. Similarly, we can simply refer to the index of an action of interest.

Within this representation, each action is a `list` containing (a) an action identifier, and (b), a `list` of parameters to that action (or an empty `list` if there are no parameters). The action identifier is linked to a separate list of actions that the tester supplies, that stores the method or variable name and type of action, i.e., assignment or method call (we will discuss this further in Sect. 7.4.4). An identifier of `-1` is reserved for the constructor.

The solution illustrated in Fig. 7.5 is not a particularly effective one. It consists of a single test case that applies seemingly random values to the class variables (the initial constructor creates what may be the world's largest two-year old). This solution only covers a small set of BMI classifications and only a tiny portion of the branching behaviour of the UUT. However, one could imagine this as a starting solution that could be manipulated over time into a set of highly effective test cases. By making adjustments to the genotype representation, guided by the score from a fitness function, we can introduce those improvements.

7.4.2 Fitness Function

As previously-mentioned, fitness functions are the cornerstone of search-based test generation. The core concept is simple and flexible—a fitness function is simply a function that takes in a solution candidate and returns a “score” describing the quality of that solution. This gives us the means to differentiate one solution from another, and more importantly, to tell if one solution is *better* than another.

Fitness functions are meant to embody the goals of the tester. They tell us how close a test suite came to meeting those goals. The fitness functions employed determine what properties the final solution produced by the algorithm will have, and shape the evolution of those solutions by providing a target for optimization.

Essentially any function can serve as a fitness function, as long as it returns a numeric score. It is common to use a function that emits either a percentage (e.g., percentage of a checklist completed) or a raw number as a score, then either maximise or minimise that score.

- A fitness function should *not* return a Boolean value. This offers almost no feedback to improve the solution, and the desired outcome may not be located.
- A fitness function should yield (largely) continuous scores. A small change in a solution should not cause a large change (either positive or negative) in the resulting score. Continuity in the scoring offers clearer feedback to the metaheuristic algorithm.
- The best fitness functions offer not just an indication of quality, but a *distance* to the optimal quality. For example, rather than measuring completion of a checklist of items, we might offer some indication of how close a solution came to completing the remaining items on that checklist. In this chapter, we use a simple fitness function to clearly illustrate search-based test generation, but in Sect. 7.5, we will introduce a distance-based version of that fitness function.

Depending on the algorithm employed, either a single fitness function or multiple fitness functions can be optimised at once. We focus on single-function optimization in this chapter, but in Sect. 7.5, we will also briefly explain how multi-objective optimization is achieved.

To introduce the concept of a fitness function, we utilise a fitness function based on the **code coverage** attained by the test suite. When testing, developers must judge: (a) whether the produced tests are effective and (b) when they can stop writing additional tests. Coverage criteria provide developers with guidance on both of those elements. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, these criteria are intended to serve as an approximation of efficacy. If the goals of the chosen criterion are met, then we have put in a measurable testing effort and can decide whether we have tested enough.

There are many coverage criteria, with varying levels of tool support. The most common criteria measure coverage of structural elements of the software, such as individual statements, branches of the software’s control flow, and complex Boolean

```

1 def calculate_fitness(metadata, fitness_function, num_tests_penalty,
2                      length_test_penalty, solution):
3     fitness = 0.0
4
5     # Get the statement coverage over the code
6     fitness += statement_fitness(metadata, solution)
7
8     # Add a penalty to control test suite size
9     fitness -= float(len(solution.test_suite) / num_tests_penalty)
10
11    # Add a penalty to control the length of individual test cases
12    # Get the average test suite length
13    total_length = 0
14    total_length = sum([len(test) for test in solution.test_suite]) / len(
15                           solution.test_suite)
16    fitness -= float(total_length / length_test_penalty)
17
18    solution.fitness = fitness

```

Fig. 7.6 The high-level calculation of the fitness function

conditional statements. One of the most common, and most intuitive, coverage criteria is **statement coverage**. It simply measures the percentage of executable lines of code that have been triggered at least once by a test suite. The more of the code we have triggered, the more thorough our testing efforts are—and, ideally, the likely we will be to discover a fault. The use of statement coverage as a fitness function encourages the metaheuristic to explore the structure of the source code, reaching deeply into branching elements of that code.

As we are already generating `pytest`-compatible test suites, measuring statement coverage is simple. The `pytest` plugin `pytest-cov` measures statement coverage, as well as branch coverage—a measurement of how many branching control points in the UUT (e.g., if-statement and loop outcomes) have been executed—as part of executing a `pytest` test class. By making use of this plug-in, statement coverage of a solution can be measured as follows:

1. Write the phenotype representation of the test suite to a file.
2. Execute `pytest`, with the `-cov = < python file to measure coverage over >` command.
3. Parse the output of this execution, extracting the percentage of coverage attained.
4. Return that value as the fitness.

This measurement yields a value between 0 and 100, indicating the percentage of statements executed by the solution. We seek to *maximise* the statement coverage. Therefore, we employ the formulation in Eq. 7.2 to obtain the fitness value of a test suite (shown as code in Fig. 7.6).

$$\text{fitness}(\text{solution}) = \text{statement_coverage}(\text{solution}) - \text{bloat_penalty}(\text{solution}) \quad (7.2)$$

The *bloat penalty* is a small penalty to the score intended to control the size of the produced solution in two dimensions: the number of test methods, and the number of actions in each test. A massive test suite may attain high code coverage or yield many different outcomes, but it is likely to contain many redundant elements as well. In addition, it will be more difficult to understand when read by a human. In particular, long sequences of actions may hinder efforts to debug the code and identify a fault. Therefore, we use the bloat penalty to encourage the metaheuristic algorithm to produce *small-but-effective* test suites. The bloat penalty is calculated as in Eq. 7.3.

$$\begin{aligned} \text{bloat_penalty}(\text{solution}) = & (\text{num_test_cases}/\text{num_tests_penalty}) \\ & + (\text{average_test_length}/\text{length_test_penalty}) \end{aligned} \quad (7.3)$$

where *num_tests_penalty* is 10 and *length_test_penalty* is 30. That is, we divide the number of test cases by 10 and the average length of a single test case (number of actions) by 30. These weights could be adjusted, depending on the severity of the penalty that the tester wishes to apply. It is important to not penalise too heavily, as that will increase the difficulty of the core optimization task—some expansion in the number of tests or length of a test is needed to cover the branching structure of the code. These penalty values allow some exploration while still encouraging the metaheuristic to locate smaller solutions.

7.4.3 Metaheuristic Algorithms

Given a solution representation and a fitness function to measure the quality of solutions, the next step is to design an algorithm capable of producing the best possible solution within the available resources. Any UUT with reasonable complexity has a near-infinite number of possible test inputs that could be applied. We cannot reasonable try them all. Therefore, the role of the metaheuristic is to intelligently sample from that space of possible inputs in order to locate the best solution possible within a strict time limit.

There are many metaheuristic algorithms, each making use of different mechanisms to sample from that space. In this chapter, we present two algorithms:

- **Hill Climber:** A simple algorithm that produces a random initial solution, then attempts to find better solutions by making small changes to that solution—restarting if no better solution can be found.
- **Genetic Algorithm:** A more complex algorithm that models how populations of solutions evolve over time through the introduction of mutations and through the breeding of good solutions.

The Hill Climber is simple, fast, and easy to understand. However, its effectiveness depends strongly on the quality of the initial guess made. We introduce it first to explain core concepts that are built upon by the Genetic Algorithm, which is slower but potentially more robust.

7.4.3.1 Common Elements

Before introducing either algorithm in detail, we will begin by discussing three elements shared by both algorithms—a metadata file that defines the actions available for the UUT, random test generation, and the search budget.

UUT Metadata File: To generate unit tests, the metaheuristic needs to know *how* to interact with the UUT. In particular, it needs to know what methods and class variables are available to interact with, and what the parameters of the methods and constructor are. To provide this information, we define a simple JSON-formatted metadata file. The metadata file for the BMI example is shown in Fig. 7.7, and we define the fields of the file as follows:

- **file:** The python file containing the UUT.
- **location:** The path of the file.
- **class:** The name of the UUT.
- **constructor:** Contains information on the parameters of the constructor.
- **actions:** Contains information about each action.
 - **name:** The name of the action (method or variable name).
 - **type:** The type of action (`method` or `assign`).
 - **parameters:** Information about each parameter of the action.

```

1  {
2      "file": "bmi_calculator",
3      "location": "example/",
4      "class": "BMICalc",
5      "constructor": {
6          "parameters": [
7              { "type": "integer", "min": -1 },
8              { "type": "integer", "min": -1 },
9              { "type": "integer", "min": -1, "max": 150 }
10         ],
11         "actions": [
12             { "name": "height", "type": "assign", "parameters": [
13                 { "type": "integer", "min": -1 } ]
14             },
15             { "name": "weight", "type": "assign", "parameters": [
16                 { "type": "integer", "min": -1 } ]
17             },
18             { "name": "age", "type": "assign", "parameters": [
19                 { "type": "integer", "min": -1, "max": 150 } ]
20             },
21             { "name": "bmi_value", "type": "method" },
22             { "name": "classify_bmi_teens_and_children", "type": "method" },
23             { "name": "classify_bmi_adults", "type": "method" }
24         ]
25     }
}

```

Fig. 7.7 Metadata definition for class `BMICalc`

type: Datatype of the parameter. For this example, we only support integer input. However, the example code could be expanded to handle additional datatypes.

min: An optional minimum value for the parameter. Used to constrain inputs to a defined range.

max: An optional maximum value for the parameter. Used to constrain inputs to a defined range.

This file not only tells the metaheuristic what actions are available for the UUT, it suggests a starting point for “how” to test the UUT by allowing the user to optionally constrain the range of values. This allows more effective test generation by limiting the range of guesses that can be made to “useful” values. For example, the age of a person cannot be a negative value in the real world, and it is unrealistic that a person would be more than 150 years old. Therefore, we can impose a range of age values that we might try. To test error handling for negative ranges, we might set the minimum value to -1 . This allows the metaheuristic to try a negative value, while preventing it from wasting time trying *many* negative values.

In this example, we assume that a tester would create this metadata file—a task that would take only a few minutes for a UUT. However, it would be possible to write code to extract this information as well.

Random Test Generation: Both of the presented metaheuristic algorithms start by making random “guesses”—either generating random test cases or generating entire test suites at random—and will occasionally modify solutions through random generation of additional elements. To control the size of the generated test suites or test cases, there are two user-controllable parameters:

- **Maximum number of test cases:** The largest test suite that can be randomly generated. When a suite is generated, a size is chosen between $1 - \text{max_test_cases}$, and that the number of test cases is generated and added to the suite.
- **Maximum number of actions:** The largest individual test case that can be randomly generated. When a test case is generated, a number of actions between $1 - \text{max_actions}$ is chosen and many actions are added to the test case (following a constructor call).

By default, we use 20 as the value for both parameters. This provides a reasonable starting point for covering a range of interesting behaviours, while preventing test suites from growing large enough to hinder debugging. Test suites can then grow or shrink over time through manipulation by the metaheuristic.

Search Budget: This search budget is the time allocated to the metaheuristic. The goal of the metaheuristic is to find the best solution possible within this limitation. This parameter is also user-controlled:

- **Search Budget:** The maximum number of generations of work that can be completed before returning the best solution found.

The search budget is expressed as a number of *generations*—cycles of exploration of the search space of test inputs—that are allocated to the algorithm. This can be

set according to the schedule of the tester. By default, we allow 200 generations in this example. However, fewer may still produce acceptable results, while more can be allocated if the tester is not happy with what is returned in that time frame.

7.4.3.2 Hill Climber

A Hill Climber is a classic metaheuristic that embodies the “guess-and-check” process we discussed earlier. The algorithm makes an initial guess purely at random, then attempts to improve that guess by making small, iterative changes to it. When it lands on a guess that is better than the last one, it adopts it as the current solution and proceeds to make small changes to that solution. The core body of this algorithm is shown in Fig. 7.8. The full code can be found at https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/hill_climber.py.

The variable `solution_current` stores the current solution. At first, it is initialised to a random test suite, and we measure the fitness of the solution (lines 2–5). Following this, we start our first generation of evolution. While we have the remaining search budget, we then attempt to improve the current solution.

Each generation, we attempt to improve the current solution through the process of *mutation*. During mutation, we introduce a small change to the current solution. In Fig. 7.9, we outline the types of change possible during mutation:

After selecting and applying one of these transformations (line 22), we measure the fitness of the mutated solution (line 23). If it is better than the current solution, we make the mutation into the current solution (lines 26–28). If it is better than the best solution seen to date, we also save it as the new best solution (lines 30–31). We then proceed to the next generation.

If the mutation is not better than the current solution, we try a different mutation to see if it is better. The range of transformations results in a very large number of possible transformations. However, even with such a range, we may end up in situations where no improvement is possible, or where it would be prohibitively slow to locate an improved solution. We refer to these situations as *local optima*—solutions that, while they may not be the best possible, are the best that can be located through incremental changes.

We can think of the landscape of possible solutions as a topographical map, where better fitness scores represent higher levels of elevation in the landscape. This algorithm is called a “Hill Climber” because it attempts to scale that landscape, finding the tallest peak that it can in its local neighbourhood.

If we reach a local optima, we need to move to a new “neighborhood” in order to find taller peaks to ascend. In other words, when we become stuck, we restart by replacing the current solution with a new random solution (lines 37–42). Throughout this process, we track the best solution seen to date to return at the end. To control this process, we use two user-controllable parameters.

```

1 # Generate an initial random solution, and calculate its fitness
2 solution_current = Solution()
3 solution_current.test_suite = generate_test_suite(metadata, max_test_cases, max_actions)
4 calculate_fitness(metadata, fitness_function, num_tests_penalty,
5                   length_test_penalty, solution_current)
6
7 # The initial solution is the best we have seen to date
8 solution_best = copy.deepcopy(solution_current)
9
10 # Continue to evolve until the generation budget is exhausted
11 # or the number of restarts is exhausted.
12 gen = 1
13 restarts = 0
14
15 while gen <= max_gen and restarts <= max_restarts:
16     tries = 1
17     changed = False
18
19     # Try random mutations until we see a better solutions,
20     # or until we exhaust the number of tries.
21     while tries < max_tries and not changed:
22         solution_new = mutate(solution_current)
23         calculate_fitness(metadata, fitness_function, num_tests_penalty,
24                           length_test_penalty, solution_new)
25
26         # If the solution is an improvement, make it the new solution.
27         if solution_new.fitness > solution_current.fitness:
28             solution_current = copy.deepcopy(solution_new)
29             changed = True
30
31         # If it is the best solution seen so far, then store it.
32         if solution_new.fitness > solution_best.fitness:
33             solution_best = copy.deepcopy(solution_current)
34
35     tries += 1
36
37     # Reset the search if no better mutant is found within a set number
38     # of attempts by generating a new solution at random.
39     if not changed:
40         restarts += 1
41         solution_current = Solution()
42         solution_current.test_suite = generate_test_suite(metadata, max_test_cases,
43                                               max_actions)
44         calculate_fitness(metadata, fitness_function, num_tests_penalty,
45                           length_test_penalty, solution_current)
46
47     # Increment generation
48     gen += 1
49
50 # Return the best suite seen

```

Fig. 7.8 The core body of the Hill Climber algorithm

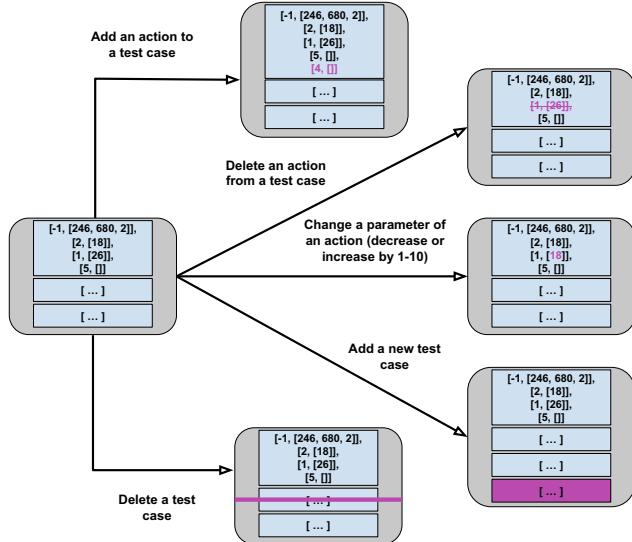


Fig. 7.9 Example mutation types

- **Maximum Number of Tries:** A limit on the number of mutations we are willing to try before restarting (`max_tries`, line 21). By default, this is set to 200.
- **Maximum Number of Restarts:** A limit of restarts we are willing to try before giving up on the search (`max_restarts`, line 15). Be default, this is set to 5.

The core process employed by the Hill Climber is simple but effective. Hill Climbers also tend to be faster than many other meta-heuristics. This makes them a popular starting point for search-based automation. Their primary weakness is their reliance on making a good initial guess. A bad initial guess could result in time wasted exploring a relatively “flat” neighbourhood in that search landscape. Restarts are essential to overcoming that limitation.

7.4.3.3 Genetic Algorithm

Genetic Algorithms model the evolution of a population over time. In a population, certain individuals may be “fitter” than others, possessing traits that lead them to thrive—traits that we would like to see passed forward to the next generation through reproduction with other fit individuals. Over time, random mutations introduced into the population may also introduce advantages that are also passed forward to the next generation. Over time, through mutation and reproduction, the overall population will grow stronger and stronger.

As a metaheuristic, a Genetic Algorithm is build on a core generation-based loop like the Hill Climber. However, there are two primary differences:

- Rather than evolving a single solution, we simultaneously manage a *population* of different solutions.
- In addition to using mutation to improve solutions, a Genetic Algorithm also makes use of a *selection* process to identify the best individuals in a population, and a *crossover* process that produces new solutions merging the test cases (“genes”) of parent solutions (“chromosomes”).

The core body of the Genetic Algorithm is listed in Fig. 7.10. The full code can be found at https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/genetic_algorithm.py.

We start by creating an initial population, where each member of the population is a randomly generated test suite (line 1). We initialise the best solution to the first member of that population (line 5). We then begin the first generation of evolution (line 12).

Each generation, we form a new population by applying a series of actions intended to promote the best “genes” forward. We form the new population by creating two new solutions at a time (line 16). First, we attempt to identify two of the best solutions in a population. If the population is large, this can be an expensive process. To reduce this cost, we perform a *selection* procedure on a randomly chosen subset of the population (lines 19–20), explained in Fig. 7.11.

The fitness of the members of the chosen subset is compared in a process called “tournament”, and a winner is selected. The winner may not be the best member of the full population but will be at least somewhat effective, and will be identified at a lower cost than comparing all population members. These two solutions may be carried forward as-is. However, at certain probabilities, we may make further modifications to the chosen solutions.

The first of these is crossover—a transformation that models reproduction. We generate a random number and check whether it is less than a user-set `crossover_probability` (line 23). If so, we combine individual genes (test cases) of the two solutions using the process in Fig. 7.12.

If the parents do not contain the same number of test cases, then the remaining cases can be randomly distributed between the children. This form of crossover is known as “uniform crossover”. There are other means of performing crossover. For example, in “single-point” crossover, a single index is chosen, and one child gets all elements from Parent A before that index, and all elements from Parent B after that index (with the other child getting the reverse). Another form, “discrete recombination”, is similar to uniform crossover, except that we make the coin flip for each child instead of once for both children at each index.

We may introduce further mutations to zero, one, or both of the solutions. If a random number is less than a user-set `mutation probability` (lines 27, 29), we will introduce a single mutation to that solution. We do this independently for both solutions. The mutation process is the same as in the Hill Climber, where we can add, delete, or modify an individual action or add or delete a full test case.

```
1 #Create initial population.
2 population = create_population(population_size)
3
4 #Initialise best solution as the first member of that population.
5 solution_best = copy.deepcopy(population[0])
6
7 # Continue to evolve until the generation budget is exhausted.
8 # Stop if no improvement has been seen in some time (stagnation).
9 gen = 1
10 stagnation = -1
11
12 while gen <= max_gen and stagnation <= exhaustion:
13     # Form a new population.
14     new_population = []
15
16     while len(new_population) < len(population):
17         # Choose a subset of the population and identify
18         # the best solution in that subset (selection).
19         offspring1 = selection(population, tournament_size)
20         offspring2 = selection(population, tournament_size)
21
22         # Create new children by breeding elements of the best solutions
23         # (crossover).
24         if random.random() < crossover_probability:
25             (offspring1, offspring2) = uniform_crossover(offspring1, offspring2)
26
27         # Introduce a small, random change to the population (mutation).
28         if random.random() < mutation_probability:
29             offspring1 = mutate(offspring1)
30         if random.random() < mutation_probability:
31             offspring2 = mutate(offspring2)
32
33         # Add the new members to the population.
34         new_population.append(offspring1)
35         new_population.append(offspring2)
36
37         # If either offspring is better than the best-seen solution,
38         # make it the new best.
39         if offspring1.fitness > solution_best.fitness:
40             solution_best = copy.deepcopy(offspring1)
41             stagnation = -1
42         if offspring2.fitness > solution_best.fitness:
43             solution_best = copy.deepcopy(offspring2)
44             stagnation = -1
45
46         # Set the new population as the current population.
47         population = new_population
48
49         # Increment the generation.
50         gen += 1
51         stagnation += 1
52
53     # Return the best suite seen
```

Fig. 7.10 The core body of the genetic algorithm

Fig. 7.11 Selecting the best solution on a subset of the population

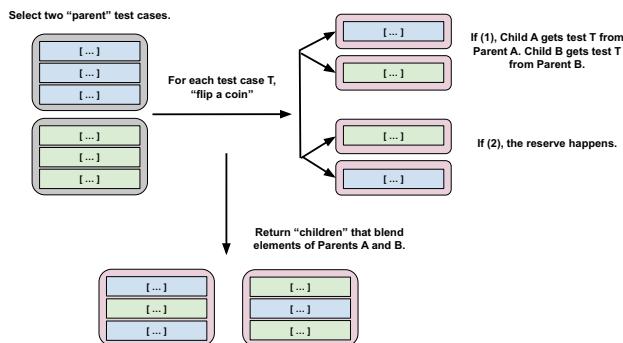
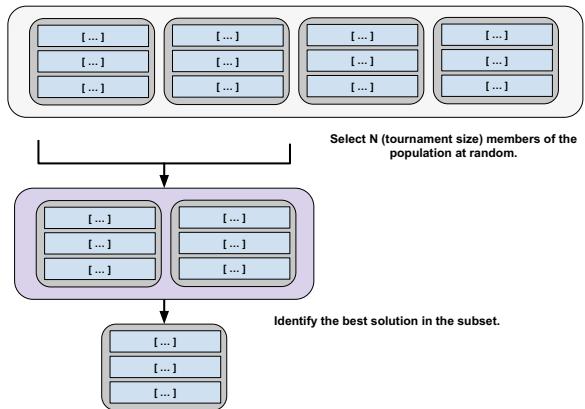


Fig. 7.12 Crossover process

Finally, we add both of the solutions to the new population (line 46). If either solution is better than the best seen to date, we save it to be returned at the end of the process (lines 38–43). Once the new population is complete, we continue to the next generation.

There may be a finite amount of improvement that we can see in a population before it becomes *stagnant*. If the population cannot be improved further, we may wish to terminate early and not waste computational effort. To enable this, we count the number of generations where no improvement has been seen (line 50) and terminate if it passes a user-set *exhaustion* threshold (line 12). If we identify a new “best” solution, we reset this counter (lines 40, 43).

The following parameters of the genetic algorithm can be adjusted:

- **Population Size:** The size of the population of solutions. By default, we set this to 20. This size must be an even number in the example implementation.
- **Tournament Size:** The size of the random population subset compared to identify the fittest population members. By default, this is set to 6.

- **Crossover Probability:** The probability that we apply crossover to generate child solutions. By default, 0.7.
- **Mutation Probability:** The probability that we apply mutation to manipulate a solution. By default, 0.7.
- **Exhaustion Threshold:** The number of generations of stagnation allowed before the search is terminated early. By default, we have set this to 30 generations.

These parameters can have a noticeable effect on the quality of the solutions located. Getting the best solutions quickly may require some experimentation. However, even at default values, this can be a highly effective method of generating test suites.

7.4.4 Examining the Resulting Test Suites

Now that we have all of the required components in place, we can generate test suites and examine the results. To illustrate what these results look like, we will examine test suites generated after executing the Genetic Algorithm for 1000 generations. During these executions, we disabled the exhaustion threshold to see what would happen if the algorithm was given the full search budget to work.

Figure 7.13 illustrates the results of executing the Genetic Algorithm. We can see the change in fitness over time, as well as the change in the number of test cases in the suite and the average number of actions in test cases. Note that fitness is penalised by the bloat penalty, so the actual statement coverage is higher than the final fitness

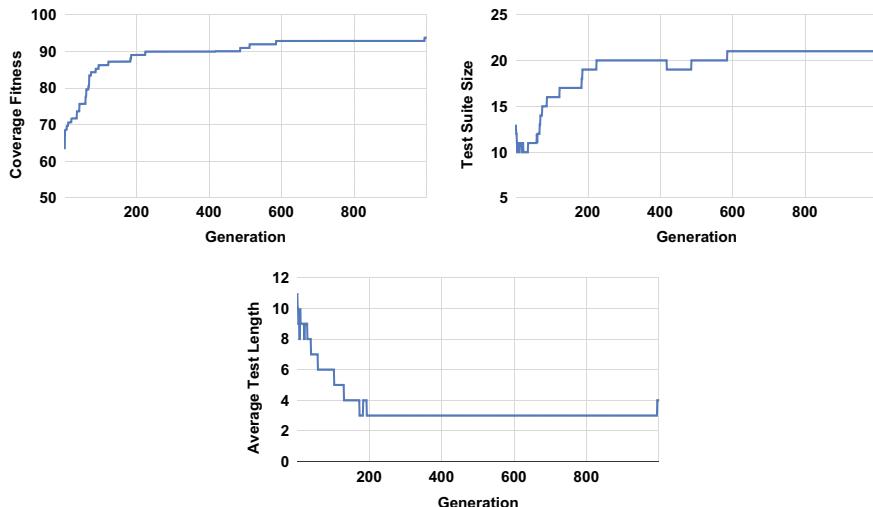


Fig. 7.13 Change in fitness, test suite size, and average number of actions in a test case over 1000 generations. Note that fitness includes both coverage and bloat penalty, and can never reach 100

value. Also note that metaheuristic search algorithms are random. Therefore, each execution of the Hill Climber or Genetic Algorithm will yield different test suites in the end. Multiple executions may be desired in order to detect additional crashes or other issues.

The fitness starts around 63, but quickly climbs until around generation 100, when it hits approximately 86. There are further gains after that point, but progress is slow. At generation 717, it hits a fitness value of 92.79, where it remains until near the very end of the execution. At generation 995, a small improvement is found that leads to the coverage of additional code and a fitness increase to 93.67. Keep in mind, again, that a fitness of “100” is not possible due to the bloat penalty. It is possible that further gains in fitness could be attained with an even higher search budget, but covering the final statements in the code and further trimming the number or length of test cases both become quite difficult at this stage.

The test suite size starts at 13 tests, then sheds excess tests for a quick gain in fitness. However, after that, the number of tests rises slowly as coverage increases. For much of the search, the test suite remains around 20 test cases, then 21. At the end, the final suite has 22 test cases. In general, it seems that additional code coverage is attained by generating new tests and adding them to the suite.

At times, redundant test cases are removed, but instead, we often see redundancy removed through the deletion of actions within individual test cases. The initial test cases are often quite long, with many redundant function calls. Initially, tests have an average of 11 actions. Initially, the number of actions oscillates quite a bit between an average of 8–10 actions. However, over time, the redundant actions are trimmed from test cases. After generation 200, test cases have an average of only three actions until generation 995, when the new test case increases the average length to four actions. With additional time, it is likely that this would shrink back to three. We see that the tendency is to produce a large number of very small test cases. This is good, as short test cases are often easier to understand and make it easier to debug the code to find faults.

More complex fitness functions or algorithms may be able to cover more code, or cover the same code more quickly, but these results show the power of even simple algorithms to generate small, effective test cases. A subset of a final test suite is shown in Fig. 7.14.¹⁰

Some test cases look like `test_0()` and `test_11()` in the example—a constructor call, followed by a BMI classification. Others will adjust the variable assignments, then make calls. For example, `test_5()` covers several paths in the code by making assignments, then getting classifications, multiple times. `test_7()` is an example of one where only a constructor call was needed, as the value supplied—a negative weight, in this case—was sufficient to trigger an exception.

There is still some room for improvement in these test cases. For example, `test_2()` and `test_17()` both contain redundant calls to a classification method. It is likely that a longer search budget would remove these calls. It would be simple

¹⁰ The full suite can be found at https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/example/test_bmi_calculator_automated_statement.py.

```

1 def test_0():
2     cut = bmi_calculator.BMICalc(120, 860, 13)
3     cut.classify_bmi_teens_and_children()
4
5 def test_2():
6     cut = bmi_calculator.BMICalc(43, 243, 59)
7     cut.classify_bmi_adults()
8     cut.height = 526
9     cut.classify_bmi_adults()
10    cut.classify_bmi_adults()
11
12 def test_5():
13     cut = bmi_calculator.BMICalc(374, 343, 17)
14     cut.age = 123
15     cut.classify_bmi_adults()
16     cut.age = 18
17     cut.classify_bmi_teens_and_children()
18     cut.weight = 396
19     cut.classify_bmi_teens_and_children()
20
21 def test_7():
22     cut = bmi_calculator.BMICalc(609, -1, 94)
23
24 def test_11():
25     cut = bmi_calculator.BMICalc(491, 712, 20)
26     cut.classify_bmi_adults()
27
28 def test_17():
29     cut = bmi_calculator.BMICalc(608, 717, 6)
30     cut.classify_bmi_teens_and_children()
31     cut.age = 91
32     cut.classify_bmi_teens_and_children()
33     cut.classify_bmi_teens_and_children()
34

```

Fig. 7.14 A subset of the test suite produced by the genetic algorithm, targeting statement coverage

to simply remove all cases where a method is called twice in a row with the same arguments from the suite. However, in other cases, those calls may have different results (e.g., if class state was modified by the calls), and you would want to leave them in place.

Search-based test generation requires a bit of experimentation. Even the Hill Climber has multiple user-selectable parameters. Finding the right search budget, for example, can require some experimentation. It may be worth executing the algorithm once with a very high search budget in order to get an initial idea of the growth in fitness. In this case, a tester could choose to stop much earlier than 1000 generations with little loss in effectiveness. For example, only limited gains are seen after 200 generations, and almost no gain in fitness is seen after 600 generations.

7.4.5 Assertions

It is important to note that this chapter is focused on *test input* generation. These test cases lack assertion statements, which are needed to check the correctness of the program behaviour.

These test cases *can* be used as-is. Any exceptions thrown by the UUT, or other crashes detected, when the tests execute will be reported as failures. In some cases, exceptions *should* be thrown. In Fig. 7.14, `test_17` will trigger an exception when `classify_bmi_teens_and_children()` is called for a 91-year-old. This exception is the desired behaviour. However, in many cases, exceptions are not desired, and these test cases can be used to alert the developer about crash-causing faults.

Otherwise, the generated tests will need assertions to be added. A tester can add assertions manually to these test cases, or a subset of them, to detect incorrect output. Otherwise, researchers and developers have begun to explore the use of AI techniques to generate assertions as well. We will offer pointers to some of this work in Sect. 7.5.

7.5 Advanced Concepts

The input generation technique introduced in the previous section can be used to generate small, effective test cases for Python classes. This section briefly introduces concepts that build on this foundation and offers pointers for readers interested in developing more complex AI-enhanced unit test automation tools.

7.5.1 Distance-Based Coverage Fitness Function

This chapter introduced the idea that we can target the maximisation of code coverage as a fitness function. We focused on statement coverage—a measurement of the number of lines of code executed. A similar measurement is the *branch coverage*—a measurement of the number of outcomes of control-altering expressions covered by test cases. This criterion is expressed over statements that determine which code will be executed next in the sequence. For example, in Python, this includes `if`, `for`, and `while` statements. Full branch coverage requires `True` and `False` outcomes for the Boolean predicates expressed in each statement.

Branch coverage can be maximised in the same manner that we maximised statement coverage—by simply measuring the attained coverage and favouring higher totals. However, advanced search-based input generation techniques typically use a slightly more complex fitness function based on *how close* a test suite came to covering each of these desired outcomes.

Let's say that we had two test suites—one that attains 50% branch coverage and one that attains 75% coverage. We would favour the one with 75% coverage, of course. However, what if both had 75% coverage? Which is better? The answer is that we want the one that is *closer* to covering the remaining 25%. Perhaps, with only small changes, that one could attain 100% coverage. We cannot know which of those two is better with our simple measurement of coverage. Rather, to make that determination, we divide branch coverage into a set of *goals*, or combinations of an expression we want to reach and an outcome we desire for that expression. Then, for each goal, we measure the *branch distance* as a score ranging from 0 – 1. The branch distance is defined as in Eq. 7.4.

$$distance(goal, suite) = \begin{cases} 0 & \text{If the branch is reached and the desired outcome is attained.} \\ distance_{min}(goal, suite) & \text{If the branch is reached, but the desired outcome is not attained.} \\ 1 & \text{If the branch has not been reached.} \end{cases} \quad (7.4)$$

Our goal is to *minimise* the branch distance. If we have *reached* the branch of interest and *attained* the desired outcome, then the score is 0. If we have not reached the branch, then the value is 1. If we have reached the branch, but not covered it, then we measure *how close* we came by transforming the Boolean predicate into a numeric function. For example, if we had the expression `if x == 5:` and desired a `True` outcome, but `x` was assigned a value of 3 when we executed the expression, we would calculate the branch distance as $abs(x - 5) = abs(3 - 5) = 2$.¹¹

We then normalise this value to be between 0 and 1. As this expression may be executed multiple times by the test suite, we take the minimum branch distance as the score. We can then attain a fitness score for the test suite by taking the sum of the branch distances for all goals in Eq. 7.5.

$$fitness = \sum_{goal \in Goals} distance(goal, suite). \quad (7.5)$$

The branch distance offers a fine-grained score that is more informative than simply measuring the coverage. Using this measurement allows faster attainment of coverage, and may enable the generation tool to attain more coverage than would otherwise be possible. The trade-off is the increased complexity of the implementation. At minimum, the tool would have to insert logging statements into the program. To avoid introducing side-effects into the behaviour of the class-under-test, measuring the branch distance may require complex instrumentation and execution monitoring.

¹¹ For more information on this calculation, and normalisation, see the explanations from McMinn, Lukasczyk, and Arcuri: [4–6].

7.5.2 *Multiple and Many Objectives*

When creating test cases, we typically have many different goals. A critical goal is to cover all important functionality, but we also want few and short test cases, we want tests to be understandable by humans, we want tests to have covered all parts of the system code, and so on. When you think about it, it is not uncommon to come up with five or more goals you have during test creation. If we plan to apply AI and optimisation to help us to create these test cases, we must encode these goals so that they are quantitative and can be automatically and quickly checked. We have the ability to do this through fitness functions. However, if we have multiple goals, we cannot settle for single-objective optimisation and instead have to consider the means to optimise all of these objectives at the same time.

A simple solution to the dilemma is to try to merge all goals together into a single fitness function which can then be optimised, often by adding all functions into a single score—potentially weighting each. For example, if our goals are high code coverage and few test cases, we could normalise the number of uncovered statements and the number of test cases to the same scale, sum them, and attempt to minimise this sum.

However, it is almost inevitable that many of your goals will compete with each other. In this two-objective example, we are punished for adding more test cases, but we are also punished if we do not cover all code. If these two goals were considered equally important, it seems possible that an outcome could be a single, large test case that tries to cover as much of the code as possible. While this might be optimal given the fitness function we formulated, it might not reflect what you really hope to receive from the generation tool. In general, it will be very hard to decide up-front how you want to trade off one objective versus the others. Even if you can in principle set weights for the different elements of the fitness function, when the objectives are fundamentally at odds with each other, there is no single weight assignment that can address all conflicts.

An alternative, and often better, solution is to keep each fitness function separate and attempt to optimise all of them at the same time, balancing optimisation of one with optimisation of each of the others. The outcome of such a multi-objective optimisation is not a single best solution, but a set of solutions that represent good trade-offs between the competing objectives. The set approximates what is known as the Pareto frontier, which is the set of all solutions that are not dominated by any other solution. A solution dominates another one if it is at least as good in all the objectives and better in at least one. This set of solutions represents balancing points, where the solution is the best it can be at some number of goals without losing attainment of the other goals. In our two-objective example of code coverage and test suite size, we might see a number of solutions with high coverage and a low number of test cases along this frontier, with some variation representing different trade-offs between these goals. We could choose the solution that best fits our priorities—perhaps taking a suite with 10 tests and 93% coverage over one with 12 tests and 94% coverage.

One well-known example of using multi-objective optimisation in software testing is the Sapienz test generation developed by Facebook to test Android applications through their graphical user interface [7]. It can generate test sequences of actions that maximise code coverage and the number of crashes, while minimising the number of actions in the test cases. The system, thus, simultaneously optimises three different objectives. It uses a popular genetic algorithm known as NSGA-II for multi-objective optimisation and returns a set of non-dominated test cases.

When the number of objectives grows larger, some of the more commonly used optimisation algorithms—like NSGA-II—become less effective. Recently, “many-objective” optimisation algorithms that are more suited to such situations have been proposed. One such algorithm was recently used to select and prioritise test cases for testing software product lines [8]. A total of nine different fitness functions are optimised by the system. In addition to the commonly used test case and test suite sizes, other objectives included are the pairwise coverage of features, dissimilarity of test cases, as well as the number of code changes the test cases cover.

7.5.3 *Human-Readable Tests*

A challenge with automated test generation is that the generated test cases typically do not look similar to test cases that human developers and testers would write. Variable names are typically not informative and the ordering of test case steps might not be natural or logical for a human reading and interpreting them. This can create challenges for using the generated test cases. Much of the existing research on test generation has not considered this a problem. A common argument has been that since we can generate so many test cases and then automatically run them there is little need for them to be readable; the humans will not have the time or interest to analyse the many generated test cases anyway. However, in some scenarios we really want to generate and then keep test cases around, for example when generating test cases to reach a higher level of code coverage. Also, when an automatically generated test case fails it is likely that a developer will want to investigate its steps to help identify what leads the system to fail. Automated generation of readable test cases would thus be helpful.

One early result focused on generating XML test inputs that were more comprehensible to human testers [9]. The developed system could take any XSD (XML Schema Definition) file as input and then create a model from which valid XML could then be generated. A combination of several AI techniques was then used to find XML inputs that were complex enough to exercise the system under test enough but not too complex since that would make the generated inputs hard for humans to understand. Three different metrics of complexity were used for each XML input (its number of elements, attributes, and text nodes) and the AI technique of Nested Monte-Carlo Search, an algorithm very similar to what was used in the AlphaGO Go playing AI [10], were then used to find good inputs for them. Results were encouraging but it was found that not all metrics were as easily optimised by the chosen

technique. Also, for real comprehensibility it will not be enough to only find the right size of test inputs; the specific content and values in them will also be critical.

Other studies have found that readability can be increased by—for example—using real strings instead of random ones (e.g., by pulling string values from documentation), inserting default values for “unimportant” elements (rather than omitting them), and limiting the use and mixture of `null` values with normal values¹² [11, 12].

A more recent trend in automated software engineering is to use techniques from the AI area of natural language processing on source code. For example, GitHub in 2021 released its Co-Pilot system, which can auto-complete source code while a developer is writing it [13]. They used a neural network model previously used for automatically generating text that looks like it could have been written by humans. Instead of training it on lots of human-written texts they instead trained it on human-written source code. The model can then be used to propose plausible completions of the source code currently being written by a developer in a code editor. In the future, it is likely that these ideas can and will also be used to generate test code. However, there are many risks with such approaches, and it is not a given that the generated test code will be meaningful or useful in actual testing. For example, it has been shown that Co-Pilot can introduce security risks [14]. Still, by combining these AI techniques with multi-objective optimisation it seems likely that we can automatically generate test cases that are both useful and understandable by humans.

7.5.4 Finding Input Boundaries

One fundamental technique to choose test input is known as boundary value testing/analysis. This technique aims to identify input values at the *boundary* between different visible program behaviours, as those boundaries often exhibit faults due to—for example—“off-by-one” errors or other minor mistakes. Typically, testers manually identify boundaries by using the software specification to define different *partitions*, i.e., sets of input that exhibit similar behaviours. Consider, for example, the creation of date objects. Testers can expect that valid days mainly lie within the range of 1–27. However, days greater or equal than 28 might reveal different outputs depending on the value chosen for the month or year (e.g., February 29th). Therefore, most testers would choose input values between 28 and 32 as *one of* the boundaries for testing both valid and invalid dates (similarly, boundary values for day between 0 and 1).

The *program derivative* measures the program’s sensitivity to behavioural changes for different sets of input values [15]. Analogous to the mathematical concept of a derivative, the program derivative conveys how function values (output) change when varying independent variables (input). In other words, we can detect boundary values by detecting notable output differences when using similar sets of inputs [16].

¹² Although, of course, some `null` values should be applied to catch common “null pointer” faults.

We quantify the similarities between input and output by applying various distance functions that quantify the similarity between a pair of values. Low distance values indicate that the pair of values are similar to each other. Some of the widely used distance functions are the Jaccard index (strings), Euclidean distance (numerical input), or even the more generic Normalised Compressed Distance (NCD).

The program derivative analyses the ratio between the distances of input and output of a program under test (Eq. 7.6). Let a and b be two different input values for program P with corresponding output values $P(a)$ and $P(b)$. We use the distance functions $d_i(a, b)$ and $d_o(P(a), P(b))$ to measure the distance between, respectively, the pair of input and their corresponding output values. The program derivative (PD) is defined as [16].

$$PDQ_{do,di}(a, b) = \frac{d_o(P(a), P(b))}{d_i(a, b)}, b \neq a \quad (7.6)$$

Note that high derivative values indicate a pair of very dissimilar output (high numerator) with similar inputs (low denominator), hence revealing sets of input values that are more sensitive to changes in the software behaviour. Going back to our Date example, let us consider the d_i and d_o for Dates as the edit distance¹³ between the inputs and outputs, respectively, when seen as strings (note that valid dates are just printed back as strings on the output side):

- $i1 = "2021-03-31"; P(i1) = "2021-03-31".$
- $i2 = "2021-04-31"; P(i2) = "Invalid date".$
- $i3 = "2021-04-30"; P(i3) = "2021-04-30".$

As a consequence, $d_i(i1, i2) = 1$ as only one character changes between those input, whereas the output distance $d_o(P(i1), P(i2)) = 12$ since there is no overlap between the outputs, resulting in the $PD = 12/1 = 12$. In contrast, the derivative $PD(i1, i3) = 2/2 = 1$ is significantly lower and does not indicate any sudden changes in the output. In other words, the derivative changes significantly for $i1$ and $i2$, indicating boundary behaviour. Figure 7.15 illustrates the program derivative of our example by varying months and dates for a fixed year value (2021) for a typical Date library. We see that automated boundary value testing can help highlight and, here, visualise boundary values.

Note that the high program derivative values delimit the boundaries for the input on those two dimensions. Therefore, program derivative is a promising candidate to be a fitness function to identify boundary values in the input space. Using our BMI example, note that we can use the program derivative to identify the pairs of height and weight that trigger changes between classifications by comparing the variation in the output distance of similar input values. For instance, the output classifications can change, e.g., from “Overweight” to “Obese” by comparing individuals of same height but different weight values.

¹³ The edit distance between two strings A and B is the number of operations (add, remove or replace a character) required to turn string A into the string B .

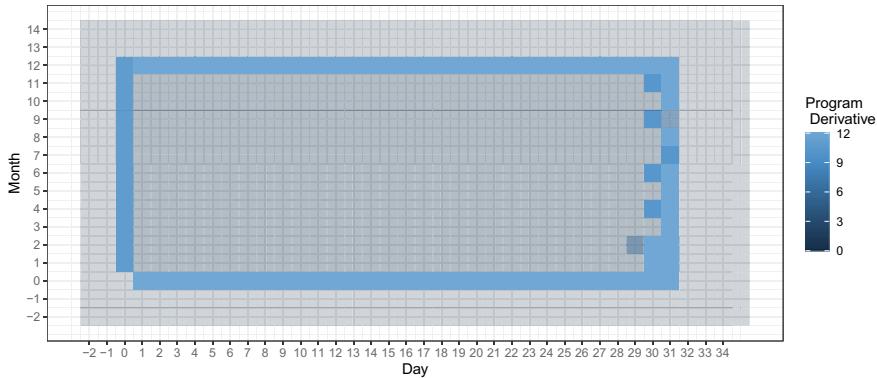


Fig. 7.15 Plot of the program derivative by varying days and months for the year 2021. Higher opacity and brighter colours indicate higher derivative values. The derivative indicates the values that divide the boundary between valid and invalid combinations of days and months. Figure adapted from [16]

However, there are still many challenges when automating the generation of boundary values. First, software input is often complex and non-numerical such as objects or standardised files, which introduces the challenge of defining a suitable and accurate distance function able to measure the distances between input/output values. Second, the input space can have many dimensions (e.g., several input arguments) of varied types and constraints such that searching through that space is costly and sometimes infeasible. Last, but not least, boundary values often involve the tester's domain knowledge or personal experience that is hard to abstract in terms of functions or quantities (e.g., think of the Millennium bug for the date 2000-01-01). More important than fully automating the search of boundaries, testers are encouraged to employ boundary value exploration (BVE) techniques. BVE is a set of techniques (e.g., the visualisation in Fig. 7.15) that propose sets of candidate boundary values to help testers in refining their knowledge of boundaries in their own programs under test [16].

7.5.5 Finding Diverse Test Suites

A natural intuition that we have as software testers is that the tests we write and run need to differ from each other for the system to be properly tested. If we repeatedly re-run the same test case, or some set of very similar test cases, they are unlikely to uncover unique behaviours of the tested system. All of these similar tests will tend to pass or fail at the same time. Many AI-based techniques—including search-based approaches—have been proposed to select a good and complementary set of test cases, i.e., a diverse test suite. For example, recent research uses reinforcement learning to adapt the strategy employed by a search-based algorithm to generate

more diverse tests for particular classes-under-test [17]. A study comparing many different test techniques found that techniques focused on diversity were amongst the best possible in selecting a small set of test cases [18].

A key problem in applying AI to find diverse test suites is how to quantify diversity. There are many ways in which we can measure how different test cases are, i.e., such as their length, which methods of the tested system they call, which inputs they provide, etc. A general solution is to use general metrics from the area of Information Theory that can be used regardless of the type of data, length, or specifics of the test cases we want to analyse. One study showed how metrics based on compression were very useful in quantifying test case diversity [19]. Their experiments also showed that test sets comprised of more diverse test cases had better coverage and found more faults.

A potential downside of these methods is that they can be expensive in terms of computations; many test cases and sets need to be considered to find the most diverse ones. Later research has proposed ways to speed diversity calculations up. One study used locality-sensitive hashing to speed up the diversity calculations [20]. Another study used the pairwise distance values of all test cases as input to a dimensionality reduction algorithm so that a two-dimensional (2D) visual “map” of industrial test suites could be provided to software engineers [21].

7.5.6 *Oracle Generation and Specification Mining*

This chapter has focused on automatically generating the test inputs and test actions of good test cases. This excludes a key element of any test case: how to judge if the behaviour of the system under test is correct. Can AI techniques help us also in generating oracles that make these judges? Or more generally, can we find or extract, i.e., mine, a specification of the SUT from actual executions of it?

Oracle generation is notoriously difficult and likely cannot be solved once and for all. While attempts have been made to “learn” a full oracle using supervised learning techniques, they are typically only viable on small and simple code examples.¹⁴ Still, some researchers have proposed that AI can at least partly help [23]. For example, one study used the Deep AI technique of neural embeddings to summarise and cluster the execution traces of test cases [24]. Their experiments showed that the embeddings were helpful in classifying test case executions as either passing or failing. While this cannot directly be used as an oracle it can be used to select test cases to show to a human tester which can then more easily judge if the behaviour is correct or not. Such interactive use of optimisation and AI in software testing has previously been shown to be effective [25].

¹⁴ An overview of attempts to use machine learning to derive oracles is offered by Fontes and Gay: [22].

7.5.7 Other AI Techniques

Many other AI and Machine Learning techniques beyond those that we have described in this chapter have been used to support unit testing tasks, from input generation, to test augmentation, to test selection during execution. The trend is also that the number of such applications grows strongly year by year. Below we provide a few additional examples.

Researchers have proposed the use of Reinforcement Learning when generating test inputs [26]. They implemented the same test data generation framework that had been previously used with traditional search-based, meta-heuristics [27] as well as with Nested Monte-Carlo Search [9] but instead used Reinforcement Learning to generate new test cases. A neural net was used to model the optimal choices when generating test inputs for testing a system through its API. Initial results showed that technique could reach higher coverage for larger APIs where more complex scenarios are needed for successful testing. Another early study showed how Deep Reinforcement Learning could develop its own search-based algorithm that achieves full branch coverage on a training function and that the trained neural network could then achieve high coverage also on unseen tested functions [28]. This indicates that modern AI techniques can be used to learn transferable testing skills.

Reinforcement learning has also been used *within* search-based test generation frameworks to adapt the test generation strategy to particular systems or problems. For example, it has been applied to automatically tune parameters of the metaheuristic [29], to select fitness functions in multi-objective search in service of optimising a high-level goal (e.g., selecting fitness functions that cause a class to throw more exceptions) [17], and to transform test cases by substituting individual actions for alternatives that may assist in testing inheritance in class hierarchies or covering private code [30].

Other researchers have proposed the use of supervised machine learning to generate test input (e.g., [31, 32]). In such approaches, a set of existing test input and results of executing that input (either the output or some other result, such as the code coverage) are used to train a model. Then, the model is used to guide the selection of new input that attains a particular outcome or interest (e.g., coverage of a particular code element or a new output). It has been suggested that such approaches could be useful for boundary identification—Budnik et al. propose an exploration phase where an adversarial approach is used to identify small changes to input that lead to large differences in output, indicating boundary areas in the input space where faults are more likely to emerge [31]. They also suggest comparing the model prediction with the real outcome of executing the input, and using misclassifications to indicate the need to re-train the model. Such models may also be useful for increasing input diversity as well, as prediction uncertainty indicates parts of the input space that have only been weakly tested [32].

7.6 Conclusion

Unit testing is a popular testing practice where the smallest segment of code that can be tested in isolation from the rest of the system—often a class—is tested. Unit tests are typically written as executable code, often in a format provided by a unit testing framework such as `pytest` for Python.

Creating unit tests is a time and effort-intensive process with many repetitive, manual elements. Automation of elements of unit test creation can lead to cost savings and can complement manually written test cases. To illustrate how AI can support unit testing, we introduced the concept of search-based unit test input generation. This technique frames the selection of test input as an optimization problem—we seek a set of test cases that meet some measurable goal of a tester—and unleashes powerful metaheuristic search algorithms to identify the best possible test input within a restricted timeframe.

Readers interested in the concepts explored in this chapter are recommended to read further on the advanced concepts, such as distance-based fitness functions, multi-objective optimization, generating human-readable input, finding input boundaries, increasing suite diversity, oracle generation, and the use of other AI techniques—such as machine learning—to generate test input.

References

1. Android Developers, Fundamentals of testing (2020). <https://developer.android.com/training/testing/fundamentals>
2. Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An empirical analysis of flaky tests, in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014 (ACM, New York, NY, USA, 2014), pp. 643–653
3. M. Eck, F. Palomba, M. Castelluccio, A. Bacchelli, Understanding flaky tests: the developer’s perspective, in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 830–840
4. Phil McMinn, Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.* **14**, 105–156 (2004)
5. S. Lukasczyk, F. Kroiß, G. Fraser, Automated unit test generation for python, in Aldeida Aleti and Annibale Panichella, editors, *Search-Based Software Engineering*, ed. by A. Aleti, A. Panichella (Springer International Publishing, Cham, 2020), pp. 9–24
6. Andrea Arcuri, It really does matter how you normalize the branch distance in search-based software testing. *Softw. Test., Verif. Reliab.* **23**(2), 119–147 (2013)
7. K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for android applications, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 94–105
8. R.M. Hierons, Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **11**(4), 448 (2002)
9. S. Pouling, R. Feldt, The automated generation of humancomprehensible xml test sets, in *Proceedings of the 1st North American Search Based Software Engineering Symposium (Nas-BASE)* (2015)

10. D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search. *Nature* **529**(7587), 484–489 (2016)
11. P. McMinn, M. Stevenson, M. Harman, Reducing qualitative human oracle costs associated with automatically generated test data, in *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10 (ACM, New York, NY, USA, 2010), pp. 1–4
12. A. Alsharif, G.M. Kapfhammer, P. McMinn, What factors make SQL test cases understandable for testers? a human study of automatic test data generation techniques, in *International Conference on Software Maintenance and Evolution (ICSME 2019)* (2019), pp. 437–448
13. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code (2021). [arXiv:2107.03374](https://arxiv.org/abs/2107.03374)
14. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, R. Karri, An empirical cybersecurity evaluation of github copilot's code contributions (2021). [arXiv:2108.09293](https://arxiv.org/abs/2108.09293)
15. R. Feldt, F. Dobslaw, Towards automated boundary value testing with program derivatives and search, in *Search-Based Software Engineering*, ed. by S. Nejati, G. Gay (Springer International Publishing, Cham, 2019), pp. 155–163
16. F. Dobslaw, F.G. de Oliveira Neto, R. Feldt, Boundary value exploration for software analysis, in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 346–353
17. H. Almulla, G. Gay, Learning how to search: generating effective test cases through adaptive fitness function selection. *CorR* (2021). [arXiv:2102.04822](https://arxiv.org/abs/2102.04822)
18. C. Henard, M. Papadakis, M. Harman, Y. Jia, Y.L. Traon, Comparing white-box and black-box test prioritization, in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (IEEE, 2016), pp. 523–534
19. R. Feldt, S. Pouling, D. Clark, S. Yoo, Test set diameter: quantifying the diversity of sets of test cases, in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (IEEE, 2016), pp. 223–233
20. B. Miranda, E. Cruciani, R. Verdecchia, A. Bertolino, Fast approaches to scalable similarity-based test case prioritization, in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (IEEE, 2018), pp. 222–232
21. F.G.D.O. Neto, R. Feldt, L. Erlenkov, J.B.D.S. Nunes, Visualizing test diversity to support test optimisation, in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)* (IEEE, 2018), pp. 149–158
22. A. Fontes, G. Gay, Using machine learning to generate test oracles: a systematic literature review, in *Proceedings of the 1st International Workshop on Test Oracles*, TORACLE 2021 (Association for Computing Machinery, New York, NY, USA, 2021), pp. 1–10
23. W.B. Langdon, S. Yoo, M. Harman, Inferring automatic test oracles, in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)* (IEEE, 2017), pp. 5–6
24. F. Tsimpourlas, G. Rooijackers, A. Rajan, M. Allamanis, Embedding and classifying test execution traces using neural networks. *IET Softw.* (2021)
25. Bogdan Marculescu, Robert Feldt, Richard Torkar, Simon Pouling, An initial industrial evaluation of interactive search-based testing for embedded software. *Appl. Soft Comput.* **29**, 26–39 (2015)
26. S. Huurman, X. Bai, T. Hirtz, Generating API test data using deep reinforcement learning, in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (2020), pp. 541–544
27. R. Feldt, S. Pouling, Finding test data with specific properties via metaheuristic search, in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (IEEE, 2013), pp. 350–359
28. J. Kim, M. Kwon, S. Yoo, Generating test input with deep reinforcement learning, in *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)* (IEEE, 2018), pp. 51–58
29. Y. Jia, M.B. Cohen, M. Harman, J. Petke, Learning combinatorial interaction test generation strategies using hyperheuristic search, in *Proceedings of the 37th International Conference on Software Engineering*, vol. 1, ICSE '15 (IEEE Press, 2015), pp. 540–550

30. W. He, R. Zhao, Q. Zhu, Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software. *Chin. J. Electron.* **24**(1), 38–45 (2015)
31. C. Budnik, M. Gario, G. Markov, Z. Wang, Guided test case generation through AI enabled output space exploration, in *Proceedings of the 13th International Workshop on Automation of Software Test*, AST '18 (Association for Computing Machinery, New York, NY, USA, 2018), pp. 53–56
32. N. Walkinshaw, G. Fraser, Uncertainty-driven black-box test data generation, in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017), pp. 253–263

Chapter 8

Artificial Intelligence Techniques in System Testing



Michael Felderer, Eduard Paul Enoiu, and Sahar Tahvili

Abstract System testing is essential for developing high-quality systems, but the degree of automation in system testing is still low. Therefore, there is high potential for Artificial Intelligence (AI) techniques like machine learning, natural language processing, or search-based optimization to improve the effectiveness and efficiency of system testing. This chapter presents where and how AI techniques can be applied to automate and optimize system testing activities. First, we identified different system testing activities (i.e., test planning and analysis, test design, test execution, and test evaluation) and indicated how AI techniques could be applied to automate and optimize these activities. Furthermore, we presented an industrial case study on test case analysis, where AI techniques are applied to encode and group natural language into clusters of similar test cases for cluster-based test optimization. Finally, we discuss the levels of autonomy of AI in system testing.

8.1 Introduction

The effectiveness and efficiency of system testing, which is conducted on a complete, integrated system to evaluate the systems' compliance with its requirements, is essential for developing high-quality software-intensive systems. However, the

M. Felderer (✉)
University of Innsbruck, Innsbruck, Austria

Blekinge Institute of Technology, Karlskrona, Sweden
e-mail: michael.felderer@uibk.ac.at

E. P. Enoiu · S. Tahvili
Mälardalen University, Västerås, Sweden
e-mail: eduard.paul.enoiu@mdh.se

S. Tahvili
e-mail: sahar.tahvili@ericsson.com

S. Tahvili
Ericsson AB, Stockholm, Sweden

degree of automation in system testing is still low (especially when compared to unit testing) and system tests are performed manually to a large extent. The main reasons for this circumstance are that system testing typically relies on less formal artifacts, especially requirements, than unit testing, which is directly linked to source code, also resulting in the fact that test automation is more complex on the system level.

Thus, Artificial Intelligence (AI) techniques like machine learning, natural language processing, or search-based optimization can significantly improve the effectiveness and efficiency of system testing. For instance, natural language processing has a high potential to increase the degree of automation in system testing, which typically relies on natural language requirements.

This chapter provides an overview of the state of the art and future trends on the application of AI techniques in different system testing activities (i.e., test planning and analysis, test design, test execution, and test evaluation). Furthermore, a concrete industrial case study is presented in which natural language processing, unsupervised machine learning, and search-based techniques have successfully been applied to automate and optimize system testing activities.

To support our concrete industrial study, we provide a project that can be used for automated functional dependency detection from natural language test specifications. The provided code is written in Python 3, so it assumes you have this installed on your local machine to execute the examples. The code repository¹ is available at <https://github.com/leohatvani/clustering-dependency-detection>. Its README file comprises instructions on how to download and execute the code.

The chapter is organized as follows: In Sect. 8.2, we provide an overview about system testing. In Sect. 8.3, we discuss the application of AI in system testing. In Sect. 8.4, we present an industrial case study on the application of selected AI techniques (i.e., natural language processing, clustering, and search-based techniques) to system testing. In Sect. 8.5, we reflect on the levels of autonomy of AI in system testing. Finally, in Sect. 8.6, we conclude this chapter.

8.2 System Testing

According to the IEEE standard glossary of software engineering terminology [17], *testing* is the process of operating a system or component (or unit) under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. The definition (and this chapter) refers to dynamic testing, where the system is executed (in contrast to static testing) and considers unit- and system-level testing. This classification based on the test level, where units of the system are tested either in isolation or together as the whole system, is important in testing. The IEEE standard glossary of software engineering terminology [17] defines

¹ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

system testing as testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. So system testing explicitly refers to requirements as a specification. The requirements can refer to functional or extra-functional system properties. A system often comprises hardware and software; however, we focus only on the software testing aspects.

Conceptually, system-level testing differs in several respects from unit-level testing. The following items differentiate system testing from unit testing:

- What constitutes a test input changes and is rather input via a user interface or a system configuration?
- Testing extra-functional properties like performance, security, or usability is more important as they can typically only be tested on the system level.
- Domain knowledge becomes more important on the system level, which requires the integration of domain experts into the test process.
- Classically, system testing is the part of testing (differing from unit testing) that is not anymore in the plain responsibility of developers as domain aspects and the entire system are essential.
- Test automation is multi-faceted, often only partially possible requires specific test engineering know-how, and tests might intermediately fail.
- The Oracle problem becomes more complex, i.e., deciding on a pass or fail is more difficult and often even not possible.
- System specifications are not provided in code but in informal, semi-formal, or formal requirements (models).

Testing, in general, and system testing, in particular, comprise several activities. The generic testing process comprises the following activities:

- *Test Planning and Analysis*. This activity comprises planning of the means and schedule for test activities as well as the analysis of test artifacts to identify test conditions.
- *Test Design*. This activity comprises the derivation and specification of test cases from test conditions like coverage criteria.
- *Test Execution*. This activity comprises the preparation of the test environment and running tests on the component or system under test.
- *Test Evaluation*. This activity comprises decisions on test results and their reporting.

In the following, we present important system testing approaches and to what test activities they relate.

Risk-based testing [11] takes explicit risk assessments into account to steer all test activities. Therefore, it is linked to all test activities mentioned before. However, in practice, risk-based testing is primarily used to steer test planning (i.e., distribution of resources) and test execution (i.e., test case prioritization and selection) purposes.

Model-based testing [36] relies on explicit behavior models that encode the intended behaviors of a system under test and/or the behavior of its environment. The main focus of model-based testing lies in automated test case generation, i.e.,

on test design. However, test models can also support test planning and analysis, test execution, and test evaluation.

Exploratory testing [1] is simultaneous learning, test design, and test execution. It is therefore not only linked to test design and execution, but also impacts test planning and evaluation.

Agent-based testing [22] is defined as the application of AI-infused agents (i.e., software bots, intelligent agents, autonomous agents, multi-agent systems) to software testing problems. Most of these approaches are used for test planning and analysis as well as test design and execution.

Hardware-in-the-loop and *Software-in-the-loop* [14] use simulations to test hardware and software, respectively, especially in the context of embedded and cyber-physical systems. The focus lies on test execution.

8.3 Application of AI in System Testing

In this section, we first present a classification scheme for the application of AI in system testing and discuss selected approaches that are classified in the defined scheme. Then, we discuss the optimization of various test activities based on AI.

8.3.1 Classification Scheme

Given the current expansion of the use of AI in software testing and the large number of ways AI can be applied during software development, we aim to outline the categories of AI applications according to their application point on the system testing process, i.e., test planning and analysis, test design, test execution as well as test evaluation.

Feldt et al. [12] proposed a taxonomy that categorizes the different ways of applying AI in software engineering according to their point of AI application, the type of AI technology used, and the automation level allowed. When using their classification scheme on some existing work, the authors found that AI application to software engineering focused on supporting stakeholders during the development process, but did not directly affect the source code or the runtime behavior of the systems. In this chapter, we focus on the dimension where AI techniques are applied in the software testing process.

Here the focus of the AI is to support or optimize the verification that the system as a whole can accomplish its task.

Even though AI can be applied in different ways during system testing, we argue that a simpler taxonomy focusing on the generic steps in a system-level testing process can help in the understanding of the points of applications during system-level testing. Many AI approaches are applied to more than one system testing step and the results can be combined.

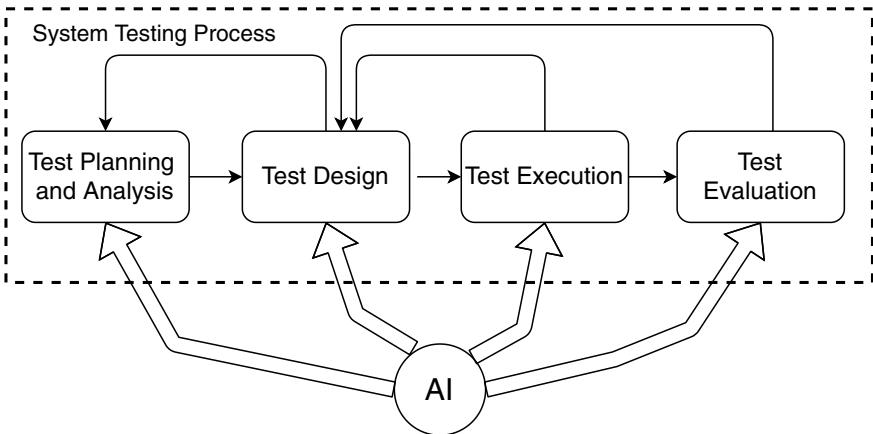


Fig. 8.1 Ways of applying AI in a generic system testing process. These activities are based on the generalized test process outlined by [6]

The following steps where AI is applied to system testing are outlined in Fig. 8.1:

- *Test Planning and Analysis.* AI can be applied in this activity by determining what is going to be tested and optimizing a test plan by analyzing the test artifacts created during software development (e.g., requirement documents, test specifications).
- *Test Design.* AI can be applied in this activity for automating system-level test design. This has been proposed to allow test cases to be created with less effort. The goal is to automatically find a small set of test cases that check the correctness of the system and guard against (previous as well as future) faults.
- *Test Execution.* After test cases have been generated, AI can be applied during the test selection and execution process in order to make the following determinations: which test cases to execute during regression testing, which each test will evaluate system configurations, and which test setups are available for the actual running of each test case.
- *Test Evaluation.* As the test cases execute, valuable data is generated that AI can exploit through data mining techniques to evaluate the test result and localize suspicious program behavior as well as to cluster similar and independent faults.

The main criterion for classifying the application of AI in system testing is the “when”, i.e., which system testing activity is supported by AI. This criterion is linked to the “what”, i.e., the software and test artifacts and data the application of AI depends on and refers to. This can be test cases, system interfaces, GUI elements, natural language requirements, defects, etc.

To give an overview and apply the classification scheme defined before, we provide selected approaches that vary in the types of AI techniques applied and the targeted test activity based on the classification scheme presented before.

Ramler and Felderer [29] integrate machine-learning-based defect prediction based on classifiers into risk-based testing. The goal of the approach is to predict risk

probability, which is used to plan system testing, in general, and the depth of testing for different system components, in particular.

Tahvili et al. [32] provide a test analysis approach that derives test cases' similarities and functional dependencies directly from the test specification documents written in natural language, without requiring any other data source. The approach uses natural language processing to detect text-semantic similarities between test cases and then groups them using different clustering algorithms. The approach is further discussed in the case study in Sect. 8.4.

Adamo et al. [2] apply reinforcement learning for automating test design and execution of GUI testing of Android apps. The authors use a test generation algorithm to systematically select events and explore the GUI of an application under test without requiring a preexisting abstract model.

Briand et al. [7] apply search-based approaches to automate test design for stress testing of real-time systems. Genetic algorithms were used to search for the sequence of arrival times of events for aperiodic tasks that would cause the most significant delays in the execution of the target task. The fitness function was expressed in an exponential form, based on the difference between the deadline of an execution and the execution's actual completion.

Memon et al. [28] use AI plan for automated test design. For example, given a set of operations, an initial state, and a goal state for a GUI, a planning system produces a sequence of operations that transforms the initial state to the goal state.

Frounchi et al. [13] use machine learning to construct an oracle for test evaluation, which can then be used to verify the correctness of image segmentations automatically.

Arcuri et al. [4] use AI to generate whole-suite system-level test cases for RESTful API web services by using the Many Independent Objective (MIO) algorithm.

8.3.2 *Test Optimization*

Optimizing test activities, i.e., test planning and analysis, test design, test execution, or test evaluation, using AI is a process we call *test optimization*. Test optimization goes beyond plain automation and defines an explicit optimization function. The overall architecture of a generic AI-based system testing process used for test optimization is shown in Fig. 8.2. This approach starts with only two inputs: the choice of the representation of the problem based on the available raw data and the definition of the heuristic function. With these two, an engineer can implement AI-based system testing using optimization algorithms and obtain results.

Test optimization helps make the test process more time and resources efficient without compromising test accuracy or coverage. Eliminating unnecessary steps and automating some steps can be considered the two main strategies in an efficient test optimization process to save time, reduce errors, and avoid duplicate work.

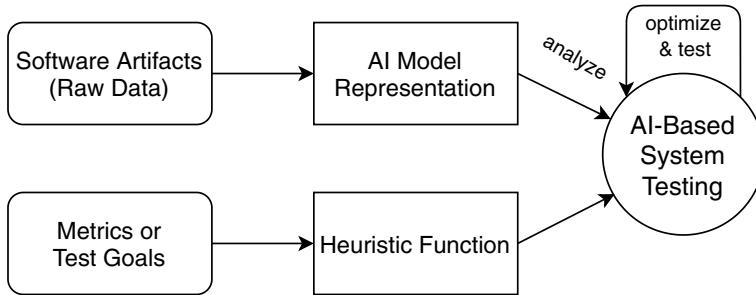


Fig. 8.2 Overall architecture of AI-based system testing approaches adapted from [15]

Employing the test optimization in an iterative testing process (e.g., where regression testing is applied) can significantly improve system testing efficiency and effectiveness.

However, guiding a testing team to work more efficiently, supported by a proper test optimization approach, is a challenging task. Generally, optimizing a continuous testing process requires hard work and effort, in terms of designing and implementing the new solution without stopping the testing process.

In this regard, evaluating the efficiency of the investment (e.g., return on investment, internal rate of return) in all optimization process is a golden key that can guide the testing team to select a proper optimization approach. Therefore, employing artificial intelligence techniques for optimizing the testing process has received a great deal of attention recently. However, training a single artificial inference model can be an ongoing costly process since the data that feeds the AI models tends to change over time. Furthermore, there's a reciprocal relationship between big data and AI especially in a continuous testing process, where large data (e.g., test results) is generated after each execution. However, one of the main purposes of AI is to minimize the need for human intervention, which might end up having higher accuracy and less uncertainty and ambiguity.

Figure 8.3 illustrates an overall overview of employing AI technologies for optimization of a testing process. As one can see in Fig. 8.3, the optimization process can be performed in three main phases:

1. **Analyze:** Everything regarding the data, e.g., data gathering and data pre-processing, is being performed in this phase. The required data for applying an AI-based solution should be captured and be ready for training the AI model. The required data might be different based on the optimization goal. However, all system and test artifacts such as requirements specification, test cases (specification or script), and test results (log files) can be considered as data.
2. **Optimize:** The AI model needs to be trained in this phase using the pre-processed data from the previous phase. Moreover, the hyperparameter tuning needs to be done in this phase. Tuning (also known as hyperparameter optimization) refers to choosing a set of optimal hyperparameters for a learning algorithm (see “*Improve*”

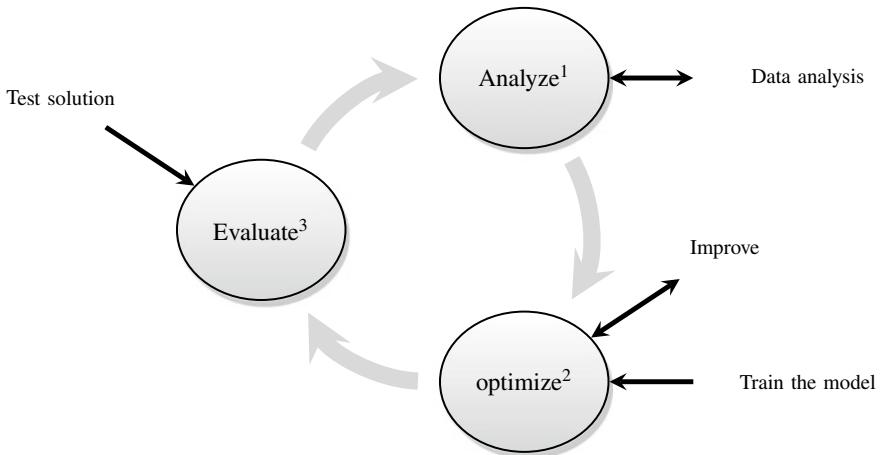


Fig. 8.3 Test optimization process using artificial intelligence

arrow in Fig. 8.3). This improvement process (tuning) can be performed via employing different approaches, e.g., grid search, random search, Bayesian optimization, population-based approaches, etc.

3. **Evaluate:** The proposed AI-based solution needs to be evaluated with respect to specific metrics in this phase.

However, in a continuous testing process, a large and new dataset might be generated, thus the optimization process should be connected to both *Analyze* and *Test* phases directly. The mentioned phases in Fig. 8.3 are exemplified in an industrial case study in the next section.

Considering all mentioned issues, especially all aspects of test automation, especially in the context of regression testing, have a high potential for test optimization. In fact, instead of regressing the same test cases each time, this process can be fully automated. Therefore, test automation can be a suitable test optimization approach in large industries where some of the features of the software hardly change when a new build is made [21]. However, the maintenance of test scripts needs to be considered a recurring expense that is usually a low cost. Regression testing tasks that benefit, in particular, from test optimization based on AI techniques are as follows:

- *Test case selection:* Refers to identifying a subset of test cases for execution which has to be sufficient for achieving given requirements.
- *Test case prioritization:* Ranks the test cases based on some priority score to guide testing activities.
- *Test suite minimization:* Keeps the test case precise and unique and eliminates non-effective test cases which reduces regression testing time and lowers costs while maintaining quality.

In addition, utilizing *exploratory testing* by employing on-the-go testing in order to make the testing process faster and to make use of domain expertise has a high potential for test optimization. This topic touches on the different levels of autonomy of AI in system testing and is discussed in Sect. 8.5.

8.4 Industrial Case Study

In this section, we describe a replicable industrial case study on system test analysis. We first motivate it and define its industrial context. Then we present the underlying Natural Language Processing approach for detecting test case similarity. Afterward, we discuss clustering similar test cases using unsupervised machine learning. Then, we discuss the data processing and visualization. Finally, we sketch test optimization application scenarios.

8.4.1 Motivation

Testing a software product in large industries is a time- and resource-consuming process. Manual testing is still an essential approach to system testing, especially in safety-critical domains. In a purely manual testing process, all test cases are created and executed manually by the tester without using any automation tool. In fact, a testing team consisting of several testers creates and later executes manual test cases based on the requirement specifications (that takes the different stakeholders' perspectives into account).

System (integration) testing is recognized as one of the most complex and critical levels of testing, where the interaction of all subsystems and the entire system should be tested. Therefore, the number of test scenarios that need to be defined for testing the interaction between modules might be very high. Considering a high number of test cases that need to be tested manually highlights the need for test optimization. However, manual test creation by different testers might lead to having similar or even duplicated test cases. Similar test cases can refer to test cases designed to test the same function or require the same system setup, preconditions, or postconditions. Identifying similar or duplicate test cases in an early stage of a testing process can help one to apply different test optimization approaches such as test case selection, prioritization, scheduling, and test suite minimization. For example, assume that we have clusters of similar test cases. We can then use these clusters for test optimization, e.g., by eliminating some clusters or just selecting some test case samples from the clusters and ranking them for test execution.

Clustering, in general, is an *unsupervised machine learning* approach that scans data and groups it without any labels. Therefore, unsupervised (machine) learning can be considered a suitable approach for solving the industrial problem of identifying similar test cases and using this information to guide system testing.

However, detecting similar test cases for a large-size product is a challenging task that might suffer from human judgment, uncertainty, and ambiguity. Since the input for the similarity section is the natural language test case specification (due to manual testing). Thus, employing AI techniques such as natural language processing (NLP), unsupervised machine learning, and deep learning can be considered a proper approach.

8.4.2 Industrial Context

In the following, we provide an industrial case study conducted at Bombardier Transportation (BT) in Sweden.² This case study focuses on finding the similarity and duplication between manual system integration test cases. It, therefore, covers a test analysis scenario, where system and test artifacts are analyzed. In this regard, the case study provided in this section can be employed for different test optimization purposes such as test case selection (selecting a subset of test cases for test execution or automation) or test scheduling (ranking the test cases for test execution). In other words, the running industrial case study in this section can easily be reused for achieving several test optimization goals simultaneously. The case study is performed in the following steps:

1. One ongoing testing project is selected as a case under study.
2. A total number of 1, 748 test specifications are extracted from the database at BT.
3. Each test case is saved in a separate file in the comma-separated values (.CSV) format.
4. Some irrelevant information such as the name of the tester, date, and time is removed from the test specification.
5. The .CSV files are utilized later as input to the natural language processing model.
6. The generated outputs of the NLP approach are then clustered.
7. The obtained clusters are later removed or ranked for the test execution.

8.4.3 Detecting Test Case Similarity Using Natural Language Processing

As stated earlier, in a manual testing process, a large number of test specifications with short or long text are processed. The provided test specifications include the testing procedure, requirements specification, system setup, etc., which are generally subjective and also semantics-oriented. Detecting the similarity between created test

² Please note that the authors have approval from the third party for the utilized case study and its code provided in the Git repository <https://github.com/leohatvani/clustering-dependency-detection>. The permission to use the code is also granted to the reader.

cases is the problem of finding the score to which test cases have similar content. Usually, two text documents are similar either lexically or semantically if the two text documents are used in the same context or have the same meaning [26]. Due to a wide application of semantic similarity detection (e.g., text summarization [20], topic extraction [18], finding duplication [23], and text document clustering and classification [34]), it always plays a vital role in the NLP domain. The following approaches can be considered as the main solutions for detecting the similarity problem between texts:

- *String distance algorithms* such as token-based, edit-based, and string-matching.
- *Normalized compression distance* such as bzip2, gzip, and zstd.
- *Deep learning* such as Doc2vec and SBERT.

However, most of the mentioned solutions for similarity detection deliver the extracted features as input to machine learning algorithms, especially clustering algorithms. In this regard, the deep learning approach can execute feature engineering by itself compared to other existing solutions.

We especially want to highlight Doc2vec, which has been applied in our approach. It is a deep learning model for representing documents as a vector which was first proposed by Le and Mikolov [27]. Doc2vec has excellent scalability and has filled gaps of previously proposed approaches in the test analysis domain where the semantics of the words were ignored (see [34]). Doc2vec reads different text documents as input, transfers them to the high-dimensional vectors, and measures the cosine distance between them. The dimension of each vector is directly related to the text size. Via applying a proper clustering method on the obtained vectors provided by Doc2vec, we can divide the similar text (test specification in this study) into different clusters [33].

8.4.4 Clustering Similar Test Cases Using Unsupervised Machine Learning

Clustering large datasets that have thousands of dimensions is a complex and challenging problem. Most of the proposed solutions in the state of the art focus on dimension reduction. In that process, data from a high-dimensional space needs to be transferred into a low-dimensional space, which can be handled by most of the clustering and classification algorithms [24]. However, during dimensionality reduction, some properties of the data might be lost, whereas some meaningful properties of the original data might remain [34].

The clustering algorithms can mainly contain the following commonly used ones:

- *Hierarchical based* such as the Agglomerative algorithm.
- *Partition based* such as the Affinity algorithm.
- *Density based* such as the HDBSCAN algorithm.

In order to keep all data properties, we recommend employing new types of clustering algorithms that can handle high-dimensional datasets. Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) is a hierarchical clustering algorithm that extracts a flat clustering based on the stability of clusters. Moreover, HDBSCAN can produce a cluster of non-clustered data points that need to be interpreted accordingly based on the application. For instance, in our setting, non-clusterable data points can be considered as non-similar test cases.

Furthermore, Fuzzy C-Means Clustering (FCM) is another alternative for solving the clustering problem. FCM generally considers each object a member of every cluster, with a variable degree of membership [33], which makes sense in the context of system testing. In fact, each test case can be similar to one or more test cases. Listing 1 shows a Python code snippet for selecting and applying HDBSCAN and FCM.

```

1 if opt_method=='hdbscan':
2     clusterer = hdbscan.HDBSCAN().fit(values)
3     mylabels = clusterer.labels_
4 elif opt_method == 'fcm':
5     values_rotated = np.rot90(values)
6     cntr, u, u0, d, jm, p, fpc = skfuzzy.cluster.
7     cmeans(values_rotated, opt_nclusters, 2, error
8         = 0.005, maxiter=1000, init=None)
9     mylabels = np.argmax(u, axis=0)
else:
    print("Clustering method unknown")

```

Listing 1 Python script for clustering

8.4.5 Data Processing and Visualization

In this case study, we combine Doc2vec with the HDBSCAN clustering algorithm in order to divide all test cases into several clusters based on their semantic similarity in the test specifications. In order to rerun this case study with your own data, the following steps need to be performed³:

1. All irrelevant information such as testing date, time, station, and testers' ID should be eliminated from the test case specifications. In other words, just the relevant information which indicates the testing purposes, requirements, steps, and testing procedure need to be kept.
2. For improving the accuracy of the model, different data pre-processing techniques such as tokenization, which is a way of separating a piece of text into smaller units called tokens, should be applied to the test case specifications.
3. The pre-processed data (i.e., the test case specifications) are the input for Doc2vec providing numeric representations of the test case specifications.

³ The source code of our work can be found online at [16], together with anonymized feature vectors and a test case graph.

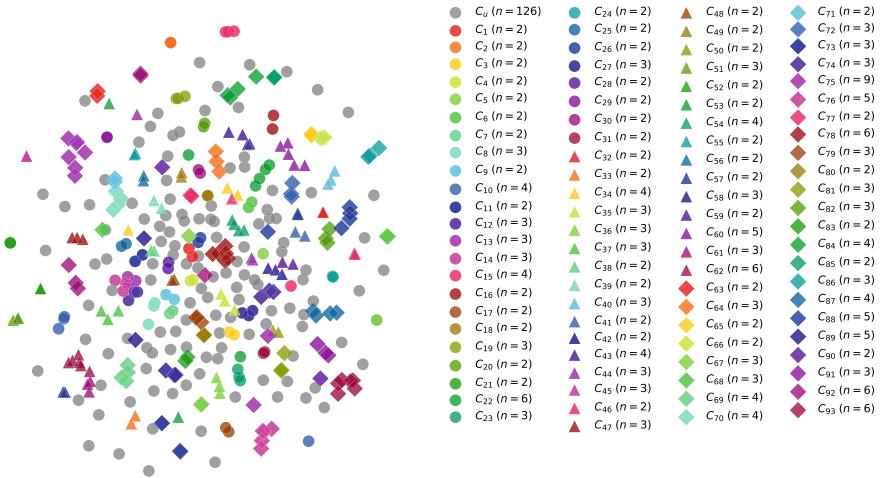


Fig. 8.4 The clustered test case using Doc2vec and HDBSCAN, where C_u represents non-clusterable data points and n indicates the size of each cluster

- Since the output of the Doc2vec is a large set of high-dimensional data, the HDBSCAN algorithm can be employed for clustering, which relaxes the pressure of dimensional reduction. However, if one wants to use other clustering algorithms, then the high-dimensional data (output from Doc2vec) might be processed by dimensionality reduction algorithms such as t-SNE. t-distributed stochastic neighbor embedding (t-SNE) is a statistical method for high-dimensional reduction that gives each datapoint a location in a two- or three-dimensional map. Listing 2 shows a Python code snippet to apply t-SNE.

```
1 def get_tsne(_values):
2     tsne = TSNE(n_components=2, random_state=0)
3     return tsne.fit_transform(_values)
```

Listing 2 Python script for t-SNE based dimensionality reduction

- As HDBSCAN utilizes an unsupervised learning approach, the algorithm itself decides on the number of clusters (including non-clusterable data points). However, the end user can decide the minimum number of data (test cases) for each cluster.

Listing 3 shows the Python code snippet to generate the graph shown in Fig. 8.4:

```
1 def clustered_graph(file_name, labels, label2cluster,
2 X_tsne):
3     sns.set_context('paper')
4     sns.set_style('white')
5     sns.set_color_codes()
6
7     plot_kwds={'alpha': 0.75, 's': 40, 'linewidths': 0}
8     fig = plt.figure()
```

```

8      fig.set_dpi(72)
9      fig.set_size_inches(6, 6)
10     ax = fig.add_subplot(111)

11
12     sns.despine(top=True, bottom=True, left=True,
13                  right=True)
13     ax.set_xticks([])
14     ax.set_yticks([])

15
16     maxcolors = ceil((max(label2cluster.values())+1)
17 /3)
17     pal = my_palette(maxcolors)

18
19     fltrdX = []
20     fltrdY = []
21     for i in range(0, len(labels)):
22         if label2cluster[labels[i]] == -1:
23             fltrdX.append(X_tsne[i, 0])
24             fltrdY.append(X_tsne[i, 1])
25     plt.scatter(fltrdX, fltrdY, c=mpl.colors.rgb2hex(
26 (0.5, 0.5, 0.5)), label="$C_u$ $(n={})$".format(len(
27 fltrdX)), **plot_kwds)

28
29     for cluster in range(0, max(label2cluster.values())
30 +1):
31         fltrdX = []
32         fltrdY = []
33         for i in range(0, len(labels)):
34             if label2cluster[labels[i]] == cluster:
35                 fltrdX.append(X_tsne[i, 0])
36                 fltrdY.append(X_tsne[i, 1])
37             marker = 'o'
38             if (cluster >= maxcolors):
39                 marker = '^'
40             if (cluster >= 2*maxcolors):
41                 marker = 'D'

42             plt.scatter(fltrdX, fltrdY, c=mpl.colors.
43 rgb2hex(pal[cluster%maxcolors]), marker=marker,
44 label="$C_{{{}{}}}$ $(n={})$".format(cluster+1, len(
45 fltrdX)), **plot_kwds)

46
47     lgd = plt.legend(bbox_to_anchor=(1.05, 1),
48 borderaxespad=0., prop = {"size": 6})
49     fig.tight_layout()
50     plt.savefig(file_name, format="pdf",
51 bbox_extra_artists=(lgd,), bbox_inches='tight')
52     plt.clf()

```

Listing 3 Python script for clustering graph

As mentioned before, in our case study in total 1,748 natural language test case specifications were analyzed. Figure 8.4 visualizes the initial results with a total of 93 clusters obtained from applying HDBSCAN. However, as one can see in Fig. 8.4

the size of each cluster is different. Note that with hyperparameter tuning we set the minimum cluster size to 2. Moreover, in total 126 test cases are detected as non-clustered data points (see C_u in Fig. 8.4), which represent the non-similar test cases. The source code of our work can be found online at [16], together with anonymized feature vectors and a test case graph.

8.4.6 Test Optimization Strategies

The results shown in Fig. 8.4 can be utilized in several ways and also for different test optimization purposes. Cluster-based test optimization strategies can be applied based on the company policies, resources, and constraints. In our context, the following cluster-based test optimization strategies were applied:

- *Test case selection*: For that purpose, one or more test cases are selected from each cluster. Since the grouped test cases inside of the obtained clusters are similar, some test cases can be selected for test execution purposes. However, to keep the test coverage and to avoid unnecessary failure, some parameters need to be checked, and some conditions need to be satisfied. For instance, the dependency between test cases [31], requirement coverage, and execution time [30] should be checked in advance. In this regard, a good test candidate from each cluster can be a test case that is independent or has the highest requirements coverage compared to other test cases of the same cluster. This strategy can also be used for the *test suite minimization* and also *test automation*, where some test cases from each cluster or even some clusters can be eliminated from the test suite. Moreover, since we are working with the manual test cases, some of the test cases from each cluster (or even some clusters) can be selected for the test automation. In fact, knowing similar test cases in advance can help the testing team to select some test specifications for test script generation. However, for the resulting clusters shown in Fig. 8.4, we recommend executing all non-similar test cases in cluster C_u as the setting is safety-critical.
- *Cluster prioritization*: The resulting clusters in Fig. 8.4 can also be ranked for execution. In this regard, those clusters which have a bigger size (see n in Fig. 8.4) can be high or low ranked for the execution. However, the mentioned constraints (dependency, requirement coverage) need to be evaluated in this strategy as well.
- *Functional dependency detection between test cases*: The entire proposed solution in this chapter can also be employed for detecting the functional dependencies between system integration test cases. Previously (see [33, 34]), we proved that two test cases can be functionally dependent on each other if there is a semantic similarity between their test specification. This hypothesis is evaluated several

times against a conducted ground truth at Bombardier Transportation. The utilized performance metrics (F1-score⁴ and AUC⁵) indicate promising results.

To sum up, we first pre-processed test specification data, then applied Doc2vec to encode the natural language text into vectors, and later applied clustering algorithms to cluster test cases into groups of similar test cases. These groups formed the basis for cluster-based test optimization.

8.5 Levels of Autonomy of AI in System Testing

The approaches to applying AI in the previous sections, i.e., search-based approaches, natural language processing, and machine learning, were applied by supporting human testers in performing testing activities. However, the AI did not operate autonomously but was triggered by humans and relied on their knowledge.

In this section, we reflect on the potential of autonomously operating AI for system testing. The term “autonomy” is generally used to mean the capacity of an artificial agent to operate independently of human guidance [19]. Agents or bots act autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. Software agents are called softbots or software robots, and it is a natural approach to use them in system testing to simulate real users or perform exploratory testing. Autonomous agents or softbots have already been used to automate different aspects of software development [10, 37, 38]. Several researchers have proposed different approaches for using agents specifically in software testing [9, 35], by considering different aspects that relate to test optimization. A recent study [22] suggests that autonomous agents are predominantly used at system-level testing functional, non-functional, and white-box testing.

Similar to the levels of autonomy in automotive engineering, we define and discuss several levels of autonomy when applying AI in system testing. Autonomy in itself can be described through two dimensions, self-sufficiency, i.e., ability to fulfill a task without outside help, and self-directness, i.e., the ability to decide upon one’s own goals as well as the involvement of an external human actor.

Overall, we can distinguish four levels of autonomy of AI in system testing⁶ from Level 0 to not apply AI at all to Level 3 of full autonomy. In the following, we describe each level in more detail and provide examples:

⁴ Is a measure of a model’s accuracy on a dataset.

⁵ The Area Under Curve provides an aggregate measure of performance across all possible classification thresholds.

⁶ Based on the levels shown by Synopsis <https://www.synopsys.com/automotive/autonomous-driving-levels.html>.

- Level 0—*AI is not applied*: System testing tasks are performed by humans or automated without AI. For example, test design at the system level can be automated using fuzzing, model-based testing, or combinatorial techniques [3].
- Level 1—*AI algorithms assist humans by performing (semi-)automate testing tasks*: AI algorithms support test analysis, design, execution, and evaluation activities, as described in the previous section for test case dependency detection and execution. Another such Level 1 approach is the adaptive test management system (ATMS) [25], which aims at selecting an appropriate set of test cases to be executed in every test cycle using test unit agents and fuzzy logic.
- Level 2—*AI replaces or mimics human behavior (e.g., agents that replace users)*: On this level not specific testing activities, but human behavior that is an integral part of testing is (partially) replaced by intelligent agents. Typical human behavior that is replaced is users of system under test or testers' performing exploratory testing. For instance, in [8], intelligent agents are applied for real-time testing of insider threat detection systems. As the number and variety of system interfaces increases, e.g., via image or voice recognition, the potential and need for the application of bots for system testing further increases. A different approach is presented in the work of Tang et al. [35]. Their study aims at automating the whole testing life cycle by using four types of agents: requirement agent, construct agent, execution, and report agent.
- Level 3—*System testing is done fully automated by AI agents*: This is the fully autonomous level, where system testing is performed fully automated by intelligent agents. Currently, this is just a vision and it is even not clear whether this is achievable both from a theoretical or practical point of view. Several approaches [5, 9] have shown some incipient but promising results on how this vision can be attained.

These levels of autonomy differ on how humans are involved and we can consider that this categorization is a continuum of autonomy rather than a dichotomy. Similar to this, Feldt et al. [12] used the Sheridan–Verplanck taxonomy to categorize the levels of AI automation based on decision and action selection performed by different actors. Nevertheless, more research is needed to characterize AI-infused system-level testing techniques that enable autonomous behavior.

8.6 Conclusion

This chapter presented where and how AI techniques can be applied to automate and optimize system testing. We first provided an overview of system testing, where testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Then, we identified different system testing activities (i.e., test planning and analysis, test design, test execution, and test evaluation) and indicated how AI techniques like optimization algorithms, natural language processing, and machine learning could be applied to automate and optimize these

activities. Furthermore, we presented an industrial case study on test case analysis. In the case study, natural language test cases are—based on natural language processing with Doc2vec and clustering—grouped into clusters of similar test cases that formed the basis for cluster-based test optimization. Finally, we discuss levels of autonomy of AI in system testing from Level 0 to not apply AI at all to Level 3 of full autonomy.

Acknowledgements This work was partially supported by the Austrian Science Fund (FWF): I 4701-N and the project ConTest funded by the Austrian Research Promotion Agency (FFG). Eduard Enoiu was partially supported by the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 957212.

References

1. A. Abran, J. Moore, P. Bourque, R. Dupuis, L. Tripp, Software engineering body of knowledge. IEEE Comput. Soc. (2004)
2. D. Adamo, M.K. Khan, S. Koppula, R. Bryce, Reinforcement learning for android GUI testing. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (2008), pp. 2–8
3. S. Anand, E. Burke, T.Y. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, P. McMinn, A. Bertolino et al., An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
4. A. Arcuri, Test suite generation with the many independent objective (MIO) algorithm. Inf. Softw. Technol. **104**, 195–206 (2018)
5. K. Baral, J. Offutt, F. Mulla, Self determination: a comprehensive strategy for making automated tests more effective and efficient, in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (IEEE, 2021), pp. 127–136
6. G. Bath, E. Van Veenendaal, Improving the Test Process: Implementing Improvement and Change-A Study Guide for the ISTQB Expert Level Module (Rocky Nook, Inc., 2013)
7. L.C. Briand, Y. Labiche, M. Shousha, Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genet. Program Evolvable Mach. **7**(2), 145–170 (2006)
8. P. Dutta, G. Ryan, A. Zieba, S. Stolfo, Simulated user bots: real time testing of insider threat detection systems, in *2018 IEEE Security and Privacy Workshops (SPW)* (IEEE, 2018), pp. 228–236
9. E. Enoiu, M. Frasher, Test agents: The next generation of test cases, in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (IEEE, 2019), pp. 305–308
10. L. Erlenhov, F.G. Oliveira Neto, R. Scandariato, P. Leitner, Current and future bots in software development, in *International Workshop on Bots in Software Engineering (BotSE)* (IEEE, 2019), pp. 7–11
11. M. Felderer, I. Schieferdecker, A taxonomy of risk-based testing. Int. J. Softw. Tools Technol. Transfer **16**(5), 559–568 (2014)
12. R. Feldt, F.G. Oliveira Neto, R. Torkar, Ways of applying artificial intelligence in software engineering, in *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)* (IEEE, 2018), pp. 35–41
13. K. Frounchi, L.C. Briand, L. Grady, Y. Labiche, R. Subramanyan, Automating image segmentation verification and validation by learning test oracles. Inf. Softw. Technol. **53**(12), 1337–1348 (2011)
14. V. Garousi, M. Felderer, Ç.M. Karapıçak, U. Yılmaz, Testing embedded software: a survey of the literature. Inf. Softw. Technol. **104**, 14–45 (2018)

15. M. Harman, P. McMinn, J.T. Souza, S. Yoo, Search based software engineering: techniques, taxonomy, tutorial, in *Empirical Software Engineering and Verification* (Springer, Berlin, 2010), pp. 1–59
16. L. Tahvili, S. Tahvili, Clustering dependency detection (2018). <https://github.com/leohatvani/clustering-dependency-detection>
17. IEEE: IEEE standard glossary of software engineering terminology. IEEE Std. 610.12-1990 (1990), pp. 1–84
18. J.Y. Jiang, M. Zhang, C. Li, M. Bendersky, N. Golbandi, M. Najork, Semantic text matching for long-form documents, in *WWW '19* (Association for Computing Machinery, New York, NY, USA, 2019)
19. M. Johnson, J. Bradshaw, P. Feltovich, C. Jonker, B. Van Riemsdijk, M. Sierhuis, The fundamental principle of coactive design: Interdependence must shape autonomy, in *International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems* (Springer, Berlin, 2010), pp. 172–191
20. A. Joshi, E. Fidalgo, E. Alegre, L. Fernández-Robles, Summcoder: an unsupervised framework for extractive text summarization based on deep auto-encoders. *Expert Syst. Appl.* **129**, 200–215 (2019)
21. M. Kane, Validating the interpretations and uses of test scores. *J. Educ. Meas.* **50**, 1–73 (2013)
22. P. Kumaresen, M. Frasher, E. Enoui, Agent-based software testing: a definition and systematic mapping study, in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (IEEE, 2020), pp. 24–31
23. D. Liang, F. Zhang, W. Zhang, Q. Zhang, J. Fu, M. Peng, T. Gui, X. Huang, Adaptive multi-attention network incorporating answer information for duplicate question detection, in *SIGIR'19* (Association for Computing Machinery, New York, NY, USA, 2019), pp. 95–104
24. L. van der Maaten, E. Postma, H. Herik, Dimensionality reduction: a comparative review. *J. Mach. Learn. Res.-JMLR* **10** (2007)
25. C. Malz, N. Jazdi, Agent-based test management for software system test, in *International Conference on Automation Quality and Testing Robotics (AQTR)*, vol. 2 (IEEE, 2010), pp. 1–6
26. M. Mansoor, Z. Rehman, M. Shaheen, M. Khan, M. Habib, Deep learning based semantic similarity detection using text data. *Inf. Technol. Control* **49** (2020)
27. I. Markov, H. Gómez-Adorno, J.P. Posadas-Durán, G. Sidorov, A. Gelbukh, Author profiling with doc2vec neural network-based document embeddings, in *Advances in Soft Computing*. ed. by O. Pichardo-Lagunas, S. Miranda-Jiménez (Springer International Publishing, Cham, 2017), pp.117–131
28. A.M. Memon, M.E. Pollack, M.L. Soffa, A planning-based approach to GUI testing, in *Proceedings of The 13th International Software/Internet Quality Week* (2000)
29. R. Ramler, M. Felderer, Requirements for integrating defect prediction and risk-based testing, in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (IEEE, 2016), pp. 359–362
30. S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, S.H. Ameerjan, Espret: a tool for execution time estimation of manual test cases. *J. Syst. Softw.* **161**, 1–43 (2018)
31. S. Tahvili, M. Bohlin, M. Saadatmand, S. Larsson, W. Afzal, D. Sundmark, Cost-benefit analysis of using dependency knowledge at integration testing, in *The 17th International Conference On Product-Focused Software Process Improvement* (2016)
32. S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, M. Bohlin, Automated functional dependency detection between test cases using doc2vec and clustering, in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)* (IEEE, 2019), pp. 19–26
33. S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, M. Saadatmand, M. Bohlin, Cluster-based test scheduling strategies using semantic relationships between test specifications, in *Proceedings of the 5th International Workshop on Requirements Engineering and Testing* (2018), pp. 1–4
34. S. Tahvili, L. Hatvani, E. Ramentol, R. Pimentel, W. Afzal, F. Herrera, A novel methodology to classify test cases using natural language processing and imbalanced learning. *Eng. Appl. Artif. Intell.* **95**, 1–13 (2020)

35. J. Tang, Towards automation in software test life cycle based on multi-agent, in *International Conference on Computational Intelligence and Software Engineering* (IEEE, 2010), pp. 1–4
36. M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.* **22**(5), 297–312 (2012)
37. M. Winikoff, Future directions for agent-based software engineering. *IJAOSE* **3**(4), 402–410 (2009)
38. M. Wooldridge, Agent-based software engineering. *IEE Proc.-Softw.* **144**(1), 26–37 (1997)

Chapter 9

Intelligent Software Maintenance



Foutse Khomh, Mohammad Masudur Rahman, and Antoine Barbez

Abstract Software systems are pervasive in our society. They play a vital role in our everyday life and are increasingly more and more complex. Software maintenance is one of the most important software development activities. It is estimated that more than 70% of the total cost of a software system is spent on maintenance activities. This cost can be reduced using Machine Learning (ML) and Artificial Intelligence (AI) in general. AI can be used in various stages of software maintenance, for example to identify design defects early on or code to reuse. We can analyse large amounts of data and identify patterns in a more efficient way by means of machine learning algorithms. In this chapter, we report about two studies that leverage (i) deep learning for code smell detection, and (ii) machine learning for reusable code search during software maintenance. We present the methodology that was followed and discuss the achieved results. We hope that practitioners and researchers reading the chapter will learn about the challenges and opportunities offered by AI, in the context of software maintenance, and that they will be able to successfully apply our proposed solutions to the context of their own software systems.

9.1 Introduction

Artificial Intelligence, in particular Machine Learning (ML), is increasingly being used in many industries, thanks to breakthroughs in deep learning and reinforcement learning. Software engineering is no exception. Software organisations and researchers are increasingly using ML techniques, to improve the efficiency of quality assurance activities, such as bugs [1, 2] and crash [3] prediction, test case prioritisation [4], and architecture improvement [5, 6], to name a few. In this chapter, we

F. Khomh (✉) · A. Barbez
Polytechnique Montreal, Montreal, Canada
e-mail: foutse.khomh@polymtl.ca

M. Masudur Rahman · A. Barbez
Dalhousie University, Halifax, Canada
e-mail: masud.rahman@dal.ca

report two case studies in which we applied ML techniques to improve the efficiency of a software maintenance task; i.e., design smell detection and code search. Design smells, also known as *anti-patterns*, are poor solutions to recurring design problems. There exists a variety of anti-patterns, which have been shown to negatively impact the maintainability [7] and comprehensibility [8] of the code. For example, the God Class anti-pattern, which happens when a class grows rapidly with the addition of new functionalities, violates the principle of single responsibility. It is characterised by low cohesion and high coupling. A variety of approaches have been proposed to detect the occurrences of anti-patterns in source code [9–11]. Most of them rely on the formal definitions of anti-patterns and attempt to identify their occurrences in source code using structural metrics (e.g., Lines Of Code) along with empirically defined thresholds. However, anti-patterns can also be detected by analysing the change history of a project [12]. Indeed, the presence of anti-patterns in a system influences how source code entities evolve with one another over time. For example, the Feature Envy anti-pattern, which happens when a method is implemented in the wrong class, can be detected by identifying methods that change more often with methods from another class than those of their own class.

Although structural and historical anti-pattern detection techniques have shown acceptable performances, they are still far from perfect (often reporting a large number of false positives and false negatives). Structural-based techniques like *InCode* [11], *DECOR* [9], *JDeodorant* [13], or *BDETEX* [14] often miss anti-pattern occurrences that can successfully be detected by techniques leveraging historical information (e.g., *HIST* [12]). Building on this observation, we proposed CAME (**C**onvolutional **A**nalysis of **c**ode **M**etrics **E**volution) [15], a deep learning-based approach to detect anti-patterns by analysing how source code metrics evolve over time when changes are applied to the system. The main idea behind this approach is to exploit the ability of deep neural networks to identify key features in raw data with the aim of detecting anti-patterns from both structural and historical information. Concretely, structural metric values related to the code components to be classified are computed at each revision of the system under investigation by mining its version control system (e.g., Git and SVN). This information is then organised into a 2D vector and fed through a Convolutional Neural Network (CNN) architecture to perform classification. We evaluated CAME on the widely known God Class anti-pattern using three software systems, and found it to significantly outperform existing anti-pattern detection approaches from the literature. CAME also outperformed existing static machine learning classifiers.

The second case study described in this chapter presents a novel technique—*NLP2API* [16]—that offers appropriate queries for reusable code search using machine learning. It automatically identifies relevant, specific API classes from the Stack Overflow Q&A site for a programming task written as a natural language query, and then reformulates the query to improve code search. In fact, software developers frequently issue generic natural language queries for code search while using code search engines (e.g., GitHub native search and Krugle). These queries often do not lead to any relevant results due to vocabulary mismatch problems, and the developers spend ≈19% of their programming time in various trials and errors [17]. *NLP2API*

expands a generic query with semantically relevant API classes using machine learning and overcomes vocabulary mismatch problems. By leveraging information from Stack Overflow about candidate API classes using pseudo-relevance feedback and two term weighting algorithms, and ranking the candidates using Borda count and the semantic proximity between query keywords and the API classes, NLP2API successfully outperformed state-of-the-art techniques [18–21] while improving the code search results provided by these contemporary search engines.

The remainder of this chapter is organised as follows. Section 9.3 presents the design and evaluation of CAME, while Sect. 9.4 describes NLP2API and reports its evaluation. Finally, Sect. 9.5 concludes the chapter.

9.2 Background

This section defines the concepts of Anti-patterns, Convolutional Neural Networks, and Regularisation.

9.2.1 *Anti-patterns*

Anti-patterns have been proposed to embody poor design choices. They identify “poor” solutions to recurring design problems and are generally introduced by developers not having sufficient knowledge and/or experience in solving a particular problem. In practice, anti-patterns are in between design and implementation, i.e., they concern the design of one or more classes, but concretely manifest themselves in the source code as classes through specific code smells. An example of an anti-pattern is the LazyClass, which occurs when a class does too little, i.e., has few responsibilities in a system. A LazyClass is a class with few methods and fields; its methods have little complexity. It often stems from speculative generality during a system design and/or implementation. Other examples of anti-patterns include God Class, Spaghetti Code, MessageChain, and Feature Envy, to name a few.

9.2.2 *Convolutional Neural Networks*

A Convolutional Neural Network (CNN) is a class of artificial neural networks that is designed to process data in the form of multiple arrays, such as 2D images and audio spectrograms, or 3D videos. A CNN contains the following specialised layers that transform the 3D input volume to a 3D output volume of neuron activations: Convolutional Layer, Activation Layer, and Pooling Layer. The Convolutional Layer provides a 2D feature map, where each unit is connected to local regions in the input data or the previous layer’s feature map through a multi-dimensional parameter called

a filter or kernel. The Activation Layer applies an activation function on the extracted feature map as an element-wise operation (i.e., per feature map output). The Pooling Layer ensures spatial pooling operation to reduce the dimensionality of each feature map and to retain the most relevant information by creating an invariance to small shifts and distortions.

9.2.3 Regularisation

Regularisation techniques have been developed to stabilise the training evolution of neural networks and prevent them from overfitting the training data.

The standard regularisation techniques consist in adding restrictions on the values of the trained parameters, i.e., by adding a penalty term $\Omega(W)$ in the loss function $loss(f^*, D)$ (see the regularised loss from Eq. 9.1) that can be seen as a soft constraint on the magnitude of parameters to restrict and smooth their corresponding distributions:

$$\tilde{loss}(W, b, D) = loss(W, b, D) + \lambda\Omega(W) \quad (9.1)$$

where λ fixes the relative contribution of the norm penalty $\Omega(W)$; so setting $\lambda = 0$ means no regularisation and larger values of λ correspond to more regularisation. The most popular penalty consists in penalising the weights of the linear computations, keeping their values closer to the origin by using L2-norm ($\Omega(W) = \frac{1}{2}\|W\|_2^2$) and/or enhancing the sparsity of the weights by using L1-norm ($\Omega(W) = \|W\|_1$).

9.3 CAME: Convolutional Analysis of code Metrics Evolution

This section presents CAME, our deep learning-based approach to detect anti-patterns from source code metrics historical information. We first describe the input of our model before describing its architecture. Finally, we discuss the procedure we followed to train this architecture for anti-pattern detection.

9.3.1 Input

To detect instances of a given anti-pattern in a system, our approach uses a deep learning-based classifier to perform a Boolean prediction on each code component (i.e., class or method) of the system. To perform this prediction, we exploit both structural and historical information. To define the input of our model, we first select a set of N_m structural metrics which can be computed for any code component of the system under investigation. Then, we walk through the history of revisions of the

system in reverse order by mining its repository. At each revision, we recalculate the values of the selected metrics for each component. We refer to the value of the j th metric computed for a component c at the i th revision of the system, as $m_{i,j}(c) \in \mathbb{R}$. Thus, for a given code component c to be classified, the input of our model is a real-valued matrix \mathbf{X}_c such as

$$\mathbf{X}_c(i, j) = m_{i,j}(c) \quad (9.2)$$

Different software systems experience a different number of revisions (i.e., commits). Also, a code component may have been introduced in a system during its creation or on the contrary during a recent revision. As a consequence, different code components may be characterised by a different number of revisions. To allow our model to receive a fixed size input, we limit the length of the metrics history (i.e., number of revisions considered for a given component) to a constant L_h , which is a hyper-parameter to be adjusted. Hence, each input matrix of our model is of shape $L_h \times N_m$. If a component has a history shorter than L_h , meaning that the component did not exist in the early revisions of the system, we pad the rest of its input matrix with zeros.

9.3.2 Architecture

Figure 9.1 overviews the architecture of the convolutional neural network used by our approach to perform classification. The model contains several convolution + pooling layers fully connected to a Multi-Layer Perceptron (MLP) model, i.e., several dense layers connected to an output layer. The convolution layers perform 1D convolutions with a *stride* of 1 and a *tanh* activation function. The output of each convolution layer is fed to max-pooling layers that reduce the dimensionality of their input by taking the maximum value across a small spatial region. Then, after being flattened, the output of the last pooling layer is fed to *tanh* dense layers. Finally, the output layer is made of one single *sigmoid* neuron which outputs the predicted probability that a component contains an anti-pattern given its input matrix.

The choice of such architecture has been motivated by the ability of CNNs to extract high-level features from high-dimensional data, e.g., deep CNNs used in image processing recognise complex shapes in raw pixels. Indeed, our conjecture is that the first convolution layer can detect that some metrics have changed between two consecutive commits. Then, the next layers build a representation of the changes that characterise an anti-pattern. Therefore, our model identifies recurrent patterns in the local variations of software metrics, which we believe constitute useful complementary information for detecting anti-patterns.

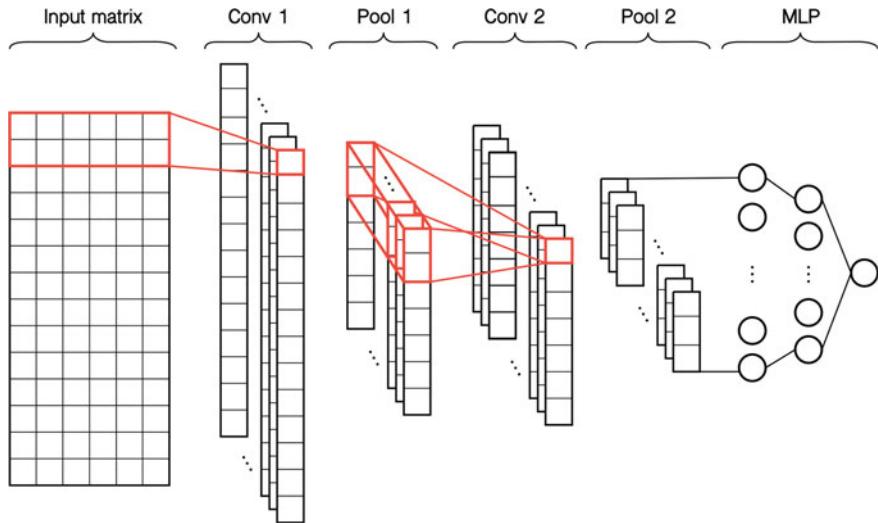


Fig. 9.1 Architecture of the CNN model used by CAME. In this example, the network contains two convolution + polling layers with filters of size 2 and two dense hidden layers

9.3.3 Training

This subsection discusses the considerations adopted to train neural networks on the task of anti-pattern detection.

Let $\mathcal{D} = \{(\mathbf{X}_i, y_i)\}_{i=1}^n$ be our training set. With $\mathbf{X}_i \in \mathbb{R}^{L_h \times N_m}$, being the input matrix (cf. Eq. 9.2) of the i th component to classify, $y_i \in \{0, 1\}$ the true label for this component and n the number of instances in the training set.

Also, we refer to the set of weights of our model as $\boldsymbol{\theta} = \{\mathbf{w}_l\}_{l=1}^L$, with \mathbf{w}_l being the weight matrix of the l th layer and L the number of layers in the network. Finally, we denote by $P_{\boldsymbol{\theta}}(1|\mathbf{X}_i) = g(\mathbf{X}_i, \boldsymbol{\theta})$ the predicted probability outputted by our model for the i th component, with $g(x) = \frac{1}{1+e^{-x}}$ the *sigmoid* function.

9.3.3.1 Loss Function

In a software system, components affected by an anti-pattern are usually a minority ($\approx 1\%$) [22]. Classifiers optimised using conventional loss functions, e.g., *cross entropy*, on such data tend to favour the majority category, thus maximising the overall accuracy [23]. This characteristic known as the “imbalanced data problem” prevents us from using such loss function to guide the optimisation of our model. Hence, we must define a loss function that maximises our evaluation metric: the F-measure expressed in Eq. 9.10.

Optimising a model through gradient descent requires computing the gradient of the loss with respect to the model weights. However, computing the number of true

positives TP and positives m_{pos} (cf. Table 9.2) requires counting elements from the probability outputted by the model, which necessarily involves discontinuous operators:

$$TP(\boldsymbol{\theta}, \mathcal{D}) = \sum_{\substack{i=1 \\ y_i=+1}}^n \delta(P_{\boldsymbol{\theta}}(1|\mathbf{X}_i) > 0.5) \quad (9.3)$$

$$m_{pos}(\boldsymbol{\theta}, \mathcal{D}) = \sum_{i=1}^n \delta(P_{\boldsymbol{\theta}}(1|\mathbf{X}_i) > 0.5) \quad (9.4)$$

where

$$\delta(x) = \begin{cases} 1 & \text{if } x = True \\ 0 & \text{if } x = False \end{cases}$$

This characteristic prevents us from using the F-measure directly to define our loss function. Consequently, we use the continuous and differentiable approximation of the F-measure provided by Jansche [24] which consists in considering the limit:

$$\delta(P_{\boldsymbol{\theta}}(1|\mathbf{x}_i) > 0.5) = \lim_{\gamma \rightarrow +\infty} g(\gamma \mathbf{X}_i \cdot \boldsymbol{\theta}) \quad (9.5)$$

Hence, we define our loss function as $loss = -\tilde{F}_m(\boldsymbol{\theta}, \mathcal{D})$ with

$$\tilde{F}_m(\boldsymbol{\theta}, \mathcal{D}) = 2 \times \frac{\sum_{\substack{i=1 \\ y_i=+1}}^n g(\gamma \mathbf{X}_i \cdot \boldsymbol{\theta})}{n_{pos} + \sum_{i=1}^n g(\gamma \mathbf{X}_i \cdot \boldsymbol{\theta})} \quad (9.6)$$

Note that the value of the hyper-parameter γ will be adjusted along with other hyper-parameters during the tuning of our model.

9.3.3.2 Regularisation

Overfitting occurs when a statistical model fails to generalise to new examples by learning irrelevant characteristics of its training data. Hence, an overfitted model performs well on the training set but achieves poor performance on unseen test data. To prevent our model from overfitting, we used the widely adopted L_2 regularisation technique, which consists in adding a term to the loss function to encourage the weights to be small [25]. This term relies on the Euclidean norm of the weight matrices, i.e., $\|\mathbf{w}\|_2 = \sqrt{\mathbf{w}^\top \mathbf{w}}$, also called L_2 -norm. Thus, the L_2 regularisation term added to the loss function can be expressed as

$$L_2 = \lambda \sum_{l=1}^{L+1} \|\mathbf{w}_l\|_2 \quad (9.7)$$

where $\lambda \in \mathbb{R}$ is a hyper-parameter adjusted during cross-validation.

9.3.4 Evaluation of CAME

In this section, we assess the effectiveness of CAME at detecting the God Class anti-pattern. We answer the following three research questions:

- **To what extent historical values of source code metrics can improve detection performances?**

This research question assesses the impact of the length of the input metrics history (i.e., L_h) on our models' performances. Hence, before comparing our approach with other detection techniques, we can confirm that metrics history provides relevant information to our model and establishes which value of L_h leads to optimal performances.

- **How does CAME compare to other static ML algorithms?**

This research question aims at comparing the performances of our approach with other static ML algorithms, i.e., which rely on one single revision of the systems.

- **How does CAME compare to existing detection techniques?**

This research question aims at comparing our approach with existing detection techniques. The results of this comparison will provide insights into the advantages of using CAME instead of other detection tools to help developers in their daily maintenance tasks.

All the information and material necessary for replicating the results of this evaluation is available online¹ at <https://github.com/antoineBarbez/CAME>.

9.3.4.1 Experimental Dataset

To answer our research questions, we base our study on eight open-source Java projects belonging to various ecosystems. Android Opt Telephony and Android Support belong to the Android APIs.² Apache Ant, Apache Tomcat, Apache Lucene, and Apache Xerces belong to the Apache Foundation.³ ArgoUML⁴ is a software design tool and Jedit⁵ a text editor. For the sake of simplicity, we chose to analyse only the directories that implement the core features of the systems and to ignore test directories.

The choice of these systems has been motivated by the fact that they have been used for a similar purpose in prior studies. Indeed, to train our model and compare its performances with those of other approaches, we needed an oracle reporting the occurrences of God Classes in a set of software systems. Unfortunately, we found no

¹ Editors note: The source code linked in this chapter could change after publishing this book. A snapshot of the source code accompanying this chapter can be found at <https://doi.org/10.5281/zenodo.6965479>.

² <https://android.googlesource.com/>.

³ <https://www.apache.org/>.

⁴ <http://argouml.tigris.org/>.

⁵ <http://www.jedit.org/>.

Table 9.1 Characteristics of the studied systems

System name	Snapshot	Directory	#Files	#God classes
Android Opt Telephony	c241cad	src/java/	190	10
Android Support	38fc0cf	v4/	104	4
Apache Ant	e7734de	src/main/	755	7
Apache Lucene	39f6dc1	src/java/	160	3
Apache Tomcat	398ca7ee	java/org/	1005	5
Apache Xerces	c986230	src/	658	15
ArgoUML	6edc166	src_new/	1246	22
Jedit	e343491	./	437	5

such large dataset in the literature. Consequently, we reused the occurrences of God Class used to evaluate the approaches HIST [12] and DECOR [9], made publicly available^{6,7} by their respective authors. Among the systems available, we kept only those for which the full history was available through Git or SVN. Also, for some systems, we found occurrences that do not belong to it or that do not exist in the current revision. In such cases, we did not incorporate the systems in our oracle. Table 9.1 overviews the main characteristics of the subject systems.

9.3.4.2 Source Code Metrics

For God Class detection, the components to classify correspond to the classes of the system. However, we chose to consider only top-level classes as potential God Classes, as we found no inner classes positively labelled in our data. To decide whether or not a given class is affected by the God Class anti-pattern, we retrieve the history of **seven** structural metrics. Note that to compute these metrics, we do not consider attributes and methods of inner- (or nested-) classes as components of the class under investigation. Indeed, the refactoring operation commonly applied to remove God Classes (Extract Class Refactoring) consists in identifying one or several groups of attributes and methods of the class dedicated to one functionality and then extract them as a separate class. Thus, we assume that inner classes may be the result of such refactoring. In the following, we define each selected metric and provide a brief rationale for their use.

- **ATFD** (Access To Foreign Data): Number of distinct attributes of unrelated classes (i.e., not inner- or super-classes) accessed (directly or via accessor methods) in the body of a class. A God Class accesses a lot of data from other classes as suggested by Lanza and Marinescu [26].

⁶ <http://www.rcost.unisannio.it/mdipenta/papers/ase2013/>.

⁷ <http://www.ptidej.net/tools/designsmells/materials/>.

- **LCOM5** (Lack of COhesion in Methods): Measures cohesion among methods of a class based on the attributes accessed by each method [27]. A God Class handles many unrelated functionalities, thus methods related to different functionalities access different sets of attributes. This metric is also part of the detection process of DECOR [9].
- **LOC** (Lines Of Code): Sum of the number of lines of code of all methods of a class. A God Class implements a high number of functionalities, thus it is mainly characterised by its size.
- **NAD** (Number of Attributes Declared): Number of attributes declared in the body of a class. This metric is part of the detection process of DECOR [9].
- **NADC** (Number of Associated Data Classes): Number of dependencies with data-classes (i.e., data holders without complex functionality other than providing access to their data). This metric is part of the detection process of DECOR [9].
- **NMD** (Number of Methods Declared): Number of non-constructor and non-accessor methods declared in the body of a class. This metric is part of the detection process of DECOR [9].
- **WMC** (Weighted Method Count): Sum of McCabe's cyclomatic complexity [28] of all methods of a class. A God Class has a high functional complexity, as suggested by Lanza and Marinescu [26].

9.3.4.3 Data Extraction

As previously evoked, we construct the input matrices of our model by mining software repositories and navigating through the different revisions of a system using its version control API (e.g., Git). To automate this process and allow replication, extension, and reuse of our work, we designed a component called `RepositoryMiner`⁸ which automatically extracts the source code metrics history of any java software system. The `RepositoryMiner` currently implements 12 class- and method-related structural metrics and allows to easily define new ones. We briefly describe each step of the process followed to extract our data below.

Input: The `RepositoryMiner` takes as input three arguments: (1) the URL of the system's repository; (2) the SHA, i.e., the identification number, of the system's snapshot (i.e., commit) we want to analyse; and (3) the sub-directories of interest, i.e., those in which we want to detect affected components.

Initialisation After downloading the system repository, we check out to the current snapshot and parse the Abstract Syntax Tree (AST) of all the `.java` files contained in the sub-directories of interest. This step creates a `SystemObject` that holds a representation of the system's classes which will be used later to compute our metrics.

⁸ <https://github.com/antoineBarbez/RepositoryMiner/>.

Data Extraction We walk through all the commits of the system in reverse order starting from the current snapshot. At each commit, we perform the following steps:

1. Retrieve the names of all the files that have been changed between the current and the previous revision.
2. If the changed files belong to the `SystemObject`, checkout and update the corresponding classes.
3. Check if any component (i.e., file, class, or method) has been renamed between the current and the next revision.
4. Compute the code metrics values for each class of the `SystemObject` by taking into account renamed components.

Output The `RepositoryMiner` outputs a collection of `.csv` metric files. Each file contains the code metrics values computed for each class of the system at a given revision. Note that before creating a new metric file, we check if at least one value has been modified with respect to the previous one.

9.3.4.4 Performance Metrics

To evaluate the performances of CAME as well as those of the competing classifiers, we selected three systems, i.e., Android Support, Apache Tomcat, and Jedit, among the eight software systems presented in Table 9.1. These systems have been selected for the sake of generalizability. Indeed, they belong to different domains: telephony framework, service container, and text editor and have different sizes (i.e., number of classes). The remaining five systems are used for training and tuning the hyper-parameters of the different ML-based approaches investigated in this work.

We compute the overall performances of each approach by running it on all instances (i.e., the java classes) of the three test systems. Indeed, each classifier is able to perform a Boolean prediction on every single instance. Then, we evaluate a classifier using the so-produced confusion matrix presented in Table 9.2.

With TP the number of true positives, FN the number of false negatives (i.e., misses), FP the number of false positives, and TN the number of true negatives, we use this matrix to compute our evaluation metrics:

Table 9.2 Confusion matrix for binary classification

		Predicted label		Total
		1	0	
True label	1	TP	FN	n_{pos}
	0	FP	TN	n_{neg}
Total		m_{pos}	m_{neg}	n

$$precision = \frac{TP}{TP + FP} \quad (9.8)$$

$$recall = \frac{TP}{TP + FN} \quad (9.9)$$

In order to evaluate each approach with a single aggregated metric, we also compute the F-measure (i.e., the harmonic mean of precision and recall):

$$F_m = 2 \times \frac{precision \times recall}{precision + recall} = 2 \times \frac{TP}{n_{pos} + m_{pos}} \quad (9.10)$$

9.3.5 Impact of Metrics History's Length on the Performance of CAME

In this section, we report the results of our experiments conducted with the aim of assessing the impact of the metrics history length (L_h) on the performances of our approach. Hence, this section answers the first research question.

9.3.5.1 Method

To answer **RQ1**, we monitor the performances achieved by CAME, in terms of precision, recall, and F-measure, with different lengths of metrics history: $L_h \in \{1, 10, 50, 100, 250, 500, 1000\}$. For each value of L_h experimented, we build and train 10 distinct CNNs in order to retrieve the mean and standard deviation of the three performance metrics achieved for each length. To avoid any bias in our conclusions, we must consider that the length of the metrics history may affect the optimal values of the other hyper-parameters of our model. Consequently, before training our model for a new value of L_h , we perform a new tuning of its hyper-parameters. As explained in Sect. 9.3.4.4, the performance metrics are computed by considering instances of the three test systems together: Apache Tomcat, Jedit, and Android Support. The remaining five systems are kept for training and hyper-parameter tuning.

9.3.5.2 Hyper-Parameters Tuning

We select the optimal set of hyper-parameters of our model for each history length investigated using a random search over 100 generations of nine hyper-parameters [29]. Table 9.3 reports for each hyper-parameter the range of values experimented. We monitor the performances achieved by our model with different sets of hyper-parameters by performing a 5-fold cross-validation. Thus, we first split the training set into 5 equal-size partitions, i.e., folds. Then, at each iteration, we generate a new random set of hyper-parameters and compute our model's prediction on each fold by leaving it out while keeping the others for training (100 epochs).

Table 9.3 CAME hyper-parameter tuning

Hyper-parameter	Range
Learning Rate (η)	$10^{-[0.0;2.5]}$
L2-norm (λ)	$10^{-[0.0;2.5]}$
Gamma (γ)	[1; 10]
# Conv Layers	[0, 1] if $L_h \leq 10$ else [1; 2] if $L_h \leq 100$ else 2
# Filters	[10; 60]
Filter size	[2; 4]
Pool size	{2, 5, 10} if $L_h \leq 100$ else {5, 10, 15, 20}
# Dense Layers	[1; 3]
Dense Layer size	[4; 100] then [4; s]

With s the size of the previous dense layer

Finally, we concatenate the obtained predictions and compute the overall F-measure. The optimal values retained after tuning for each history length can be found in [15].

For each history length investigated, we train our model using the previously found hyper-parameter values. We perform a mini-batch-based stochastic gradient descent optimisation during 300 epochs, with 5 mini-batches and an exponential learning rate decay of .5 every 100 epochs. It is important to remember that we train 10 randomly initialised CNNs per history length investigated in order to compute the mean and standard deviation of their performance metrics.

9.3.5.3 Results

Figure 9.2 presents the results of our comparison for God Class, i.e., mean values and standard deviations of CAME’s performance metrics over the three test systems, obtained using different sizes of metrics history (1, 10, 50, 100, 250, 500, 1000).

RQ1: To what extent historical values of source code metrics can improve detection performances?

For God Class detection, our results show that in terms of F-measure, the overall performances achieved by CAME on the three test systems significantly increase with the size of the metrics history. Specifically, after training our model ten times per history length, we observe that the overall F-measure improves from 0.51 ± 0.06 for $L_h = 1$ to 0.68 ± 0.08 for $L_h = 500$ (improvement of $33\% \pm 6\%$). Regarding the other two performance metrics, we see that although CAME achieves a better recall on average for $L_h > 1$, there does not seem to be any real correlation between recall and history length. As shown in Fig. 9.2, the recall strongly increases in the range [1; 100] but then drops to recover its initial value at $L_h = 1000$. Finally, we can see that the precision clearly improves with the history length similar to the F-measure. By comparing the precision for $L_h = 1$ with respect to the history length that led to the best results ($L_h = 500$), we observe an overall improvement of $61\% \pm 11\%$.

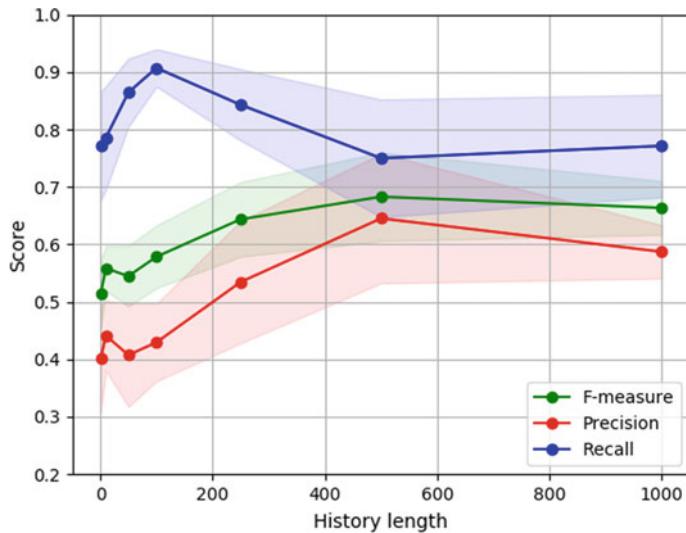


Fig. 9.2 Comparison of CAME’s performances for different sizes of metrics history. The lines show mean values while areas show standard deviations over 10 trainings

Also, we can see that the performances of CAME (especially precision) decrease for $L_h > 500$. This observation is surprising considering that a longer metrics history includes the shorter ones and thus should lead to a greater or equal F-measure. We believe this could be due to a sub-optimal choice of hyper-parameters. Indeed, the randomness of the process and the finite number of combinations experimented make it hard to conclude that the hyper-parameters selected are optimal. Furthermore, while tuning our model, we limited the number of convolution layers to 2. Hence, it is possible that more convolution layers may be needed to process a history of size $L_h = 1000$ and achieve optimal performances.

Our results confirm that our model properly leverages historical information about source code metrics by decreasing the number of false positives. Specifically, for $L_h > 1$ we observe a recall greater or equal to that obtained with a single revision of the input metrics and more importantly, we see that the precision clearly increases with the length of the input metrics history.

9.3.6 Comparison of CAME with Existing Techniques

This section reports the results of our experiments aiming to compare CAME with other approaches. To avoid redundancies, we report the results for our last two research questions together.

9.3.6.1 Method

Similar to the previous study, we evaluate the performances of the different approaches on our three test systems (i.e., Apache Tomcat, Jedit, and Android Support) while keeping the other five for training and hyper-parameter calibration. To compute the performances of CAME on these systems, we reuse the ten CNNs trained during our previous study (cf. Sect. 9.3.5.1) with $L_h = 500$. As shown in Fig. 9.2, after being trained, each CNN achieves different performances due to the randomness of its initialisation. Hence, the final performances of CAME are computed from the outputs of the ten CNNs using an ensemble method, also known as *boosting technique*. Such methods have been shown to lead to better performances than those of each independent classifier [30]. In the context of this study, we use the widely adopted Bayesian averaging heuristic to compute the ensemble prediction. Thus, the final predicted probability that a class c is a God Class can be expressed as

$$P_{ensemble}(1|\mathbf{X}_c) = \frac{\sum_{i=1}^{10} P_i(1|\mathbf{X}_c)}{10} \quad (9.11)$$

with $P_i(1|\mathbf{X}_c)$, the predicted conditional probability given by the i th CNN and \mathbf{X}_c the input matrix of the class c .

To answer **RQ2**, we compare CAME with three ML classifiers: Decision Tree, Multi-Layer Perceptron (MLP), and Support Vector Machine (SVM). The input of these classifiers consists in the same seven metrics used by CAME (cf. Sect. 9.3.4.2) computed on the current revision of each studied system. Similar to CAME, we first calibrate the hyper-parameters of each classifier and compute the final performances using the Bayesian averaging ensemble method on ten pre-trained models.

To answer **RQ3**, we compare CAME with three state-of-the-art detection approaches: Two static code analysis techniques DECOR [9] and JDeodorant [13], and one approach that exploits change history information to detect anti-patterns, HIST [12]. We chose to compare CAME with these approaches to increase the scope of our study. Indeed, they rely on radically different strategies to detect anti-patterns and are thus likely to be complementary. DECOR relies on the use of *Rule Cards* that encode the formal definitions of anti-patterns using structural and lexical information. JDeodorant detects affected components by identifying refactoring opportunities. Finally, HIST exploits change history information derived from version control systems. To compute the performances of each approach, we used the implementations made publicly available by their respective authors whenever

Table 9.4 Hyper-parameter tuning of the competing ML classifiers

Model	Hyper-parameter	Range
Decision tree	Max Features	{sqrt, log2, None}
	Max Depth	$10 \times [1; 10]$
	Min Sample Leaf	{1, 2, 4, 6}
	Min Sample Split	{2, 5, 10, 15}
MLP	Learning Rate (η)	$10^{-[0.0; 2.5]}$
	L2-norm (λ)	$10^{-[0.0; 2.5]}$
	Gamma (γ)	[1; 10]
	# Dense Layers	[1; 3]
	Dense Layer size	[4; 100] then [4; s]
SVM	Penalty	{0.001, 0.01, 0.1, 1, 10, 100}
	Gamma	{0.001, 0.01, 0.1, 1, 10, 100}
	Kernel	{linear, rbf, sigmoid}

With s the size of the previous dense layer

possible and replicated the approaches for which no implementation was available. Thus, we ran DECOR using the Ptidej API⁹ and JDeodorant using its Eclipse plugin.¹⁰ For HIST, we implemented the detection rules as described in its original paper [12]. Also, we implemented our own component¹¹ to extract code changes at a class-level granularity because the original component was not available due to its license.

9.3.6.2 Hyper-Parameter Tuning

We calibrate the hyper-parameters of each ML algorithm using the same procedure adopted for CAME in the previous study (see Sect. 9.3.5.2). Hence, we use a random search of 100 iterations over a variety of hyper-parameters related to each classifier. Also, to monitor the performances induced by each set of hyper-parameters, we use a 5-fold cross-validation with instances of the five training systems. Table 9.4 reports for each classifier, the set of hyper-parameters tuned, as well as the ranges of values experimented. To train the MLP model, we used rigorously the same procedure followed for training CAME.

9.3.6.3 Results

Table 9.5 reports the performances for God Class detection, in terms of precision, recall, and F-measure, achieved by CAME along with those of the competing tech-

⁹ <https://github.com/ptidejteam/v5.2/>.

¹⁰ <https://marketplace.eclipse.org/content/jdeodorant/>.

¹¹ <https://github.com/antoineBarbez/HistoryExtractor>.

Table 9.5 Performances for God Class detection

Approaches	Apache tomcat			JEdit			Android platform support			Overall		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
DECOR	68%	40%	50%	17%	60%	26%	—	0%	—	24%	36%	29%
HIST	—	0%	—	25%	40%	31%	18%	100%	31%	20%	43%	27%
JDeodorant	2%	60%	5%	5%	60%	9%	50%	50%	50%	4%	57%	8%
Decision tree	33%	20%	25%	100%	40%	57%	100%	25%	40%	68%	29%	40%
MLP	25%	100%	40%	68%	80%	73%	100%	75%	86%	41%	86%	56%
SVM	50%	20%	29%	100%	20%	33%	—	0%	—	68%	14%	24%
CAME	50%	100%	68%	100%	80%	89%	100%	75%	86%	71%	86%	77%

niques. The performances are reported for each subject system and overall, i.e., considering the three test systems as a single one. When an approach did not detect any occurrence of God Class in a system, it was not possible to compute the precision or the F-measure. In these cases, a “–” is indicated in the corresponding cell.

RQ2: How does CAME compare to other static ML algorithms?

For God Class detection, our results show that overall, our approach significantly outperforms the three classifiers. In terms of F-measure, the performances improve from 56% to 77% (improvement of 38%) with respect to the classifier with the best performances (the MLP). Unsurprisingly, CAME achieves the same recall as the MLP. Indeed, it is interesting to remark that a MLP model is equivalent to the CNN used by CAME with no convolution layers and considering $L_h = 1$. Hence, we make the same observation as in the previous study regarding the recall. Considering the other two classifiers, we clearly see that they cannot compete with CAME in terms of recall with 29% for the Decision Tree algorithm and 14% for the SVM against 86% for our approach. Finally, overall, CAME ensured a better precision than any other classifier (+ 18% on average). Regarding the performances achieved on each system, we see that CAME achieves the highest precision (100%) on two of the three test systems but only 50% on Apache Tomcat. This may be due to the fact that this system is the largest of our training set but contains only five occurrences of God Class. Hence, any model is more likely to have a low precision on it. Finally, our approach seems to have a more stable recall with values ranging from 75% to 100%.

CAME significantly outperforms other static ML classifiers. Indeed, none of the competing algorithms performs better than our approach on any system and considering any performance metric.

RQ3: How does CAME compare to existing detection techniques?

Our results show that CAME clearly outperforms existing detection methods in detecting the God Class anti-pattern. Indeed, CAME shows, overall, a precision of 71% and a recall of 86% (F-measure of 77%) over the test systems. With respect to the tool that performs the best for each performance metric, we see that CAME improves the precision by 196%, the recall by 51%, and the F-measure by 166%. Also, as we can see, each tool achieves poor performances on at least one system, which is not the case for our approach. This confirms that CAME performs well independently of the system characteristics.

However, some factors can explain the poor performances reported for some of the approaches experimented with. First, as the original implementation of HIST is not publicly available, we had to replicate this approach from the directives given by the authors in the paper [12]. We are aware that some differences in our respective implementations may have impacted the performances reported. Second, it is important to note that JDeodorant is not, strictly speaking, an anti-pattern detection tool. Instead, JDeodorant suggests opportunities to apply refactoring operations in

the system. Hence, it is not surprising that it achieves a high recall at the expense of its precision, because there may exist a high number of classes in the subject systems that could benefit from an Extract Class Refactoring operation without necessarily being God Classes.

In general, CAME significantly outperforms all the existing techniques investigated in this work for God Class detection. This suggests that our approach should be considered by practitioners for identifying affected code components to be refactored.

9.4 NLP2API: A Query Reformulation Technique for Improved Code Search

This section presents NLP2API, our machine learning-based approach to reformulate queries for reusable code search on the Internet. NLP2API is comprised of 13 steps depicted in Fig. 9.3. It leverages information from Stack Overflow about candidate API classes using pseudo-relevance feedback and two term weighting algorithms, and ranks the candidates using Borda count and the semantic proximity between query keywords and the API classes. Algorithm 1 provides a pseudo-code description of the proposed approach. In the following, we first describe candidate API selection and API relevance estimation using Borda count and semantic proximity analysis. Then we conduct experiments to demonstrate the benefit of NLP2API over several existing alternatives.

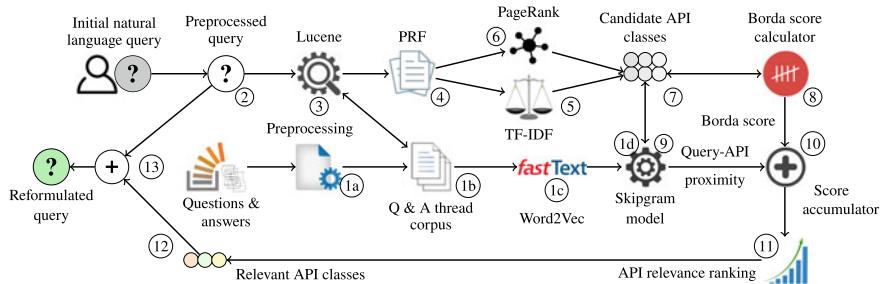


Fig. 9.3 Schematic diagram of the proposed query reformulation technique—NLP2API

9.4.1 Development of Candidate API Lists

We collect candidate API classes from the Stack Overflow Q&A site to reformulate a generic query (i.e., Fig. 9.3, Steps 1a, 1b, 2–7). Stack Overflow is a large body of crowd knowledge with 14 million questions and 22 million answers across multiple programming languages and domains [31]. Hence, it might contain at least a few questions and answers related to any programming task at hand. Earlier studies from the literature [20, 31, 32] also strongly support this conjecture. Given that relevant program elements are a better choice than generic natural language terms for code search [33], we collect API classes as candidates for query reformulation by mining the programming Q&A threads of Stack Overflow.

Corpus Preparation: We collect a total of 656,538 Q&A threads related to Java (i.e., using `<java>` tag) from Stack Overflow for corpus preparation (Fig. 9.3, Steps 1a, 1b, Algorithm 1, Line 3). We use the public data dump [34] released in March 2018 for data collection. Since we are mostly interested in the API classes discussed in the Q&A texts, we adopt certain restrictions. First, we make sure that each question or answer contains a bit of code, i.e., the thread is about coding. For this, we check the existence of `<code>` tags in their texts like the earlier studies [35–38]. Second, to ensure high-quality content, we chose only such Q&A threads where the answer was accepted as a solution by the person who submitted the question [20, 39]. Once the Q&A threads are collected, we perform standard natural language preprocessing (i.e., removal of stop words, punctuation marks and programming keywords, and token splitting) on each thread, and normalise their contents. Given the controversial evidence on the impact of stemming on source code [40], we avoid stemming on these threads given that they contain code segments. Our corpus is then indexed using *Lucene*, a widely used search engine by the literature [41–43], and later used for collecting feedback on a generic natural language query.

Pseudo-Relevance Feedback (PRF) on the NL Query: Nie et al. [20] first employ Stack Overflow in collecting pseudo-relevance feedback on a given query. Their idea was to extract software-specific words relevant to a given query, and then use them for query reformulation. Similarly, we also collect pseudo-relevance feedback on the query using Stack Overflow. We first normalise a natural language query using standard natural language preprocessing (i.e., stop word removal, token splitting), and then use it to retrieve Top-M (e.g., $M = 35$, check RQ₁ for detailed justification) Q&A threads from the above corpus with *Lucene* search engine (i.e., Fig. 9.3, Steps 2–4, Algorithm 1, Lines 4–8). The baseline idea is to extract appropriate API classes from them using appropriate selection methods [44], and, then, to use them for query reformulation. We thus extract the program elements (e.g., code segments, API classes) from each of the threads by analysing their HTML contents. We use *Jsoup* [45], a Java library for HTML scraping. We also develop two separate sets of code segments from the questions and answers of the feedback threads. Then we use two widely used term-weighting methods—*TF-IDF* and *PageRank*—for collecting candidate API classes from them.

Listing 9.1 An example code snippet for the programming task—“Convert image to grayscale without losing transparency”, (taken from [46])

```
BufferedImage master = ImageIO.read(new URL (
    "http://www.java2s.com/style/download.png"));
BufferedImage gray = new BufferedImage(master.getWidth (),
    master.getHeight(), BufferedImage.TYPE_INT_ARGB);

ColorConvertOp op = new ColorConvertOp (
    ColorSpace.getInstance(ColorSpace.CS_GRAY), null);
op.filter(master, gray);

ImageIO.write(master, "png", new File("path/to/master"));
ImageIO.write(gray, "png", new File("path/to/gray/image"));
```

API Class Weight Estimation with TF-IDF: Existing studies [20, 42, 47] often apply Rocchio’s method [48] for query reformulation where they use TF-IDF to select appropriate expansion terms. Similarly, we adopt TF-IDF for selecting potential reformulation candidates from the code segments that were collected above. In particular, we extract all API classes from each code segment (i.e., feedback document) with the help of island parsing (i.e., uses regular expressions) [49], and then determine their relative weight (i.e., Fig. 9.3, Step 5, Algorithm 1, Lines 11–12) as follows:

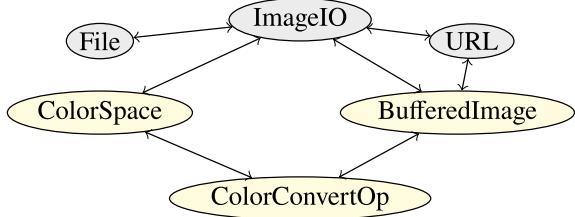
$$TF - IDF(A_i) = (1 + \log(TF_{A_i})) \times \log\left(1 + \frac{N}{DF_{A_i}}\right)$$

Here, TF_{A_i} refers to the total occurrence frequency of an API class A_i in the collected code segments, N refers to total Q&A threads in the corpus, and DF_{A_i} is the number of threads that mentioned API class A_i in their texts.

API Class Weight Estimation with PageRank: Semantics of a term are often determined by its contexts, i.e., surrounding terms [50, 51]. Hence, the inter-dependence of terms is an important factor in the estimation of term weight. However, TF-IDF assumes term independence (i.e., ignores term contexts) in the weight estimation. Hence, it might fail to identify highly important but not so frequent terms from a body of texts [52, 53]. We thus employ another term weighting method that considers dependencies among terms in the weight estimation. In particular, we apply PageRank algorithm [53, 54] to the relevance feedback documents, i.e., relevant code segments, and identify the important API classes as follows.

Construction of API Co-occurrence Graph: Since the PageRank algorithm operates on a graph-based structure, we transform pseudo-relevance feedback documents into a graph of API classes (i.e., Fig. 9.3, Step 6, Algorithm 1, Line 13). In particular, we extract all API classes from each code segment using island parsing [49], and then develop an ordered list by preserving their initialisation order in the code. For example, the code snippet in Listing 9.1 is converted into a list of six API classes. Co-occurrences of items in a certain context has long been considered as an indication of

Fig. 9.4 API co-occurrence graph for the code segment in Listing 9.1



relatedness among the items [51, 53]. We thus capture the immediate co-occurrences of API classes in the above list, consider such co-occurrences as connecting edges, and then develop an API co-occurrence graph (e.g., Fig. 9.4). We repeat the same step for each of the code segments, and update the connectivities in the graph. We develop one graph for the code segments from questions and another graph for the code segments from answers which were returned as a part of the pseudo-relevance feedback.

API Class Rank Calculation: PageRank has been widely used for web link analysis [54] and term weighting in Information Retrieval domain [53]. It applies the underlying mechanism of recommendation or voting for determining the importance of an item (e.g., web page, term) [43]. That is, PageRank considers a node as important only if it is recommended (i.e., connected to) by other important nodes in the graph. The same idea has been widely used for separating legitimate pages from spam pages [55]. Similarly, in our problem context, if an API class co-occurs with other important API classes across multiple code segments that are relevant to a programming task, then this API class is also considered to be important for the task.

We apply the PageRank algorithm on each of the two graphs (i.e., Fig. 9.3, Step 6, Algorithm 1, Line 14), and determine the importance $ACR(v_i)$ (i.e., API Class Rank) of each node v_i by recursively applying the following equation:

$$ACR(v_i) = (1 - \phi) + \phi \sum_{j \in In(v_i)} \frac{ACR(v_j)}{|Out(v_j)|} \quad (0 \leq \phi \leq 1)$$

Here, $In(v_i)$ refers to nodes providing inbound links (i.e., votes) to node v_i whereas $Out(v_i)$ refers to nodes that v_i is connected to through outbound links, and ϕ is the damping factor. In the context of the world wide web, [54] considered ϕ as the probability of visiting a web page and $1 - \phi$ as the probability of jumping off the page by a random surfer. We use a value $\phi = 0.85$ for our work like the previous studies [52–54]. We initialise each node with a score of 0.25, and run an *iterative version* of PageRank on the graph. The algorithm pulls out weights from the surrounding nodes recursively, and updates the weight of a target node. This recursive process continues until the scores of the nodes converge below a certain threshold (e.g., 0.0001 [53]) or the total iteration count reaches the maximum (e.g., 100 [53]). Once the computation is over, each node (i.e., API class) is left with a score which is considered as a numerical proxy to its relative importance among all nodes.

Algorithm 1 Query reformulation using relevant API classes

```

1: procedure NLP2API( $Q$ ) ▷  $Q$ : natural language query
2:    $R \leftarrow \{\}$  ▷  $R$ : Relevant API classes
3:    $C \leftarrow \text{developQ\&ACorpus}(SO\ Dump)$  ▷  $C$ : SO corpus
4:    $Q_{pp} \leftarrow \text{preprocess}(Q)$ 
5:   ▷ collecting pseudo-relevance feedback
6:    $PRF \leftarrow \text{getPRF}(Q_{pp}, C)$ 
7:    $PRF_Q \leftarrow \text{getQuestionCodeSegments}(PRF)$ 
8:    $PRF_A \leftarrow \text{getAnswerCodeSegments}(PRF)$ 
9:   ▷ collecting candidate API list
10:  for PRF  $prf \in \{PRF_Q, PRF_A\}$  do
11:     $TW \leftarrow \text{calculateTFIDF}(prf, C)$ 
12:     $WC[prf] \leftarrow \text{getTopKWeightedClasses}(TW)$ 
13:     $G \leftarrow \text{developAPICo-occurrenceGraph}(prf)$ 
14:     $ACR \leftarrow \text{calculateAPIClassRank}(G)$ 
15:     $RC[prf] \leftarrow \text{getTopKRankedClasses}(ACR)$ 
16:  end for
17:  ▷ training the fastText model
18:   $M_{ft} \leftarrow \text{getFastTextModel}(\text{preprocess}(SO\ Dump))$ 
19:  ▷ API relevance estimation
20:   $A \leftarrow \text{getAllCandidateAPIClasses}(RC \cup WC)$ 
21:  for CandidateAPIClass  $A_i \in A$  do
22:    ▷ calculate Borda score
23:     $S_B[A_i] \leftarrow \text{getBordaScore}(A_i, RC, WC)$ 
24:    ▷ semantic relevance between API class and query
25:     $S_P[A_i] \leftarrow \text{getQuery-APIProximity}(A_i, Q_{pp}, M_{ft})$ 
26:     $R[A_i].score \leftarrow S_B[A_i] + S_P[A_i]$ 
27:  end for
28:  ▷ ranking of the API classes
29:   $\text{rankedClasses} \leftarrow \text{sortByFinalScore}(R)$ 
30:  ▷ reformulation of the initial query
31:  return  $Q_{pp} + \text{rankedClasses}$ 
32: end procedure

```

Selection of Candidate API Classes: Once two weights—TF-IDF and PageRank—of each of the potential candidates are calculated, we rank the candidates according to their weights. Then we select Top-N (e.g., $N = 16$, check RQ₁ for justification) API classes from each of the four lists (i.e., two lists for each term weight, Fig. 9.3, Step 7, Algorithm 1, Lines 9–16). In the Stack Overflow Q&A site, a question often describes a programming problem (or a task), whereas the answer offers a solution. Thus, API classes involved with the problem and API classes forming the solution should be treated differently for identifying the *relevant* and *specific* API classes for the task. We leverage these inherent differences of context and semantics between questions and answers, and treat their code segments separately unlike the earlier study by Nie et al. [20] that overlooks such differences.

9.4.2 Borda Score Calculation

Borda count is a widely used election method where the voters sort their political candidates on a scale of preference [56, 57]. In the context of Software Engineering, Holmes [58] first applies Borda count to recommend relevant code examples for the code under development in the IDE. They apply this method to six ranked lists of code examples collected using six structural heuristics, and then suggest the most frequent examples across these lists as the most relevant ones. Similarly, we apply this method to our four candidate API lists (i.e., Fig. 9.3, Step 8, Algorithm 1, Lines 22–23) where each of the API classes are ranked based on their importance estimates (e.g., TF-IDF, API Class Rank). We calculate Borda score S_B for each of the API classes ($\forall A_i \in A$) from these ranked candidate lists— $WRC = \{WC_Q, WC_A, RC_Q, RC_A\}$ —as follows:

$$S_B(A_i \in A) = \sum_{RL_j \in WRC} 1 - \frac{rank(A_i, RL_j)}{|RL_j|}$$

Here, A refers to the set of all API classes extracted from the ranked candidate lists— WRC , $|RL_j|$ denotes each list size, and $rank(A_i, RL_j)$ returns the rank of class A_i in the ranked list. Thus, an API class that occurs at the top positions in multiple candidate lists is likely to be more important for a target programming task than the ones that either occur at the lower positions or do not occur in multiple lists.

9.4.3 Query-API Semantic Proximity Analysis

Pseudo-relevance feedback, PageRank (Sect. 9.4.1), and Borda count (Sect. 9.4.2) analyse local contexts of the query keywords within a set of tentatively relevant documents (i.e., Q&A threads) and then extract candidate API classes for query reformulation. Although local context analysis is useful, existing studies report that such analysis alone might cause topic drift from the original query [32, 59]. We thus further analyse global contexts of the query keywords, and determine the semantic proximity between the given natural language query and the candidate API classes as follows.

Word2Vec Model Development using Machine Learning: Mikolov and colleagues [60] proposed a neural network-based tool—*word2vec*—for learning word embeddings from an ultra-large body of texts where they employ continuous bag of words (CBOW) and skip-gram models. While other studies attempt to define the context of a word using co-occurrence frequencies or TF-IDF [39, 51, 61], they offer a probabilistic representation of the context. In particular, they learn *word embeddings* for each of the words from the corpus, and map each word to a point in the semantic space so that semantically similar words appear in close proximity. We leverage this notion of *semantic proximity*, and determine the relevance of a candidate API class to the given query. It should be noted that such proximity measure could be an effective

tool to overcome the *vocabulary mismatch issues* [62]. We thus develop a *word2vec* model where 1.3 million programming questions and answers (i.e., 656,538 Q&A pairs, collected in Sect. 9.4.1) are employed as the corpus. We normalise each question and answer using standard natural language preprocessing, and learn the word embeddings (Fig. 9.3, Step 1b, 1c, 1d, Algorithm 1, Lines 17–18) using skip-gram model. For our learning, we use *fastText* [63], an improved version of *word2vec* that incorporates sub-word information into the model. We performed the learning offline and it took about one hour. It should be noted that our model is learned using default parameters (e.g., output vector size = 100, context window size = 5, and minimum word occurrences = 5) provided by the tool.

Semantic Relevance Estimation: While a given query contains multiple keywords, a candidate API class might not be semantically close to all of them. We thus capture the maximum proximity estimate between an API class and any of the query keywords as the relevance estimate of the class. In particular, we collect word embeddings (i.e., a vector of 100 real-valued estimates of the contexts) of each candidate API class $A_i \in A$ and each keyword $q \in Q$, and determine their semantic proximity S_P using *cosine similarity* (i.e., Fig. 9.3, Step 9, Algorithm 1, Lines 24–25) as follows:

$$S_P(A_i \in A) = \{f(A_i, q) \mid f(A_i, q) > f(A_i, q_0) \forall q_0 \in Q\}$$

$$f(A_i, q) = \text{Cos}(\text{fastText}(A_i), \text{fastText}(q))$$

Here, *fastText*(.) returns the learned word embeddings of either a query keyword or an API class, and $f(A_i, q)$ returns the *cosine similarity* between their word embeddings. We use `print-word-vectors` option of *fastText*, and collect the word embeddings from our learned model on Stack Overflow.

9.4.4 API Class Relevance Ranking and Query Reformulation

Once Borda score S_B and semantic proximity score S_P are calculated, we normalise both scores between 0 and 1, and then sum them up using a *linear combination* (i.e., Line 26, Algorithm 1) for each of the candidate API classes. While fine-tuned relative weight estimation for these two scores could have been a better approach, we keep that as a part of future work. Besides, equal weights also reported pretty good results (e.g., 82% Top-10 accuracy) according to our investigation. The API classes are then ranked according to their final scores, and Top-K (e.g., $K = 10$) classes are suggested as the relevant classes for the programming task stated as a generic query (i.e., Fig. 9.3, Steps 10–12, Algorithm 1, Lines 19–29).

Table 9.6 Reformulations of an NL query for improved code search

Technique	Reformulated query	QE
Baseline	Convert image to grayscale without losing transparency	115
QECK [20]	{Convert image grayscale losing transparency} + {hsb pixelsByte png iArray img correctly HSB mountainMap enhancedImagePixels file}	11
Google	Convert image to grayscale without losing transparency	02
Proposed	{Convert image grayscale losing transparency} + { BufferedImage Grayscale ImageEdit ColorConvertOp File Transparency ColorSpace BufferedImageOp Graphics ImageEffects}	02

QE = Rank of the first correct result returned by the query

These API classes are then appended to the given query as reformulations [42] (i.e., Fig. 9.3, Steps 13, Algorithm 1, Lines 30–31). Table 9.6 shows our reformulated query for the showcase natural language query using the suggested API classes.

9.4.5 Evaluation of NLP2API

In this section, we assess the effectiveness of NLP2API in reformulating queries for reusable code search. We answer the following research questions:

- **RQ₁**: How does NLP2API perform in recommending relevant API classes for a given query?
- **RQ₂**: Can NLP2API outperform the state-of-the-art technique on relevant API class suggestion for a query?
- **RQ₃**: Can NLP2API significantly improve the results provided by state-of-the-art code or web search engines?

All the information and material necessary for replicating the results of this evaluation is available online at <https://github.com/masud-technope/NLP2API-Replication-Package>.

9.4.5.1 Experimental Dataset

Dataset Collection: We collect 310 code search queries from four popular programming tutorial sites—KodeJava [64], Java2s [65], CodeJava [66], and JavaDB [67]—for our experiments. While 150 of these queries were taken from a publicly available dataset [39], we attempted to extend the dataset by adding 200 more queries. However, after removing the duplicates and near duplicates, we ended up with 160

queries. Thus, our dataset contains a total of 310 (i.e., 150 old + 160 new) search queries. Each of these sites above discusses hundreds of programming tasks as Q&A threads where each thread generally contains (1) a question title, (2) a solution (i.e., code), and (3) prose explaining the code succinctly. The question title (e.g., “*How do I decompress a GZip file in Java?*” [68]) generally comprises a few important keywords and often *resembles* a real-life search query. We thus use these titles from tutorial sites as code search queries in our experiments, as were also used by the earlier studies [39, 69].

Ground Truth Preparation: The prose that explains code in the tutorial sites above often includes one or more API classes from the code (e.g., `GZipInputStream`, `FileOutputStream`). Since these API classes are chosen to explain the code that implements a programming task, they are generally relevant and specific to the task. We thus consider these *relevant* and *specific* API classes as the *ground truth* for the corresponding question title (i.e., our search query) [39]. We develop a *ground truth API set* to evaluate the performance of our technique in the API class suggestion. We also collect the code segments from each of the 310 Q&A threads from the tutorial sites above as the *ground truth code segments*, and use them to evaluate the query reformulation performance (i.e., in terms of code retrieval) of our technique. Given that these API classes and code segments are publicly available online and were consulted by thousands of technical users over the years, subjectivity associated with their relevance to the corresponding tasks (i.e., our selected queries) is minimised [69]. Our dataset preparation step took ≈ 25 man hours.

9.4.5.2 Performance Metrics

We choose five performance metrics that were widely adopted by relevant literature [19, 20, 39, 43, 69, 70], for the evaluation and validation of our technique as follows:

Top-K Accuracy/Hit@K: It is the percentage of search queries for each of which at least one item (e.g., API class) from the *ground truth* is returned within the Top-K results.

Mean Reciprocal Rank@K (MRR@K): Reciprocal Rank@K is defined as the multiplicative inverse of the rank of the first relevant item (e.g., API class from ground truth) in the Top-K results returned by a technique. Mean Reciprocal Rank@K (MRR@K) averages such measures for all queries.

Mean Average Precision@K (MAP@K): Precision@K is the precision calculated at the occurrence of Kth item in the ranked list. Average Precision@K (AP@K) averages the precision@K for all relevant items (e.g., API class from ground truth) within the Top-K results for a search query. Mean Average Precision@K is the mean of Average Precision@K for all queries from the dataset.

Mean Recall@K (MR@K): Recall@K is defined as the percentage of ground truth items (e.g., API classes) that are correctly recommended for a query in the Top-K results by a technique. Mean Recall@K (MR@K) averages such measures for all queries from the dataset.

Query Effectiveness (QE): It is defined as the rank of the first correct item (i.e., ground truth code segment) in the result list returned by a query. The measure is an approximation of the developer’s effort in locating the first code segment relevant to a given query. Thus, the lower the effectiveness measure is, the more effective the query is [41, 43]. We use this measure to evaluate the improvement of a query through reformulations offered by a technique.

9.4.5.3 Result

RQ₁: How does NLP2API perform in recommending relevant API classes for a given query?

From Table 9.7, we see that our technique returns relevant API classes for 73% of the queries with 51% mean average precision and 40% recall when only Top-5 results are considered. That is, half of the suggested classes come from the *ground truth*, and our approach succeeds for seven out of 10 queries. More importantly, it achieves a mean reciprocal rank of 0.54. That means, on average, the first relevant API class can be found at the second position of the result list. Such classes can also be found at the first position for 42% of the queries. All these statistics are highly promising according to relevant literature [19, 39]. Figure 9.5 further demonstrates our performance measures for Top-1 to Top-10 results. We see that accuracy, recall, and reciprocal rank measures increase monotonically which is expected. Interestingly, the precision measure shows an almost steady behaviour. That means, as more results were collected, our technique was able to *filter out* the *false positives* which demonstrates its high potential for API suggestion.

Table 9.7 Performance of NLP2API in Relevant API suggestion

Performance Metric	Top-1	Top-3	Top-5	Top-10
Top-K Accuracy	41.94%	64.19%	72.90%	81.61%
Mean Reciprocal Rank@K	0.42	0.52	0.54	0.55
Mean Average Precision@K	41.94%	50.62%	50.56%	47.85%
Mean Recall@K	12.53%	30.17%	40.28%	57.87%

Top-K = Performance measures for Top-K suggestions

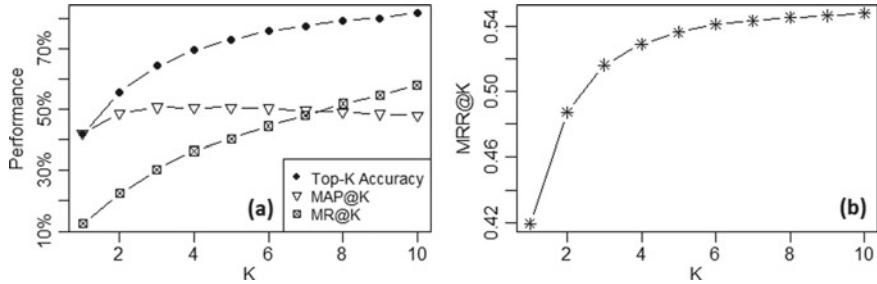


Fig. 9.5 Performance of NLP2API in API class suggestion for various Top-K results

9.4.5.4 Comparison of NLP2API with Existing Techniques

RQ₂: Can NLP2API outperform the state-of-the-art technique on relevant API class suggestion for a query?

We compare our technique with the state-of-the-art approach—RACK [39]—on API class suggestion for a natural language query. RACK [39] employs two heuristics—Keyword-API Co-occurrence (KAC) and Keyword-Keyword Coherence (KKC)—for suggesting relevant API classes from Q&A threads of Stack Overflow for a given query. It is reported to outperform earlier approaches [19, 69] and is therefore considered to be the state of the art in relevant API class suggestion. We collected the authors’ implementation of RACK from the corresponding web portal, ran the tool as is on our dataset, and then extracted the evaluation results.

From Table 9.8, we see that our technique—NLP2API—outperforms RACK, especially in precision, recall, and reciprocal rank. It should be noted that our reported performance measures for RACK are pretty close to the authors’ reported measures [39], which indicates a *fair comparison*. We see that RACK recommends API classes

Table 9.8 Comparison with the state of the art in API class suggestion

Technique	Metric	Top-1	Top-3	Top-5	Top-10
RACK [39]	Top-K Accuracy	20.97%	52.90%	64.19%	77.10%
	MRR@K	0.21	0.35	0.37	0.39
	MAP@K	20.97%	34.76%	36.76%	36.38%
	MR@K	6.25%	20.81%	28.06%	39.22%
NLP2API(Proposed)	Top-K Accuracy	41.94%	64.19%	72.90%	81.61%
	MRR@K	0.42	0.52	0.54	0.55
	MAP@K	41.94%	50.62%	50.56%	47.85%
	MR@K	12.53%	30.17%	40.28%	57.87%

Top-K = Performance measures for Top-K suggestions

correctly for 64% of the queries with 37% precision, 28% recall, and a reciprocal rank of 0.37 when Top-5 results are considered. On the contrary, our technique recommends correctly for 73% of the queries with 51% precision, 40% recall, and a promising reciprocal rank of 0.54 in the same context. These are 14%, 38%, 44%, and 46% improvements respectively over the state-of-the-art performance measures. Statistical tests for various Top-K results (i.e., $1 \leq K \leq 10$) also reported significance (i.e., all $p\text{-values} \leq 0.05$) of our technique over the state of the art with large effect sizes (i.e., $0.39 \leq \Delta \leq 0.90$).

Our technique outperforms the state-of-the-art approach on relevant API class suggestions, and it suggests relevant API classes with **38%** higher precision and **46%** higher reciprocal rank than those of the state of the art.

RQ₃: Can NLP2API significantly improve the results provided by state-of-the-art code or web search engines?

Although our approach outperforms the state-of-the-art studies [20, 39] on relevant API suggestion and query reformulation, we further compare with two popular web search engines—*Google*, *Stack Overflow native search*—and one popular code search engine—*GitHub code search*. Given the enormous and *dynamic index database* and *restrictions* on the *query length or type*, a full-scale or direct comparison with these search engines is neither feasible nor fair. We thus investigate whether results returned by these contemporary search engines for generic queries could be significantly improved or not with the help of our reformulated queries.

Collection of Search Results and Establishment of Ground Truth: We first collect Top-30 results returned by each search engine for each of the 310 queries. For result collection, we make use of *Google’s custom search API* [71] and the native API endpoints provided by Stack Overflow and GitHub. Since our goal is to find relevant code snippets, we adopt a pragmatic approach in the establishment of ground truth for this experiment. In particular, we analyse those 30 results semi-automatically, look for *ground truth code segments* in their contents, and then select Top-10 results as *ground truth search results* that contain either the ground truth code or highly similar code. It should be noted that ground truth code segments and our suggested API classes are taken from two different sources.

Comparison between Initial Search Results and Re-ranked Results with Reformulated Queries: While the search engines return results mostly for the natural language queries, we further re-rank the results with our reformulated queries (i.e., generic search keywords + relevant API classes) using lexical similarity analysis (e.g., cosine similarity [73]). We then evaluate Top-10 results both by each search engine and by our re-ranking approach against the *ground truth search results*, and demonstrate the potential of our reformulations.

From Table 9.9, we see that the re-ranking approach that leverages our reformulated queries improves the initial search results returned by each of the engines. In particular, the performances are improved in terms of precision and discounted cumulative gain. For example, Google returns search results with 66% precision and

Table 9.9 Comparison with popular web/code search engines

Technique	Hit@10 (%)	MAP@10 (%)	MRR@10	NDCG@10
Google	100.00	65.50	0.80	0.47
NLP2API_{Google}	100.00	76.73	0.83	0.61
Stack Overflow	90.65	59.46	0.67	0.40
NLP2API_{SO}	91.29	79.95	0.87	0.67
GitHub	88.06	53.06	0.55	0.41
NLP2API_{GitHub}	89.03	70.69	0.78	0.59

NDCG = Normalised Discounted Cumulative Gain [72]

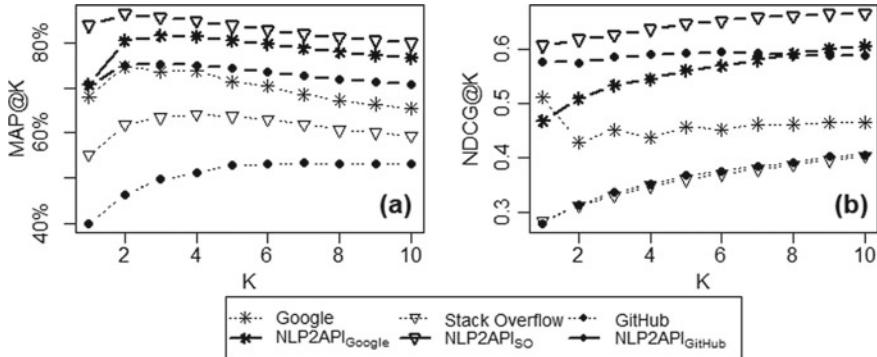


Fig. 9.6 Comparison between popular web/code search engines and NLP2API in relevant code segment retrieval using **a** MAP@K and **b** NDCG@K

0.47 NDCG when Top-10 results are considered. Our approach, $\text{NLP2API}_{\text{Google}}$, improves the ranking and achieves a MAP@10 of 77% and a NDCG@10 of 0.61 which are 17% and 30% higher, respectively. That is, although Google performs high as a *general-purpose* web search engine, it might always not be precise for *code search* due to the lack of appropriate contexts. Our approach incorporates context into the search using relevant API names, and delivers more precise code search results. As shown in Table 9.9 and Fig. 9.6, similar findings were also achieved against GitHub code search and Stack Overflow native search.

Our technique improves upon the result ranking of all three popular search engines using its reformulated queries. It achieves **17%** higher precision and **30%** higher NDCG than Google, i.e., the best performing search engine.

9.5 Conclusion

In this chapter, we reported two case studies in which AI is leveraged to improve a software maintenance task. In the first case study, we described CAME, a deep learning-based approach that relies on both structural and historical information to detect anti-patterns. CAME exploits historical values of structural code metrics and uses a CNN classifier to infer the presence of anti-patterns from this information. An evaluation of CAME for the detection of the God class anti-pattern in three software systems shows that it significantly outperforms existing detection tools from the literature. Moreover, the performances of CAME increase with the length of the metrics history fed through our model, and CAME significantly outperforms other ML-based classifiers that do not rely on historical data.

In the second case study, we described NLP2API; an ML-based approach to reformulate queries for reusable code search on the Internet. NLP2API automatically identifies relevant specific API classes from the Stack Overflow Q&A site for a programming task written as a natural language query, and then reformulates the query to improve code search. Evaluation results show that NLP2API outperforms the state-of-the-art approach on relevant API class suggestions. NLP2API also achieves 17% higher precision and 30% higher NDCG than Google, which is the best performing search engine.

References

1. M. Shepperd, D. Bowes, T. Hall, Researcher bias: the use of machine learning in software defect prediction. *IEEE Trans. Softw. Eng.* **40**(6), 603–616 (2014)
2. S.E.S. Taba, F. Khomh, Y. Zou, A.E. Hassan, M. Nagappan, Predicting bugs using antipatterns. *IEEE International Conference on Software Maintenance* **2013**, 270–279 (2013)
3. L. An, F. Khomh, Y.-G. Guéhéneuc, An empirical study of crash-inducing commits in mozilla firefox. *Softw. Qual. J.* **26**, 553–584 (2017)
4. M. Bagherzadeh, N. Kahani, L. Briand, Reinforcement learning for test case prioritization. *IEEE Trans. Softw. Eng.* **01**, 1 (2021)
5. T. Mariani, S.R. Vergilio, A systematic review on search-based refactoring. *Inf. Softw. Technol.* **83**, 14–34 (2017). <https://www.sciencedirect.com/science/article/pii/S0950584916303779>
6. R. Morales, R. Saborido, F. Khomh, F. Chicano, G. Antoniol, Earmo: an energy-aware refactoring approach for mobile apps. *IEEE Trans. Software Eng.* **44**(12), 1176–1206 (2018)
7. A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: an empirical study, in *Proceedings of the 2013 International Conference on Software Engineering* (IEEE Press, 2013), pp. 682–691
8. M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, vol. 2011 (IEEE, 2011), pp. 181–190
9. N. Moha, Y. Guéhéneuc, D. Laurence, L.M. Anne-Francoise, Decor: a method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng. (TSE)* **36**(1), 20–36 (2010)
10. N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.* **35**(3), 347–367 (2009)

11. R. Marinescu, G. Ganea, I. Verebi, Incode: continuous quality assessment and improvement, in *2010 14th European Conference on Software Maintenance and Reengineering (CSMR)* (IEEE, 2010), pp. 274–275
12. F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, A.D. Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in *ASE* (2013), pp. 268–278
13. M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Jdeodorant: identification and application of extract class refactorings,” in *2011 33rd International Conference on Software Engineering (ICSE)* (IEEE, 2011), pp. 1037–1039
14. F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, Bdtx: a gqm-based bayesian approach for the detection of antipatterns. *J. Syst. Softw.* **84**(4), 559–572 (2011), The Ninth International Conference on Quality Software. <https://www.sciencedirect.com/science/article/pii/S0164121210003225>
15. A. Barbez, F. Khomh, Y.-G. Guéhéneuc, Deep learning anti-patterns from code metrics history, in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2019), pp. 114–124
16. M.M. Rahman, C.K. Roy, Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics, in *Proceedings of the ICSME* (2018), p. 12
17. J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, S. Klemmer, Two studies of opportunistic programming: interleaving web foraging, learning, and writing code, in *Proceedings of the SIGCHI* (2009), pp. 1589–1598
18. RACK Website. <http://homepage.usask.ca/~masud.rahman/rack>
19. F. Thung, S. Wang, D. Lo, J. Lawall, Automatic recommendation of API methods from feature requests, in *Proceedings of the ASE* (2013), pp. 290–300
20. L. Nie, H. Jiang, Z. Ren, Z. Sun, X. Li, Query expansion based on crowd knowledge for code search. *TSC* **9**(5), 771–783 (2016)
21. R. Sirres, T.F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, Y.L. Traon, Augmenting and structuring user queries to support efficient free-form code search. *EMSE* (2018)
22. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empir. Softw. Eng.* **23**(3), 1188–1221 (2018)
23. H. He, E.A. Garcia, Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.* **9**, 1263–1284 (2008)
24. M. Jansche, Maximum expected f-measure training of logistic regression models, in *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing* (Association for Computational Linguistics, 2005), pp. 692–699
25. I.H. Witten, E. Frank, M.A. Hall, C.J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques* (Morgan Kaufmann, 2016)
26. M. Lanza, R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (Springer Science & Business Media, 2007)
27. B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity* (Prentice-Hall, Inc., 1995)
28. T.J. McCabe, A complexity measure. *IEEE Trans. Softw. Eng.* **4**, 308–320 (1976)
29. J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* **13**, 281–305 (2012)
30. T.G. Dietterich, Ensemble methods in machine learning, in *International Workshop on Multiple Classifier Systems* (Springer, 2000), pp. 1–15
31. B.A. Campbell, C. Treude, Nlp2code: Code snippet content assist via natural language tasks, in *Proceedings of the ICSME* (2017), pp. 628–632
32. Z. Li, T. Wang, Y. Zhang, Y. Zhan, G. Yin, Query reformulation by leveraging crowd wisdom for scenario-based software search, in *Proceedings of the Internetware* (2016), pp. 36–44
33. S.K. Bajracharya, C.V. Lopes, Analyzing and mining a code search engine usage log. *EMSE* **17**(4–5), 424–466 (2012)
34. Stack Exchange Archive. <https://archive.org/download/stackexchange>

35. L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, D. Fullerton, Improving low quality stack overflow post detection, in *Proceedings of the ICSME* (2014), pp. 541–544
36. S. Ercan, Q. Stokkink, A. Bacchelli, Automatic assessments of code explanations: predicting answering times on stack overflow, in *Proceedings of the MSR* (2015), pp. 442–445
37. M.M. Rahman, S. Yeasmin, C.K. Roy, Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions, in *Proceedings of the CSMR-WCRE* (2014), pp. 194–203
38. M. Duijn, A. Kucera, A. Bacchelli, Quality questions need quality code: classifying code fragments on stack overflow, in *Proceedings of the MSR* (2015), pp. 410–413
39. M.M. Rahman, C.K. Roy, D. Lo, RACK: automatic API recommendation using crowdsourced knowledge, in *Proceedings of the SANER* (2016), pp. 349–359
40. E. Hill, S. Rao, A. Kak, On the use of stemming for concern location and bug localization in Java, in *Proceedings of the SCAM* (2012), pp. 184–193
41. L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, A. Marcus, Query-based configuration of text retrieval solutions for software engineering tasks, in *Proceedings of the ESEC/FSE* (2015), pp. 567–578
42. S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, T. Menzies, Automatic query reformulations for text retrieval in software engineering, in *Proceedings of the ICSE* (2013), pp. 842–851
43. M. Rahman, C.K. Roy, STRICT: information retrieval based search term identification for concept location, in *Proceedings of the SANER* (2017), pp. 79–90
44. Z. Lin, Y. Zou, J. Zhao, B. Xie, Improving software text retrieval using conceptual knowledge in source code, in *Proceedings of the ASE*, 2017, pp. 123–134
45. Jsoup: Java HTML Parser. <http://jsoup.org>
46. Example code snippet. <https://goo.gl/BBxPwH>
47. G. Gay, S. Haiduc, A. Marcus, T. Menzies, On the use of relevance feedback in IR-based concept location, in *Proceedings of the ICSM*, 2009, pp. 351–360
48. J. Rocchio, *The SMART Retrieval System—Experiments in Automatic Document Processing* (Prentice-Hall, Inc)
49. P.C. Rigby, M.P. Robillard, Discovering essential code elements in informal documentation, in *Proceedings of the ICSE* (2013), pp. 832–841
50. X. Ye, H. Shen, X. Ma, R. Bunescu, C. Liu, From word embeddings to document similarities for improved information retrieval in software engineering, in *Proceedings of the ICSE* (2016), pp. 404–415
51. T. Yuan, D. Lo, J. Lawall, Automated construction of a software-specific word similarity database, in *Proceedings of the CSMR-WCRE* (2014), pp. 44–53
52. M.M. Rahman, C.K. Roy, Improved query reformulation for concept location using coderank and document structures, in *Proceedings of the ASE* (2017), pp. 428–439
53. R. Mihalcea, P. Tarau, Textrank: bringing order into texts, in *Proceedings of the EMNLP* (2004), pp. 404–411
54. S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* **30**(1–7), 107–117 (1998)
55. F.J. Ortega, C. Macdonald, J.A. Troyano, F. Cruz, Spam detection with a content-based random-walk algorithm, in *Proceedings of the SMUC* (2010), pp. 45–52
56. Y. Zhang, W. Zhang, J. Pei, X. Lin, Q. Lin, A. Li, Consensus-based ranking of multivalued objects: a generalized borda count approach. *TKDE* **26**(1), 83–96 (2014)
57. Borda count. <https://goo.gl/oFDUTN>
58. R. Holmes, G. Murphy, Using structural context to recommend source code examples, in *Proceedings of the ICSE* (2005), pp. 117–125
59. C. Carpineto, G. Romano, A survey of automatic query expansion in information retrieval. *ACM Comput. Surv.* **44**(1), 1:1–1:50 (2012)
60. T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space (2013). CoRR, <arXiv:abs/1301.3781>

61. A. Marcus, A. Sergeyev, V. Rajlich, J.I. Maletic, An information retrieval approach to concept location in source code, in *Proceedings of the WCRE* (2004), pp. 214–223
62. G.W. Furnas, T.K. Landauer, L.M. Gomez, S.T. Dumais, The vocabulary problem in human-system communication. *Commun. ACM* **30**(11), 964–971 (1987)
63. P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information (2016). arXiv preprint [arXiv:1607.04606](https://arxiv.org/abs/1607.04606)
64. KodeJava: Java Examples. <http://kodejava.org>
65. Java2s: Java Tutorials. <http://java2s.com>
66. CodeJava. <http://www.codejava.net>
67. JavaDB: Java Code Examples. <http://www.javadb.com>
68. How Do I Decompress a gzip File in Java? <https://goo.gl/14QkXq>
69. W. Chan, H. Cheng, D. Lo, Searching connected API subgraph via text phrases, in *Proceedings of the FSE* (2012), pp. 10:1–10:11
70. C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, C. Fu, Portfolio: finding relevant functions and their usage, in *Proceedings of the ICSE* (2011), pp. 111–120
71. Google custom search. <https://developers.google.com/custom-search>
72. Y. Wang, L. Wang, Y. Li, D. He, T. Liu, A theoretical analysis of NDCG type ranking measures, in *Proceedings of the COLT* (2013), pp. 25–54
73. M.M. Rahman, C.K. Roy, On the use of context in recommending exception handling code examples, in *Proceedings of the SCAM* (2014), pp. 285–294

Part IV

AI Techniques from Scratch

Chapter 10

Metaheuristics in a Nutshell



Javier Ferrer and Pedro Delgado-Pérez

Abstract Previous chapters have covered diverse software engineering areas, describing how a considerable number of activities throughout the software life cycle can benefit from optimisation and machine learning approaches to automate costly and laborious tasks. Within the content of those chapters, several search-based techniques are roughly explained or just mentioned, often assuming that the reader is familiar with these techniques and the principles on which they are founded. In order to fill this gap, this chapter aims to give a definition of the main terms surrounding the concept of *metaheuristic*, to provide a quick overview of the most popular search algorithms, and, ultimately, to serve as a handy reference to help the reader understand the background information of this book. This includes a taxonomy to classify existing techniques depending on the nature of the search strategy (trajectory-based and population-based techniques), differentiation of single and multi-objective optimisation problems—and the specific algorithms that can be employed in each case—a general explanation of the basic mechanics of some of the most frequently used techniques on the search-based software engineering community, and an enumeration of quality indicators and statistical tests to evaluate the results.

10.1 Introduction

Solving techniques for optimisation can be classified into *Exact* and *Approximate*. Exact techniques, which find the optimal solution mathematically or through an exhaustive search, guarantee the optimality of the obtained solution. However, these techniques present some drawbacks. The time they require, though bounded, may be too large, especially for NP-hard problems. Furthermore, it is not always possible

J. Ferrer (✉)

ITIS Software, University of Malaga, Bulevar Louis Pasteur 35, 29010 Malaga, Spain

e-mail: ferrer@lcc.uma.es

P. Delgado-Pérez

Department of Computer Science and Engineering, University of Cádiz, Cádiz, Spain

e-mail: pedro.delgado@uca.es

to find such an exact technique for every problem. As a result, exact techniques are not always a good choice, since both their time and memory requirements can become unreasonably high for large-scale problems. For this reason, approximate techniques have been widely used by the international research community in the last few decades. These methods sacrifice the guarantee of finding an optimum in favour of providing some satisfactory solution within a reasonable consumption of resources [20].

The word *heuristic*, from the ancient greek “to find, to discover”, is used with these two closely-related meanings since a heuristic is a strategy that leads to a discovery. Currently, adaptations of the term have been made in different areas. In Computer Science, it refers to a simple and intuitive technique that employs a practical methodology to produce close to optimal solutions for a given complex problem using specific information about the problem. That solution is not guaranteed to be optimal or perfect, but sufficient for the immediate goals [29]. When finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of reaching a satisfactory solution. Heuristics can be seen as mental shortcuts that ease the cognitive load of making a decision. Heuristics are customised to the problem at hand and they try to take full advantage of the particularities of the problem. However, they can get trapped in a local optimum, failing to obtain a global optimum or optimal solution. In addition, they have the drawback of being problem-specific techniques, so a good heuristic for some given problem will offer little assistance when solving a different problem. Consequently, a more general-purpose technique was proposed: Metaheuristics [20].

The word *metaheuristic* is the combination of “meta” (which means “beyond”, here with the sense of a higher level of abstraction) and heuristic. Therefore, a metaheuristic is a high-level heuristic problem-independent technique. They are generally conceived as high-level heuristics able to use heuristic methods by guiding them over the search space in order to exploit their best capabilities to achieve better solutions, especially with incomplete or imperfect information or limited computation capacity. The main advantage of metaheuristics with respect to a simple heuristic is that they count on effective mechanisms to avoid getting trapped in local optima.

Metaheuristics have been successfully applied to a broad range of activities that have to be accomplished at different phases during the software development process, from project management to software maintenance. As a result, they have enabled the possibility to reach reasonably good solutions to heretofore impractical problems faced by software professionals. In fact, the application of search methods to deal with these types of problems has given rise to a body of knowledge on this domain, the so-called *Search-Based Software Engineering* (SBSE) [18], and even to some specific subareas within SBSE, like *Search-Based Software Testing* (SBST) [28], which focuses on different testing tasks where exhaustive testing is generally not viable. While this chapter offers a general perspective of popular metaheuristics, the rest of the chapters show how they can be tailored to form useful and robust techniques for different software engineering activities.

This chapter serves as a presentation of metaheuristics and the main metaheuristic techniques used to solve the different software engineering problems tackled in

this book. The rest of this chapter is organised as follows. Section 10.2 formally defines what is an optimisation problem, a multi-objective optimisation problem, and a metaheuristic. In Sect. 10.3, we classify the main metaheuristics. We describe trajectory-based metaheuristics in Sect. 10.4 and population-based ones in Sect. 10.5, including multi-objective techniques. Section 10.6 explores the different aspects that should be considered when evaluating the designed metaheuristics, including quality indicators and statistical analysis. Finally, Sect. 10.7 presents conclusions.

10.2 Background: Formal Definitions

This section is devoted to those readers who want to know some formal insights on the concepts of optimisation problems and metaheuristics. This section establishes some formal context by defining formally single-objective optimisation problems, multi-objective optimisation problems, and metaheuristics.

10.2.1 Optimisation Problem Definition

Definition 1 (*Optimisation problem*) An *optimisation problem* is defined as a pair (S, f) , where $S \neq \emptyset$ is called the solution space (or search space), and f is a function named *objective function* or *fitness function*, defined as $f : S \rightarrow \mathbb{R}$.

Solving an optimisation problem consists in finding a solution $x^* \in S$ such that:

$$f(x^*) \leq f(x), \quad \forall x \in S . \quad (10.1)$$

Note that Eq. 10.1 assumes minimisation without loss of generality, as explained later on. The objective function measures the quality of each possible solution in the search space. Therefore, a metaheuristic tries to find the best solution based on how valuable they are considering this function. In addition, the objective function allows particularising the metaheuristic algorithm to the problem that is being addressed. As such, it has to be devised for each specific problem to capture the essence of what makes a solution close or far from optimal in that particular domain.

Depending on its nature, the optimisation can be formulated as a maximisation problem (e.g., increase the benefits) or minimisation problem (e.g., reduce the time spent on a task). This formulation (Eq. 10.1) does not restrict the generality of the results, since an equivalence can be made between the two cases in the following manner

[1, 17]:

$$\max\{f(x)|x \in S\} \equiv \min\{-f(x)|x \in S\} . \quad (10.2)$$

Depending on the domain S , we have *binary* ($S \subseteq \mathbb{B}$), *integer* ($S \subseteq \mathbb{Z}$), *continuous* ($S \subseteq \mathbb{R}$), or *heterogeneous* optimisation problems ($S \subseteq (\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R})$). These are the main domains used in optimisation but there are more, such as permutations or trees.

10.2.2 Multi-objective Optimisation Problems

Single-objective techniques focus on trying to minimise (or maximise) the values obtained with one single fitness function f . But, most of the real-world optimisation problems require the optimisation of more than one objective function, usually in conflict with each other, i.e., if one objective is improved, some of the others will be worsened. For example, in the testing domain, two factors in conflict could be the generation of a test suite as short as possible while maximising the coverage—usually, increasing the coverage implies adding more test cases. In the absence of any further information, all the objectives of a Multi-Objective Optimisation Problem (MOP) are considered equally important.

Informally, a MOP can be defined as the problem of finding a vector of decision variables that satisfies a set of constraints and optimises a number of objective functions. Those functions define a set of performance criteria which are in conflict with each other. Thus, in this context, the term *optimisation* refers to the search of such a vector, which has acceptable values for all the objective functions. The formulation of a MOP extends the classic definition of single-objective optimisation by considering the existence of two or more objective functions. This implies, in general, the existence of a set of solutions where each one provides a different trade-off among the objectives. To choose the optimal solution for a problem, one popular option is to make use of the Pareto Optimality theory. In the following, we formally define the most important concepts related to MOPs, assuming, without loss of generality, that all objectives are minimised.

Definition 2 (Pareto Dominance) Given a vector function $f : S \rightarrow \mathbb{R}^k$, a solution $x^1 \in S$ is said to *dominate* a solution $x^2 \in S$, denoted with $x^1 \prec x^2$, if and only if $f_i(x^1) \leq f_i(x^2)$ for $i = 1, 2, \dots, k$, and there exists at least one j ($1 \leq j \leq k$) such that $f_j(x^1) < f_j(x^2)$.

Definition 3 (Pareto Optimal Set) We say that a solution x is *non-dominated* with respect to the solution space S , if there is no other solution $x^* \in S$ such that $f(x^*) \prec f(x)$. The set of non-dominated solutions X^* with respect to the solution space S is called *Pareto Optimal Set*.

Approximating the *Pareto Optimal Set* of a problem is the main goal of multi-objective optimisation techniques.

Definition 4 (Pareto Optimal Front) The *Pareto Optimal Front PF* is the image by f of the *Pareto Optimal Set* X^* (in the objective space), that is, $PF = f(X^*)$.

Fig. 10.1 Examples of dominated and non-dominated solutions

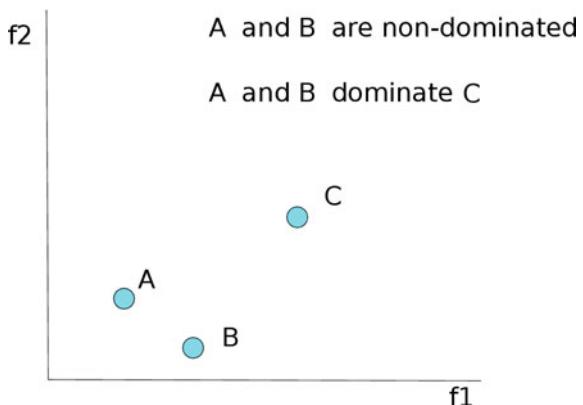


Figure 10.1 depicts some examples of dominated and non-dominated solutions. In this figure, A dominates C because $f_1(A) < f_1(C)$, and $f_2(A) < f_2(C)$. Similarly, B dominates C . A and B are non-dominated solutions because A is better than B in the first objective function ($f_1(A) < f_1(B)$), but B is better than A in the second objective function ($f_2(A) > f_2(B)$).

Taking into account this definition of Pareto dominance, a MOP is defined as follows:

Definition 5 (*Multi-objective Optimisation Problem*)

A MOP is defined as a 2-tuple (S, f) , where $S \neq \emptyset$ is called the solution space (or search space), and f is a vector function. A MOP consists in finding the Pareto Optimal Set X^* with respect to the solution space S considering the vector function f .

10.2.3 Metaheuristics Definition

A formal definition of metaheuristics can be found in [27], with an extension in [5]. A basic formulation of a metaheuristic is presented in the following definition:

Definition 6 (*Metaheuristic*) A metaheuristic \mathcal{M} is a tuple consisting of eight components as follows:

$$\mathcal{M} = \langle \mathcal{T}, \Xi, \mu, \lambda, \Phi, \sigma, \mathcal{U}, \tau \rangle , \quad (10.3)$$

where:

- \mathcal{T} is the set of elements operated by the metaheuristic. This set contains the search space, and in many cases, they both coincide.
- $\Xi = \{(\xi_1, D_1), (\xi_2, D_2), \dots, (\xi_v, D_v)\}$ is a collection of v pairs. Each pair is formed by a state variable of the metaheuristic and the domain of said variable.

- μ is the number of solutions operated by \mathcal{M} in a single step.
- λ is the number of new solutions generated in every iteration of \mathcal{M} .
- $\Phi : \mathcal{T}^\mu \times \prod_{i=1}^v D_i \times \mathcal{T}^\lambda \rightarrow [0, 1]$ represents the operator that produces new solutions from the existing ones. The function must verify for all $x \in \mathcal{T}^\mu$ and for all $t \in \prod_{i=1}^v D_i$,

$$\sum_{y \in \mathcal{T}^\lambda} \Phi(x, t, y) = 1 . \quad (10.4)$$

- $\sigma : \mathcal{T}^\mu \times \mathcal{T}^\lambda \times \prod_{i=1}^v D_i \times \mathcal{T}^\mu \rightarrow [0, 1]$ is a function that selects the solutions that will be manipulated in the next iteration of \mathcal{M} . This function must verify for all $x \in \mathcal{T}^\mu$, $z \in \mathcal{T}^\lambda$ and $t \in \prod_{i=1}^v D_i$,

$$\sum_{y \in \mathcal{T}^\mu} \sigma(x, z, t, y) = 1 , \quad (10.5)$$

$$\begin{aligned} \forall y \in \mathcal{T}^\mu, \sigma(x, z, t, y) = 0 \vee \sigma(x, z, t, y) > 0 \wedge \\ (\forall i \in \{1, \dots, \mu\}, (\exists j \in \{1, \dots, \mu\}, y_i = x_j) \vee (\exists j \in \{1, \dots, \lambda\}, y_i = z_j)) . \end{aligned} \quad (10.6)$$

- $\mathcal{U} : \mathcal{T}^\mu \times \mathcal{T}^\lambda \times \prod_{i=1}^v D_i \times \prod_{i=1}^v D_i \rightarrow [0, 1]$ represents the updating process for the state variables of the metaheuristic. This function must verify for all $x \in \mathcal{T}^\mu$, $z \in \mathcal{T}^\lambda$ and $t \in \prod_{i=1}^v D_i$,

$$\sum_{u \in \prod_{i=1}^v D_i} \mathcal{U}(x, z, t, u) = 1 . \quad (10.7)$$

- $\tau : \mathcal{T}^\mu \times \prod_{i=1}^v D_i \rightarrow \{\text{false}, \text{true}\}$ is a function that decides the termination of the algorithm.

The previous definition represents the typical stochastic behaviour of most metaheuristics. In fact, the functions Φ , σ and \mathcal{U} should be considered as conditional probabilities. For instance, the value of $\Phi(x, t, y)$ is the probability to generate the offspring vector $y \in \mathcal{T}^\lambda$, since the current set of individuals in the metaheuristic is $x \in \mathcal{T}^\mu$, and its internal state is given by the state variables $t \in \prod_{i=1}^v D_i$. Notice that the constraints imposed over the functions Φ , σ , and \mathcal{U} enable them to be considered as functions that return the conditional probabilities.

10.3 Taxonomy

Among approximate algorithms, we can find two types: heuristics (*ad hoc*) and metaheuristics. We focus on the latter in this chapter, although we first mention *ad hoc* heuristics, which can in turn be divided into *constructive heuristics* and *local search methods* [31]. Later in this section, we present the taxonomy selected to structure the rest of the chapter, divided into *trajectory-based* and *population-based* algorithms.

10.3.1 Constructive Heuristics and Local Search Methods

Constructive heuristics are usually the swiftest methods. They construct a solution from scratch by iteratively incorporating components until a complete solution is obtained, which is returned as the algorithm output. Finding some constructive heuristics can be easy in many cases, but the obtained solutions are of low quality in general since they use simple rules for such construction. In fact, designing one such method that actually produces high-quality solutions is a nontrivial task, since it mainly depends on the problem, and requires a thorough understanding of it. For example, in problems with many constraints, it could happen that many partial solutions do not lead to any feasible solution.

Local search or gradient descent methods start from a complete solution. They rely on the concept of *neighbourhood* to explore a part of the search space defined for the current solution until they find a *local optimum*. The neighbourhood of a given solution s , denoted as $N(s)$, is the set of solutions (neighbours) that can be reached from s through the use of a specific modification operator (generally referred to as a *movement*). These are usually problem-specific movements that depend on the nature of the problem being addressed and the solution encoding—ranging from those applying a small change to the value of one of the components that comprise the solution to those modifying the order of the components of a solution. A local optimum is a solution having equal or better objective function value than any other solution in its own neighbourhood. The process of exploring the neighbourhood, finding and keeping better neighbours, is repeated until the local optimum is found. Complete exploration of a neighbourhood is often unapproachable, therefore, some modification of this generic scheme has to be adopted. Depending on the movement operator, the neighbourhood varies and so does the manner of exploring the search space, simplifying or complicating the search process as a result.

In contrast to these heuristic techniques, a metaheuristic is a general template for a stochastic process that has to be filled with specific data from the problem to be solved and that can tackle problems with high-dimensional search spaces. The *solution representation* is one of the design choices that needs to be made when designing a metaheuristic for its success. The solution encoding should be designed in a way that allows representing all possible solutions of the search space related to

the problem addressed. Arrays of binary, integer or real values are commonly used to represent solutions, but more complex ones can be required depending on the problem domain, as noted previously in Sect. 10.2.1. A proper codification is also essential for the correct operation of the defined objective function, which should be able to determine the real quality of each solution based on that information. Then, this representation has to be complemented with specific *search operators* to manipulate them so that solutions can transform into others by applying those operators. For example, as it will be explained later on in Sect. 10.5.1, evolutionary algorithms require the design of appropriate crossover and mutation operators for the solutions in the context of the problem to be solved, thereby producing new valid descendants from the selected parents.

In these techniques, success also depends on the correct balance between *diversification* and *intensification*. The term diversification refers to the evaluation of solutions in distant regions of the search space (with some distance function previously defined for the solution space); it is also known as *exploration* of the search space. The term intensification refers to the evaluation of solutions in small bounded regions, or within a neighbourhood (*exploitation* of the search space). The balance between these two opposed aspects is of utmost importance, since the algorithm has to quickly find the most promising regions (exploration), but also those promising regions have to be thoroughly searched (exploitation).

10.3.2 Trajectory-Based and Population-Based Metaheuristics

There are many ways to classify metaheuristics [4]. Depending on the chosen features, we can obtain different taxonomies: nature-inspired vs. non-nature-inspired, memory-based vs. memory-less, one or several neighbourhood structures, etc. One of the most popular classifications distinguishes *trajectory-based* metaheuristics from *population-based* ones. Those of the first type handle a single solution of the search space at a time, while those of the latter work on a set of solutions (the population). This taxonomy is graphically represented in Fig. 10.2, where the most representative techniques are also included.

In the chosen classification, we can distinguish two kinds of search strategies in metaheuristics. First, there are “intelligent” extensions of local search methods (trajectory-based metaheuristics in Fig. 10.2). These techniques add some mechanism to escape from local optima to the basic local search method (which would otherwise stick to it). *Tabu Search* (TS) [14], *Iterated Local Search* (ILS) [16], *Variable Neighbourhood Search* (VNS) [30], *Simulated Annealing* (SA) [22] or *Greedy Randomised Adaptive Search Procedure* (GRASP) [13] are some techniques of this kind. These metaheuristics operate with a single solution at a time, and one (or more) neighbourhood structures. It can be deduced from the above that simple strategies are often less resource-consuming, while complex ones offer the possibility to find bet-

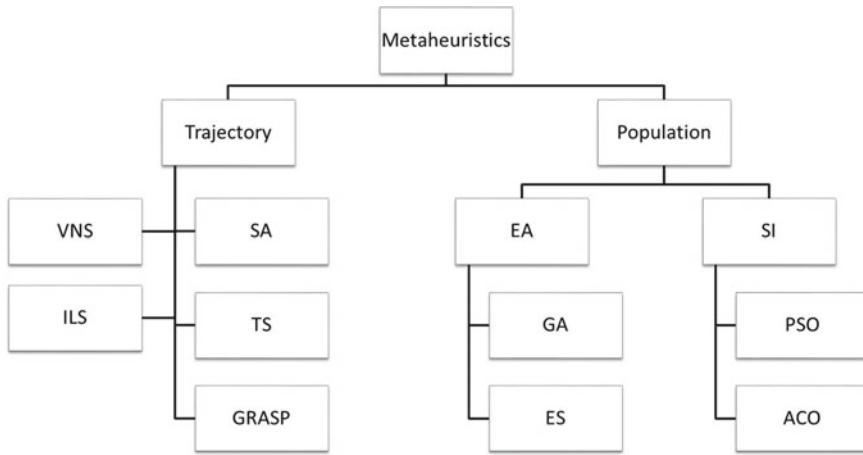


Fig. 10.2 Classification of metaheuristics

ter solutions at the expense of spending more time and/or memory (e.g., by keeping track of a number of solutions to explore).

A different strategy is followed in *Swarm Intelligence* (SI), such as *Ant Colony optimisation* (ACO) [10] and *Particle Swarm optimisation* (PSO) [6], or in *Evolutionary Algorithms* (EAs) [16], such as *Genetic Algorithms* (GA) [17] or *Evolutionary Strategies* (ES) [3]. These techniques operate with a set of solutions at any time (called colony, swarm or population, depending on the case), and use a learning factor as they, implicitly or explicitly, try to grasp the correlation between design variables in order to identify the regions of the search space with high-quality solutions (population-based techniques in Fig. 10.2). In this sense, these methods perform a biased sampling of the search space.

Regarding MOPs, specialised algorithms have been developed which are able to optimise multiple objectives at the same time. Among others, *Non-dominated Sorting Genetic Algorithm-II* (NSGA-II) [9] or *Strength Pareto Evolutionary Algorithm 2* (SPEA2) [37] are two of the most popular algorithms to this purpose.

10.4 Trajectory-Based Metaheuristics

Trajectory-based algorithms are those that seek to find a solution to a problem by iteratively improving a single valid solution. The name *trajectory* makes reference to the path that the algorithm traces out in the solution space when moving from the initial solution (generally, selected at random) to the final one. In this iterative process, the current solution is replaced by another complete solution—considered as a neighbouring configuration—until reaching a state that cannot be further optimised by following the same procedure. Therefore, these algorithms move not only from

a feasible solution to another feasible solution, but also to a better solution in an attempt to get closer to the best possible solution.

In the following, different techniques are presented to illustrate the diversity of existing trajectory-based algorithms.

10.4.1 Hill Climbing

Hill climbing [24] is one of the simplest and best-known local search algorithms, from which most of the trajectory-based techniques have been derived. The iterative process carried out by this technique reminds the movement of a person on a hill since the underlying loop continuously moves towards one direction:

- **Uphill**, when the objective function is maximised.
- **Downhill**, when the objective function is minimised.

This is because the hill-climbing strategy accepts any new state generated from the actual state that improves the objective function. At this point, *simple* hill climbing applies first-improvement local search, that is, the algorithm directly selects the first choice that improves the current solution, whereas *steepest-ascent* hill climbing—applying best-improvement local search—selects the best of all its neighbours. In the latter, the algorithm starts with an initial state (Line 1, Algorithm 1) and then generates its neighbours in the local space by changing the current state based on predefined movements (Line 4). Afterwards, it selects the best candidate based on the value of the objective function (line 5) and compares it with that of the current state (Line 6). The current state is replaced by the new one when it provides a better value for the objective function (*is better than* depends on whether the objective function is being maximised or minimised).

Algorithm 1 Pseudocode of a Hill Climbing algorithm

```

1: bestSolution ← random_solution()
2: end ← false
3: while not end do
4:   neighbours ← generate_neighbours(bestSolution)
5:   bestNeighbour ← select_best_neighbour(neighbours)
6:   if (bestNeighbour is_better_than bestSolution) then
7:     bestSolution ← bestNeighbour
8:   else
9:     end ← true
10:  end if
11: end while
12: return bestSolution

```

In this kind of algorithm, it is easy to get stuck in a local optimum. In these cases, the algorithm reaches a point where all of the neighbours of the current state have the

same or worst value for the objective function, and it stops as a result of not finding a better solution; however, there could be another solution in the search space that is a global optimum, which cannot be reached through that path and the given movements to generate the neighbourhood.

10.4.2 Iterated Local Search

A more sophisticated method than hill climbing is Iterated Local Search (ILS) [26], which allows escaping from local optima. In this method, the same local search is applied several times but starting from different initial configurations. Instead of generating a new random solution each time—i.e., a repeated local search—, ILS applies a *perturbation* to the previously obtained solution in order to produce better starting solutions for the subsequent local search process. This perturbation should be sufficient so as to escape from the found local optima and allow for the exploration of a new region of the search space and, hopefully, the discovery of a better local optimum. Figure 10.3 graphically represents the idea behind ILS; once a solution x_1 is found, the application of the perturbation leads to the generation of a new solution x_2 , which in turn will guide the local search process to find a new local optimum, x_3 .

In its basic form, ILS is usually more efficient than repeated local search [25]—especially in high-dimensional problems—, which can be explained by the fact that finding other local optima is easier when starting from other ones instead of from a random configuration. However, the perturbation should be configured carefully, because high jumps could ultimately work as a random start approach whereas small

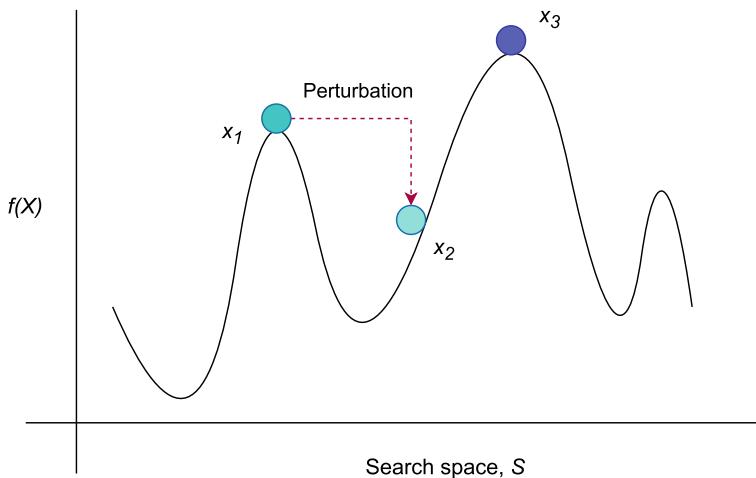


Fig. 10.3 Representation of iterated local search

changes may not be useful to foster the intended diversification of solutions (see Sect. 10.3 for a definition).

The pseudocode of ILS is shown in Algorithm 2, where a solution is generated in first place with the local search method (Line 2), followed by an iterative process (Lines 4-9). While the stopping criterion is not met, the best solution found so far is perturbed (Line 5) and then the new solution is subject to the same local search process (Line 6). In other variants of the algorithm, the perturbation could be applied to other solutions (e.g., the last solution), and not necessarily to the best solution found so far. The algorithm will accept or discard the new solution based on an acceptance test (Line 7)—usually whether the new solution improves over the current one or not. In more complex configurations of the algorithm, the loop can retain a memory of already visited states so that the perturbation scheme and even the acceptance function are adapted based on previous solutions.

Algorithm 2 Pseudocode of an ILS algorithm

```

1: bestSolution ← random_solution()
2: bestSolution ← local_search(bestSolution)
3: evals ← 0
4: repeat
5:   perturbedSolution ← perturb_solution(bestSolution)
6:   newSolution ← local_search(perturbedSolution)
7:   bestSolution ← acceptance_test(newSolution, bestSolution)
8:   evals ← evals + 1
9: until evals == totalEvals
10: return bestSolution
  
```

Hill climbing stops when no better solution is found within the neighbourhood. Unlike hill climbing, the stopping condition for this algorithm (and the rest of the methods in this section) is based on a number of evaluations, which represents the number of attempts to reach a global optimum.

10.4.3 Variable Neighbourhood Search

Instead of restarting the search, or applying a perturbation to explore other areas of the search space like in ILS, Variable Neighbourhood Search (VNS) [30] changes the system to produce new neighbours as a strategy. This change in the generation of the neighbourhood is put into practice when a local optimum is found; that local optimum could not be improved with the type of movements applied up to that point, but that solution could certainly be improved by varying the kind of movements—which may lead to solutions not yet seen. The process finishes when none of the systems designed in order to create different neighbours is able to produce an improvement given the current solution.

10.4.4 GRASP

The technique known as GRASP [13] stands for *Greedy Randomised Adaptive Search Procedure*. The iterative procedure followed by this metaheuristic algorithm consists of two phases:

1. **Constructing a solution** with the help of a greedy randomised algorithm.
2. **Improving the solution** by means of a local search.

Algorithm 3 shows the general procedure followed by this algorithm, where a solution is first generated with a greedy randomised algorithm (Line 4) and then improved through a local search (Line 5).

Algorithm 3 Pseudocode of a GRASP algorithm

```

1: bestSolution ← {}
2: evals ← 0
3: while evals < totalEvals do
4:   solution ← generate_solution()
5:   improvedSolution ← improve_local_search(solution)
6:   if (improvedSolution is_better_than bestSolution) then
7:     bestSolution ← improvedSolution
8:   end if
9:   evals ← evals + 1
10: end while
11: return bestSolution

```

Algorithm 4 Pseudocode of *generate_solution*

```

1: solution ← {}
2: while solution can be completed with new elements do
3:   rcl ← create_restricted_candidate_list()
4:   element ← select_random_element(rcl)
5:   solution ← add_element(solution, element)
6: end while
7: return solution

```

For the generation of the solution in the first phase (see Algorithm 4), the greedy algorithm selects, step by step, the components that will become part of the solution. Typically, a greedy approach is deterministic because it adds those components that are seemingly the best based on the heuristic. In GRASP, instead, the added component is randomly selected from a list of candidates, known as *restricted list of candidates* (Lines 3 and 4). This list can be configured to establish which candidates are included in it (e.g., the best n elements) and the probability for them to be selected.

Regarding the second phase, the generated solution is taken as the initial state for a local search algorithm (e.g., hill climbing). This process is repeated several times in

the hope that one of the local optima found coincides with a global optimum. Finally, the method is adaptive in the sense that the function that ranks the components can be adjusted dynamically based on previous results (in contrast to the static approach, which assigns the values in advance).

Given that GRASP starts with an initially empty solution and uses a greedy algorithm to form a valid solution in the first step, it is often classified as a constructive method, or even as a hybrid method because of combining both a greedy and a local search algorithm.

10.4.5 Tabu Search

Tabu Search (TS) [15] seeks to overcome one of the problems of local search: the small region of the search space is usually explored in each execution. To do that, this method promotes the exploration of a larger part of the space of solutions by explicitly enforcing that movements not recently taken are also selected. This is achieved by complementing the algorithm with a memory so that the following action not only depends on the current solution, but also on the solutions generated so far. The name given to the technique stems from its most distinguishing feature: some movements are temporarily forbidden at some points during the execution and, thus, they are marked as *tabu* and cannot be selected. Depending on the case, the memory can also store attributes of the solution or even the whole solution.

Likewise, the set of movements stored in memory to avoid repeated explorations of neighbours is called a tabu list. The most basic strategy to handle this tabu list is to save the movements explored in the last n steps; the movements in this list cannot be revisited in the next iteration. Among other options, another more complex strategy involves focusing on the internal solution representation and encouraging the modification of those components not subject to changes in the last iterations. The size of this list, usually called tabu tenure, determines whether the prohibition favours intensification to strengthen the evaluation of solutions close to that region (short-term list) or diversification to allow for the exploration of new distant areas (long-term list).

Algorithm 5 presents the pseudocode of Tabu Search. At the beginning of the process, an initial solution is selected to be the best solution found so far (it can be either a previously known solution or a random one) and it is also set as the best candidate (Lines 1 and 2). Once in the loop, the best neighbour of the current best solution is chosen; however, that candidate cannot be formed by using a movement contained in the tabu list. In other words, the selected movement is the one that provides a better value for the objective function which is not in the tabu list in that iteration (Line 6). For example, Fig. 10.4 shows two candidate movements (x_2 and x_5) to form the next solution. Even though x_2 is a better option than x_5 —according to the fitness function—, x_5 is finally chosen because x_2 was recently added to the tabu list.

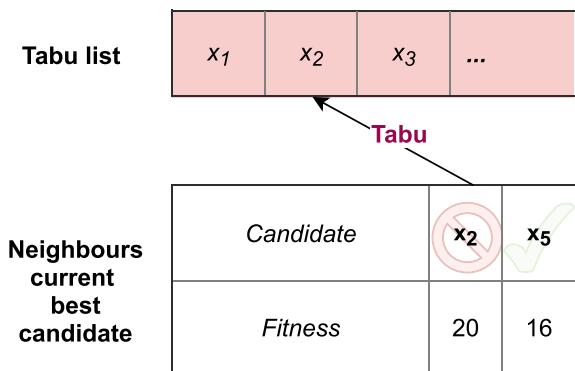
Algorithm 5 Pseudocode of a Tabu search algorithm

```

1: bestSolution ← random_solution()
2: bestCandidate ← bestSolution
3: evals ← 0
4: while evals < totalEvals do
5:   neighbours ← generate_neighbours(bestCandidate)
6:   bestCandidate ← select_best_neighbour_not_in_tabu(neighbours)
7:   if (bestCandidate is_better_than bestSolution) then
8:     bestSolution ← bestCandidate
9:   end if
10:  tabuList ← update_tabu_list(bestCandidate)
11:  evals ← evals + 1
12: end while
13: return bestSolution

```

Fig. 10.4 Influence of tabu list in the selection of the next candidate



Afterwards, the best candidate is replaced by the new solution if it has better fitness (Lines 7-9). Finally, the tabu list is updated (Line 10); this means both that the movement applied in that iteration is added to the list and that the oldest movement in the list is removed if the list is full (or if its penalisation expires, depending on how this list has been conceived). Note that, with this approach, worse solutions than the current one are allowed to be the next best candidate—the best solution found during the whole execution is maintained in another independent variable (*bestSolution*), which is returned in the end (Line 13). Thanks to this, the algorithm can have the possibility to escape from local optima, and consequently, expand the search area.

10.4.6 Simulated Annealing

Hill climbing always selects the best movements at each local step, which can eventually lead the technique to get stuck in local optima. Simulated Annealing (SA) [22] attempts to overcome this issue by allowing worse movements with a certain probability during the search. Simulated Annealing is inspired by the physical process of

annealing (e.g., cooling of metal alloys), where the material is heated and cooled in a controlled way. In the search process, the initial temperature is very high, consequently, almost any new solution (neighbour of the current solution) can be accepted, with an acceptance function that favours better solutions but also allows movements that worsen the objective function. However, as the search progresses, the temperature is gradually lowered, which means that the probability to accept worse solutions decreases. In this technique, we have to establish the *cooling strategy* to decide how the temperature evolves as the iterative process of cooling advances.

10.5 Population-Based Metaheuristics

Unlike trajectory-based methods, which handle a single solution, population-based methods are characterised by working with a set of solutions at a time, usually named *the population*. Population-based methods maintain and improve multiple candidate solutions using population characteristics to guide the search. In the following, the most common population-based algorithms are presented.

10.5.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) [2] are metaheuristic search techniques loosely based on the principles of natural evolution, namely, adaptation, and survival of the fittest. These techniques have been shown to be very effective in solving hard optimisation tasks. EAs base their operation on a set of tentative solutions (individuals) called *population*, which are evolved in the hope to form new better solutions. The problem knowledge is usually enclosed in an objective function, the so-called *fitness function* or *objective function*, which assigns a quality value to the individuals.

Algorithm 6 briefly presents the evolutionary algorithm in pseudocode. Initially, the algorithm creates, randomly or by using a seeding procedure, a population of μ individuals (Line 1). At each step, the algorithm applies stochastic operators such as selection (Line 5), recombination or crossover (Line 6), and mutation operators (Line 7) in order to compute a set of λ descendant individuals (the *offspring*):

- The objective of the selection operator is to select some individuals from the population, which will be later subject to the application of the other types of operators to create new individuals. Different kinds of selection strategies exist: *roulette wheel*, *tournament* or *random*, among others.
- The recombination operator generates a new individual from several ones by combining their solution components. This operator is able to put together good solution components that are scattered in the population.
- On the other hand, the mutation operator modifies one individual at a time and is the source of new different solution components in the population.

- Finally, the replacement operator discards the worst individuals in the old population and introduces promising ones from the new population using different strategies such as $(\mu + \lambda)$, or (μ, λ) , where μ represents the number of parent solutions and λ the number of generated children.

Recombination and mutation are common to all EAs. The only exception is Evolution Strategies, where recombination is not usual. In some cases, some EAs may use other *variation operators* like local search or *ad hoc* techniques.

Algorithm 6 Pseudocode of evolutionary algorithms

```

1: P  $\leftarrow$  initialise_population( $\mu$ )
2: while not Termination_Condition() do
3:   Q  $\leftarrow$   $\emptyset$ 
4:   for  $i \leftarrow 1$  to (popSize) do
5:     parents  $\leftarrow$  selection(P)
6:     offspring  $\leftarrow$  recombination(parents)
7:     offspring  $\leftarrow$  mutation(offspring)
8:     evaluate_fitness(offspring)
9:     insert(offspring, Q)
10:    end for
11:    P  $\leftarrow$  replacement(P, Q)
12:  end while
13: return bestSolution (P)
  
```

The individuals created are evaluated according to the fitness function (Line 8). The last step of the loop is a replacement operation (Line 12) in which the new population is formed by using the new offspring (μ, λ replacement) or by combining individuals selected from the offspring and the old population ($\mu + \lambda$ replacement). This process is repeated until a stop criterion is fulfilled (Line 2), such as reaching a pre-programmed number of iterations of the algorithm or finding an individual with a preset target quality. Depending on the individual representation and on how these phases are implemented, different instances of EAs arise such as *Genetic Algorithm* (GA) and *Evolution Strategy* (ES).

- **Genetic Algorithms** [17] appear for the first time in the early 70s as a result of the work done by John Holland, and nowadays is a widely recognised optimisation method. The genetic algorithm typically uses a parent selection method that selects the best candidate solutions and elitist replacement for the next population, that is, the best individuals of the current population are included in the next one.
- **Evolutionary Strategies** [3] follow the same general scheme shown in Algorithm 6 for EAs. It was created in the early 1960s and developed further in the 1970s by Ingo Rechenberg and Hans-Paul Schwefel [32]. In an ES, each individual is composed of a vector of real numbers representing the problem variables (\mathbf{x}), a vector of standard deviations (σ), and a vector of angles (ω). These two last vectors are used as parameters for the main operator of this technique: the Gaussian

mutation [19]. They are evolved together with the problem variables themselves, thus allowing the algorithm to self-adapt the search to the landscape.

10.5.2 Swarm Intelligence

The collective and social behaviour of living creatures motivated researchers to undertake the study of *Swarm Intelligence* [21]. Swarm intelligence techniques are based on the cooperation that takes place among a large number of homogeneous agents. Such intelligence is decentralised, self-organising, and distributed throughout an environment. In nature, such systems are commonly used to solve problems, such as effective foraging for food, prey evading, or colony re-location. The information is typically stored throughout the participating homogeneous agents, or is stored or communicated in the environment itself, such as through the use of pheromones in ants, dancing in bees, and proximity in fish and birds.

Swarm Intelligence systems are typically made up of a population of simple agents (an entity capable of performing/executing certain operations) interacting locally with each other and with their environment. Although there is normally no centralised control structure dictating how individual agents should behave, local interactions among such agents often lead to the emergence of global behaviour. The paradigm consists of two dominant sub-fields: Ant Colony optimisation (ACO) and Particle Swarm optimisation (PSO).

10.5.2.1 Ant Colony Optimisation

Ant Colony optimisation [11] is an optimisation algorithm that belongs to the family of Swarm Intelligence algorithms. They are inspired by the foraging behaviour of real ants in the search for food. The main idea consists of simulating the ants' behaviour in a graph, called construction graph, in order to search for the shortest path from an initial set of nodes to the objective ones. The cooperation between the different simulated ants is a key factor in the search which is performed indirectly by means of pheromone trails—a model of the chemicals real ants use for their communication. The main procedures of an ACO algorithm are the construction phase and the pheromone update. These two procedures are scheduled during the execution of ACO until a given stopping criterion is fulfilled. In the construction phase, each artificial ant follows a path in the construction graph. In the pheromone update, the pheromone trails of the arcs are modified, attempting to learn a probability distribution.

Before describing Algorithm 7, it is convenient to clarify some issues related to the notation. In the pseudocode, the path traversed by the k -th artificial ant is denoted with a^k . Thus, we use $|a^k|$ to refer to the length of the path, a_j^k to denote the j -th node of the path, and a_*^k to indicate the last node of the path.

Algorithm 7 Pseudocode of ACO

```

1:  $\tau \leftarrow \text{initialise\_pheromone}();$ 
2: while not termination_condition() do
3:   for  $k = 1$  to colsiz do
4:     while  $|a^k| \leq \lambda_{ant} \wedge T(a_*^k) - a^k \neq \emptyset$  do
5:       node  $\leftarrow \text{select\_neighbour}(a_*^k, T(a_*^k), \tau, \eta);$ 
6:        $a^k \leftarrow a^k + \text{node};$ 
7:     end while
8:   end for
9:    $\tau \leftarrow \text{pheromone\_evaporation}(\tau, \rho);$ 
10:   $\tau \leftarrow \text{pheromone\_update}(\tau, a^{best});$ 
11: end while
12: return  $a^{best}$ 
```

The algorithm works as follows. First, the pheromone trails are initialised in Line 1 with default values. After the initialisation, the algorithm enters a loop that is executed until a termination condition is reached, typically a given maximum number of steps (Line 2). In Line 4, we use the expression $T(a_*^k) - a^k$ to refer to the elements of $T(a_*^k)$ that are not in the sequence a^k . That is, in that expression, we interpret a^k as a set of nodes. In the loop, each ant k stochastically selects the next node (Line 5) according to the pheromone (τ_{ij}) and the heuristic value (η_{ij}) associated with each arc (i, j) . Then, the next node is selected with a probability

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{s \in T(i)} [\tau_{is}]^\alpha [\eta_{is}]^\beta}, \text{ for } j \in T(i), \quad (10.8)$$

where α and β are two parameters of the algorithm determining the relative influence of the pheromone trail and the heuristic value on the path construction, and T is the set of available transitions from i .

The whole construction phase is iterated until the ant reaches the maximum length λ_{ant} , or it fulfils the stop criterion. When all the ants have built their paths, a pheromone update phase is performed. In Line 9, all the pheromone trails are reduced, simulating the real-world evaporation of pheromone trails, according to the expression $\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}$, where ρ is the pheromone evaporation rate and it holds that $0 < \rho \leq 1$. Then, the pheromone trails associated with the arcs traversed by the best-so-far ant (a^{best}) are increased (Line 10). Once the termination condition has been met, the algorithm returns the best solution.

10.5.2.2 Particle Swarm Optimisation

PSO [12] is a population-based metaheuristic inspired by the social behaviour of birds within a flock. Even though the algorithm was initially designed for continuous optimisation problems, a *quantisation* method can be applied in case the problem

requires solutions encoded within a vector of integers. In this way, each time a new particle is generated, this quantisation transforms its continuous values into discrete ones.

In PSO, each potential solution to the problem is called a particle position and the population of particles is called the swarm. In this algorithm, each particle position x^i is updated each iteration g according to the velocity of the particle in each moment of the optimisation process. For more details on the equations that govern the update processes, see [12]. The pseudocode of PSO is introduced in Algorithm 8. The algorithm starts by initialising the swarm, which includes both the positions and velocities of the particles p_i (candidate solutions). Positions correspond to locations in the decision space and velocities are used to modify positions in future iterations.

Algorithm 8 Pseudocode of PSO

```

1: swarm  $\leftarrow$  initialise_swarm()
2: evaluate_fitness(swarm)
3: update_leaders(swarm)
4: while not termination_condition() do
5:   for i to SwarmSize() do
6:     update_velocity( $p_i$ )
7:     update_position( $p_i$ )
8:     evaluate_fitness( $p_i$ )
9:   end for
10:  update_leaders(swarm)
11: end while
12: return bestSolution (swarm)

```

The position of each particle is randomly initialised, and the leader is computed as the best particle of the swarm. Then, for a maximum number of evaluations, the velocity of each particle is updated. After that, the position is updated according to the velocities, and it is evaluated according to the strategy used. At the end of each iteration, the leader of the swarm is also updated. Finally, the best solution found so far is returned.

10.5.3 Multi-objective Algorithms

Multi-objective algorithms aim is solving a problem with more than one objective function to be optimised simultaneously. When we deal with a nontrivial multi-objective problem, no single solution exists that optimises all objectives under consideration at the same time. This is the main reason why multi-objective algorithms belong to the population-based methods, nevertheless there exist some trajectory-based methods that are used to solve multi-objective problems. In the following, we describe the most popular multi-objective metaheuristics.

10.5.3.1 Non-dominated Sorting Genetic Algorithm-II

Non-dominated Sorting Genetic Algorithm-II (NSGA-II), proposed by Deb et al. [9], is a genetic algorithm which is the reference algorithm in multi-objective optimisation (with over 38,986 citations at the time of writing¹). Its pseudocode is presented in Algorithm 9. NSGA-II makes use of a population (P) of candidate solutions. In each generation, it works by creating new individuals after applying the genetic operators to P , in order to create a new population Q (Lines 4 to 9). Then, both the current (P) and the new population (Q) are joined; the resulting population (R) is ordered according to a ranking procedure which divides the population into different non-dominated fronts and a density estimator known as crowding distance (Line 12). The crowding-distance computation requires sorting the population according to each objective function value in ascending order of magnitude to get a normalised distance among solutions. Then, the overall crowding-distance value is calculated as the sum of individual distance values corresponding to each objective (for further details, please see [9]). Finally, the population P is updated with the best individuals in R (Line 13). These steps are repeated until the termination condition is met.

Algorithm 9 Pseudocode of NSGA-II

```

1:  $P \leftarrow \text{initialise\_population}()$ 
2: while not termination_condition() do
3:    $Q \leftarrow \emptyset$ 
4:   for  $i \leftarrow 1$  to ( $\text{popSize}$ ) do
5:      $\text{parents} \leftarrow \text{selection}(P)$ 
6:      $\text{offspring} \leftarrow \text{recombination}(\text{parents})$ 
7:      $\text{offspring} \leftarrow \text{mutation}(\text{offspring})$ 
8:      $\text{evaluate\_fitness}(\text{offspring})$ 
9:      $\text{insert}(\text{offspring}, Q)$ 
10:    end for
11:     $R \leftarrow P \cup Q$ 
12:     $\text{ranking\_and\_crowding}(\text{params}, R)$ 
13:     $P \leftarrow \text{select\_best\_individuals}(\text{params}, R)$ 
14:  end while

```

10.5.3.2 Strength Pareto Evolutionary Algorithm 2

Strength Pareto Evolutionary Algorithm (SPEA2) is a multi-objective evolutionary algorithm proposed by Zitzler et al. in [37]. We show the algorithm's pseudocode in Algorithm 10. SPEA2 uses a population and an archive simultaneously in its operation. In it, each individual is assigned a fitness value (Line 4), that is the sum of its strength raw fitness and density estimation. The strength value of a solution i represents the number of solutions (in either the population P_t or the archive \bar{P}_t)

¹ Data from Google Scholar: 38,986 citations on February, 2022.

that are dominated by that solution, that is $S(i) = |\{j | j \in P_t \cup \overline{P}_t \wedge i \succ j\}|$. On the contrary, the strength raw fitness value of a given solution i is the sum of strengths of all the solutions that dominate it, and is subject to minimisation, that is, $R(i) = \sum_{j \in P_t \cup \overline{P}_t, j \succ i} S(j)$. The algorithm applies the selection, recombination, and mutation operators to fill an archive of individuals (Lines 11–13); then, the non-dominated individuals of both the original population and the archive are copied into a new population. If the number of non-dominated individuals is greater than the population size, a truncation operator (Line 7) based on calculating the distances to the k -th nearest neighbour is used (a typical value is $k = 1$), $D(i) = \frac{1}{\sigma_i^k + 2}$, where σ_i^k is the distance from solution i to its k -th nearest neighbour. This way, the individuals having the minimum distance to any other individual are chosen.

Algorithm 10 Pseudocode of SPEA2

```

1:  $t \leftarrow 0$ 
2: Initialise( $P_0, \overline{P}_0$ )
3: while not termination_condition() do
4:   evaluate_fitness( $P_t, \overline{P}_t$ )
5:    $\overline{P}_{t+1} \leftarrow \text{non\_dominated}(P_t \cup \overline{P}_{t+1})$ 
6:   if  $|\overline{P}_{t+1}| > N$  then
7:      $\overline{P}_{t+1} \leftarrow \text{truncate}(\overline{P}_{t+1})$ 
8:   else
9:      $\overline{P}_{t+1} \leftarrow \text{fill\_with\_dominated}(\overline{P}_t)$ 
10:  end if
11:  parents  $\leftarrow \text{selection}(\overline{P}_{t+1})$ 
12:  offspring  $\leftarrow \text{crossover}(\text{parents})$ 
13:   $P_{t+1} \leftarrow \text{mutation}(\text{offspring})$ 
14:   $t \leftarrow t + 1$ 
15: end while
  
```

10.6 Methodology for Evaluating Results

Metaheuristics are non-deterministic techniques, hence different executions of the same algorithm over the same problem instance can produce different results. This can cause inconveniences to researchers and practitioners at the time of evaluating and assessing those results, and when comparing the performance of different algorithms. Although there are works that tackle the theoretical analysis of many heuristic methods and problems [34], this kind of theoretical analysis still involves a great deal of complexity. Therefore, the most commonly adopted approach is to establish the comparisons on the basis of empirical data. To this end, some quality indicators have to be defined that enable such comparisons.

Once the indicators have been determined, a given number of unrelated or independent executions of the experimental configuration (algorithmic configuration and problem instance) are required to obtain statistically consistent results. A value of 30

executions is a commonly adopted and accepted minimum. The mere use of mean value and standard deviation, albeit quite frequent in the literature, may not be sufficient and can lead to wrong conclusions. Thus, a global statistical analysis should be applied on the results before stating whether the observed differences are meaningful, and not just the result of the inherent randomness of the techniques. This section contains the discussion of the indicators used in the first place (for quality and performance), then the statistical tests that are used to assess the significance of the results.

10.6.1 Quality Indicators

Quality indicators or metrics are of vital importance when assessing metaheuristics. They are defined in many ways depending on whether the optimal solution is known or unknown for the problem at hand (in a benchmark or a classic literature problem the optimum is often known, but for real problems this is hardly the case). As stated before, there are specific indicators for evaluating the solutions of single-objective and multi-objective problems.

10.6.1.1 Single-objective Indicators

When an optimum is known beforehand, a simple and intuitive quality indicator for the metaheuristic is the expectation of actually finding the optimum, or hit rate. This indicator is defined as the ratio or percentage of the number of executions in which the optimum is found over the total number of independent executions that have been performed. Another approach is to fix the goal (the objective value) and measure the effort required to reach that value. In this case, the time, or the number of fitness function evaluations are also common indicators. Unfortunately, knowing the optimum is not the common case for real problems. Furthermore, even in cases where the optimum is known, none of the executions are able to achieve it afterwards because it is quite difficult to obtain. In fact, experiments with metaheuristics are normally tailored to finish after a given computational budget has been spent, like visiting a maximum number of points of the search space, or running the algorithm for a given time or iterations.

For these cases in which the optimum is not known in advance, or that the hit rate cannot be used, we need to resort to other indicators. The most popular are the mean and median of the best fitness value found in each independent execution. In general, other statistical data are required, such as the standard deviation, and an appropriate statistical analysis should be performed to assess the statistical confidence on the observed results.

10.6.1.2 Multi-objective Indicators

In theory, a Pareto front could contain a large number of points. In practice, a usable approximate solution will only contain a limited number of them; thus an important goal is that solutions should be as close as possible to the exact Pareto front and uniformly spread; otherwise, they would not be of great help to make a decision. Besides, closeness to the Pareto front ensures that we are dealing with near-optimal solutions, while a uniform spread of the solutions means that we have made a good exploration of the objective space (i.e., no regions are left unexplored). There exist some well-known density estimators in the literature [7]: niching, adaptive grid, crowding, and the k -nearest neighbour distance.

Three different issues are normally considered for assessing the quality of the results computed by a multi-objective optimisation algorithm [36]:

1. To minimise the distance of the computed solution set by the proposed algorithm to the optimal Pareto front (convergence towards the optimal Pareto front).
2. To maximise the spread of solutions found, so that we can have a distribution as smooth and uniform as possible (diversity).
3. To maximise the number of elements of the Pareto optimal set found.

Figure 10.5 depicts these issues of convergence and diversity. The left front (a) depicts an example of good convergence and bad diversity: the approximation set contains Pareto optimal solutions but there are some unexplored regions of the objective space. The approximation set depicted on the right (b) illustrates poor convergence but good diversity: it has a diverse set of solutions but they are not Pareto optimal. Finally, the lowermost front (c) depicts an approximation front with both good convergence and diversity.

A number of quality indicators have been proposed in the literature that try to capture the three aforementioned issues, but for the moment, there is not a single metric which captures all of them. Consequently, researchers should use more than one to measure different aspects of the solutions generated by the multi-objective techniques. Among them, we can distinguish between *Pareto compliant* and *non Pareto compliant* indicators [23]. Given two Pareto fronts, A and B, if A dominates B, the value of a Pareto compliant quality indicator is higher for A than for B; meanwhile, this condition is not fulfilled by the non-compliant indicators. Thus, the use of Pareto compliant indicators should be preferable. To apply some quality indicators, it is usually necessary to know the optimal Pareto front. However, the location of the optimal front is usually unknown. Therefore, the front composed of all the non-dominated solutions computed by all analysed approaches is used to obtain a reference Pareto front which it is called the *reference Pareto optimal front*. Many quality indicators have been proposed in the literature. Next, we highlight the advantages and disadvantages of some of the most common ones:

- **Number of Pareto optimal solutions.** This non-compliant indicator is very simple as it calculates the number of solutions included in the optimal Pareto front. Its main advantage lies in the fact that it is very easy to compute. In contrast, the

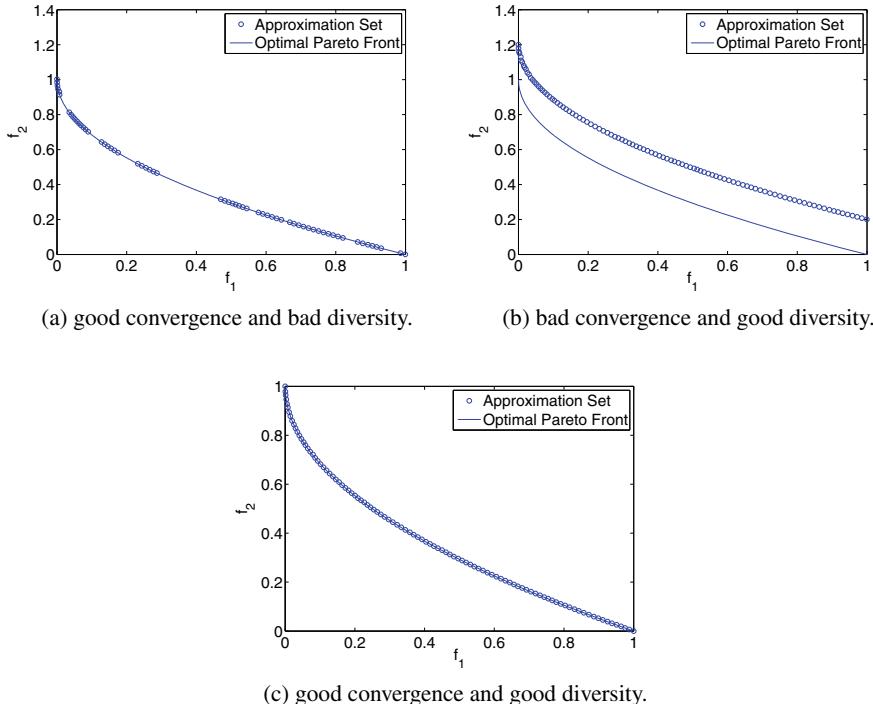


Fig. 10.5 Examples of Pareto fronts with different behaviour of convergence and diversity

disadvantages are the lack of information about the diversity of solutions and the requirement of knowing the optimal Pareto front.

- **Hypervolume (HV)** [38]. This is a very popular indicator that calculates the volume (in the objective space) covered by members of a non-dominated set of solutions Q for problems where all objectives are to be minimised. Mathematically, for each solution $i \in Q$, a hypercube v_i is constructed with a reference point W and the solution i as the diagonal corners of the hypercube. The reference point can simply be found by constructing a vector of the worst objective function values. Thereafter, a union of all hypercubes is found and its hypervolume (HV) is calculated

$$HV = \text{volume} \left(\bigcup_{i=1}^{|Q|} v_i \right). \quad (10.9)$$

We apply this metric after a normalisation of the objective function values to the range [0..1]. A Pareto front with a higher HV than another one could be due to: some solutions in the higher HV front dominate solutions in the other, or, solutions in the higher HV front are more widely distributed than in the other. Since both properties are considered to be good, algorithms with larger values of HV

are considered to be desirable. To apply this quality indicator, it is usually necessary to know the optimal Pareto front (for normalisation purposes). Of course, typically, we do not know the location of the optimal front. Therefore, we employ the *reference Pareto optimal front*.

The main advantages of the hypervolume are that it is Pareto compliant (for any reference point), it considers the convergence as well as the diversity of the solutions, and it doesn't require the optimal Pareto front. A drawback is that it depends on the reference point selected. Different reference points produce different results. This could be critical to compare the results with existing approaches in the literature.

- **Spread (SD)** [8]. It is a diversity non Pareto-compliant quality indicator that measures the distribution of individuals over the non-dominated region. This measure is based on the distance between solutions, so Pareto fronts with a smaller value of Spread are more desirable. It takes a zero value for an ideal distribution, denoting a perfect spread of the solutions in the Pareto front. It is defined as follows:

$$\Delta = \frac{d_f + d_l + \sum_{i=1}^{N-1} |d_i - \bar{d}|}{d_f + d_l + (N - 1)\bar{d}}, \quad (10.10)$$

where d_i is the Euclidean distance between consecutive solutions, \bar{d} is the mean of these distances, and d_f and d_l are the Euclidean distances of the solutions at the extremes of the Pareto front in the objective space. This indicator is defined for problems with two objectives, although it can be generalised for three or more. The main advantage of this measure is that it summarises the diversity of a Pareto front in one single scalar value. The main disadvantage is that it does not consider the other quality aspect, i.e., the solution set could be very well distributed, but the solutions could be far from the optimal Pareto front. Thus, other quality indicators should be used to complement the Spread.

10.6.2 Statistical Analysis Procedure

As mentioned before, because of their stochastic nature, we need to perform a series of independent runs for each algorithm's configuration (an accepted minimum is 30 executions) in order to obtain a distribution of results and quality indicators. Once we have the results, we must compare the distributions by means of statistical tests, which are indispensable tools to validate and to provide confidence to our empirical analysis.

A representation of the statistical procedure, which is recommended by the scientific community [33], can be seen in Fig. 10.6. First, the kind of test to perform (non-parametric or parametric) should be chosen. To do so, we perform a *Kolmogorov-Smirnov* test to check whether the samples are distributed according to a normal distribution (Gaussian) or not. After this, the homoskedasticity (i.e., equality of variances) is then checked using the *Levene* test. If all distributions are normal and the

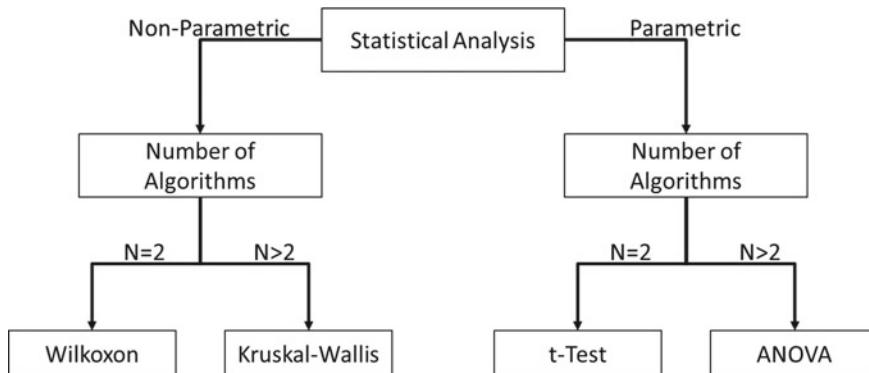


Fig. 10.6 Statistical validation procedure for experimental results

Levene returns a positive value, then we use the parametric procedure. In this case, a *t-test* for comparing two distributions, and an *ANOVA* test for comparisons of three or more distributions.

For non-parametric procedure, we use the Wilcoxon test to check the significance of the differences between two distributions, and the Kruskal-Wallis test for multiple comparisons. When kruskal-Wallis provides a negative answer (there are differences), several pairwise comparison should be performed between the algorithms. For these pairwise comparisons to have the desired significance level, the Bonferroni or Holm correction must be applied. If there are no statistical differences, the procedure finishes without rejecting the null hypothesis (equality of distributions). In the literature, tests are usually set with a confidence level of 95% or 99%, meaning that statistical differences can be found in distributions when resulted tests are with a *p-value* < 0.05 or *p-value* < 0.01, respectively.

In addition, it is always advisable to report effect size measures, which serve to properly interpret the results of statistical tests. For that purpose, we may use the non-parametric effect size measure \hat{A}_{12} statistic proposed by Vargha and Delaney [35]. It reports how often, on average, one technique outperforms the other in a range 0 to 1. It could be used to determine the probability of yielding higher performance by different algorithms. Given a performance measure M , \hat{A}_{12} estimates the probability that running algorithm A yields higher M values than running another algorithm B . If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A value of $\hat{A}_{12} = 0.3$, however, means that running algorithm A would lead to achieving higher values for M 30% of the time. Therefore, the closer this measure is to 0 or 1, the higher is the effect size and the better is expected to perform one algorithm when compared to the other.

10.7 Conclusions

This chapter has presented the working principles of metaheuristics as well as a diversity of well-known metaheuristic techniques. These techniques allow reaching near-to-optimal solutions in a reasonable time, and many studies exist that investigate the efficiency of each of them when addressing different engineering activities. Frequent real-world optimisation problems in software development are examined in the rest of the chapters of this book, pointing to useful search methods in each case.

Most of the presented metaheuristics have been explained from a general perspective in the interest of achieving the goal of describing *metaheuristics in a nutshell*. While this chapter shows the basis of these techniques, we should note that any of these techniques have to be adapted and properly configured by practitioners if they want to achieve fruitful variants for the problems faced by the industry. This means that the presented algorithms should be refined and customised as much as possible—even combining some of their characteristics—until reaching more sophisticated and problem-specific versions that provide the greatest added value to their companies in each situation and can also contribute to addressing novel aspects of software engineering.

References

1. T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms* (Oxford University Press, Oxford, UK, 1996)
2. T. Bäck, D.B. Fogel, Z. Michalewicz, *Handbook of Evolutionary Computation* (Oxford University Press, New York, 1997)
3. H.G. Beyer, H.G. Beyer, H.P. Schwefel, H.P. Schwefel, Evolution strategies - A comprehensive introduction. *Nat. Comput.* **1**(1), 3–52 (2002)
4. C. Blum, A. Roli, Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Comput. Surv.* **35**(3), 268–308 (2003)
5. F. Chicano, Metáheurísticas e Ingeniería del software. Ph.D. thesis, University of Málaga (2007)
6. M. Clerc, *Particle Swarm Optimization* (Wiley, 2010)
7. C. Coello Coello, G.B. Lamont, D. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer Series (2007)
8. K. Deb, *Multi-objective Optimization Using Evolutionary Algorithms* (Wiley, Inc., 2001)
9. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm?: NSGA-II. *IEEE Trans. Evol. Comput.* **6**(2), 182–197 (2002)
10. M. Dorigo, Optimization, learning and natural algorithms. Ph.D. thesis, Politecnico di Milano, Italy (1992)
11. M. Dorigo, T. Stützle, *Ant Colony Optimization* (The MIT Press, 2004)
12. R. Eberhart, J. Kennedy, A new optimizer using particle swarm theory, in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science* (1995), pp. 39–43
13. T.A. Feo, M.G. Resende, Greedy randomized adaptive search procedures. *J. Global Optim.* **6**(2), 109–133 (1995)
14. F. Glover, Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.* **13**(5), 533–549 (1986)
15. F. Glover, Tabu search-Part I. *ORSA J. Comput.* **1**(3), 190–206 (1989)

16. F. Glover, *Handbook of Metaheuristics* (Kluwer, 2003)
17. D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st edn. (Addison-Wesley Longman Publishing Co. Inc, Boston, 1989)
18. M. Harman, B.F. Jones, Search-based software engineering. *Inf. Softw. Technol.* **43**(14), 833–839 (2001)
19. R. Hinterding, Gaussian mutation and self-adaption for numeric genetic algorithms, in *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, vol. 1 (1995), pp. 384–389
20. J.P. Kelly, *Meta-Heuristics: Theory and Applications* (Kluwer Academic Publishers, Norwell, 1996)
21. J. Kennedy, R. Eberhart, *Swarm Intelligence* (Morgan Kaufmann Publishers, San Francisco, 2001)
22. S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing. *Science* **220**, 4598(4598), 671–680 (1983)
23. J. Knowles, L. Thiele, E. Zitzler, A tutorial on the performance assessment of stochastic multiobjective optimizers, in *TIK Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich* (2006)
24. S. Lin, B.W. Kernighan, An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.* **21**(2), 498–516 (1973)
25. H.R. Lourenço, O.C. Martin, T. Stützle, *Iterated Local Search* (Springer US, Boston, 2003), pp. 320–353
26. H.R. Lourenço, O.C. Martin, T. Stützle, Iterated local search: framework and applications, in *Handbook of Metaheuristics* (Springer, 2019), pp. 129–168
27. G. Luque, Resolución de Problemas Combinatorios con Aplicación Real en Sistemas Distribuidos. Ph.D. thesis, University of Malaga (2006)
28. P. McMinn, Search-based software testing: past, present and future, in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (IEEE, 2011), pp. 153–163
29. Z. Michalewicz, D.B. Fogel, *How to Solve It: Modern Heuristics* (Springer Inc, New York, 2004)
30. N. Mladenovic, P. Hansen, Variable neighborhood search. *Comput. Oper. Res.* **24**(11), 1097–1100 (1997)
31. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley Longman Publishing Co. Inc, Boston, 1984)
32. I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution* (Fromman-Holzboog Verlag, Stuttgart, 1973)
33. D.J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures* (Chapman & Hall/CRC, UK, 2007)
34. J. Silberholz, B. Golden, S. Gupta, X. Wang, Computational comparison of metaheuristics, in *Handbook of Metaheuristics* (Springer, 2019), pp. 581–604
35. A. Vargha, H.D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educat. Behav. Stat.* **25**(2), 101–132 (2000)
36. E. Zitzler, K. Deb, L. Thiele, Comparison of multiobjective evolutionary algorithms: empirical results. *Evol. Comput.* **8**(2), 173–95 (2000)
37. E. Zitzler, M. Laumanns, L. Thiele, SPEA2: improving the strength pareto evolutionary algorithm, in *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, vol. 103 in 1 (2001), pp. 95–100
38. E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans. Evol. Comput.* **3**(4), 257–271 (1999)

Chapter 11

Foundations of Machine Learning for Software Engineering



Aurora Ramírez and Breno Miranda

Abstract Data is everywhere. When we build software, we generate not only code artefacts and documentation, but also tonnes of data about the development process and the software itself. If we collect and analyse such data, we can better understand our own profession, make informed decisions and anticipate problematic situations. Many ways of analysing data exist, but in this chapter, we will focus on machine learning (ML). ML is the branch of artificial intelligence that gives programs the ability to solve tasks by learning from experience, without explicitly programming them. This chapter presents the elementary foundations of ML, providing the reader with a gentle but sufficiently technical introduction to its main concepts and methods. Therefore, this chapter is intended to help the reader deepen in the techniques applied in the examples presented in the rest of the book.

11.1 Introduction

The development of software systems generates a huge amount of data that, conveniently processed and analysed, can help project managers, engineers, and development teams to improve their processes and ultimately build better software. The amount and nature of the data that appear during the software life cycle are vast, mostly due to the variety of artefacts (documentation, specifications, source code, traces, etc.) that coexist in the project and the duration of the project itself.

Software analytics (SA), i.e., the application of data-driven approaches to analyse software data, is becoming increasingly popular in the last years [8]. These methods allow software practitioners to explore many forms of data and get insights from them,

A. Ramírez (✉)

Dept. Computer Science and Numerical Analysis, University of Córdoba, Córdoba, Spain

e-mail: aramirez@uco.es

B. Miranda

Informatics Center, Federal University of Pernambuco, Recife, Brazil

e-mail: bafm@cin.ufpe.br

seeking actionable information that will support project tasks or enhance the software quality [29]. Machine learning (ML) techniques are probably one of the most widely used in SA, not only because of the vast collection of algorithms and tools, but also the different types of analyses that can be conducted. With the increasing adoption of software repositories and automatic processes, the ability of ML to combine sources of information and digest fast-generated data also become relevant factor. To focus the reader on the scope of this chapter, we next present a list of exemplary situations in which ML is applicable:

- A project manager should estimate the time and effort required to complete a task based on past project experiences.
- A maintainer needs support to detect code clones so that he/she can improve the quality of the code during refactoring.
- A developer wants to know if his/her last change is likely to introduce bugs or produce build crashes before doing a commit.
- A tester would like to predict which test cases are more prone to fail, so that he/she can prioritise them.

In this chapter, the reader is introduced to the main concepts required to address the practical use of ML for solving these types of tasks. Therefore, a general overview of the ML pipeline is presented first. From that, the reader will be able to distinguish the different types of learning approaches depending on the goal(s) pursued. In the list of examples above, a reader could easily identify different purposes (“estimate”, “detect”, and “predict”), which will determine the suitable learning approach. Choosing the right ML algorithm is an essential step that also depends on factors like the frequency of data update or the availability of a “ground truth”, and usually require preliminary experimentation to find the best configuration [29]. In this sense, important methodological aspects are included in the chapter with the aim of explaining how to prepare the data, build ML models, evaluate their quality, and interpret their results. The rest of the chapter is focused on explaining specific techniques, with special emphasis on those applied in the rest of the book.

The field of ML is extensive and very active, so trying to cover in detail all existing approaches and techniques in one chapter is not possible. A good number of manuals and reference books exist, so we will refer to them for detailed algorithmic descriptions and advanced topics for interested readers. On the other hand, we will mention some complementary techniques that appear in the case studies in this book in combination with ML, but might not be necessarily found in ML books. More specifically, natural language processing (NLP) will be briefly described for the sake of completeness.

11.2 The ML Pipeline

This section provides the reader with a global understanding of the role of ML in the general process of knowledge discovery. First, an overview of the ML pipeline is presented. Next, each step in the pipeline is described, focusing on those supported by computational techniques: *data preprocessing* and *learning from data*.

11.2.1 Overview

Tom Michell defines *learning* as the ability of a computer program to “improve its performance at some task through experience” [30]. More formally, he provides the following definition:

“A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E ”.

Machine learning then consists in finding the logic that makes such learning happen. The application of an ML algorithm is not an isolated activity, but part of a data-driven process that starts from the identification of a problem and leads to its resolution. Figure 11.1 shows how running the ML algorithm is embedded into the so-called *Knowledge Discovery from Data* (KDD) process [19]. Here, the KDD process has been adapted assuming that ML is the approach to extract knowledge from the data, but other statistical techniques or data science methods are applicable too. The process is usually represented as a sequence of five steps with feedback to previous ones, as some aspects might need redefinition after getting insights from the results.

Next, we briefly present the purpose of each step:

1. **Problem definition.** This step implies setting goals that will determine the variables to be measured, and how they will be processed thereafter. Therefore, this step is highly dependant on the application domain and requires expertise in order to guide the next steps in the process [33].
2. **Data collection and exploration.** Once the problem has been specified, the data must be collected, i.e., observations from the selected variables are measured and stored [33]. In the context of software engineering (SE), data can come from sources like documentation and formal specifications, repositories for code development (issue trackers, commit information, etc.), tools for automated testing, and logging systems to name a few. In the ML terminology, the variables extracted from these sources are known as *features*, i.e., the characteristics that represent an object. Another essential term is *instance*, which refers to the feature values

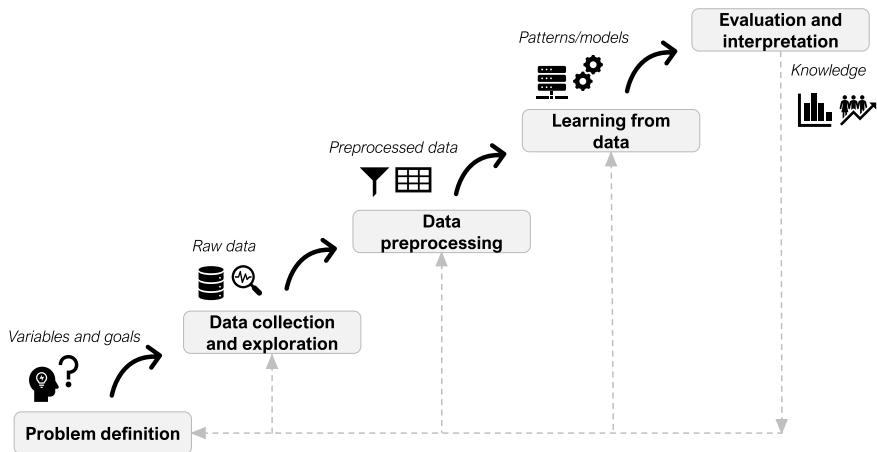


Fig. 11.1 Overview of the ML pipeline, adapted from the knowledge discovery process

measured for a particular object. Data collection also involves other aspects, such as the frequency of measurement and the type of variable. This latter decision is particularly relevant as the nature of the software data is quite heterogeneous.

- 3. Data preprocessing.** Data preprocessing comprises a number of procedures to convert raw data into a *data set* that can be effectively managed by the ML algorithm [14]. Although time-consuming, enhancing the completeness and consistency of data has positive effects on the learning results. Preprocessing tasks are usually classified into two areas, namely, data preparation and data reduction, which are covered in Sect. 11.2.2.
- 4. Learning from data.** The ML algorithm is applied over the preprocessed data set with the aim of discovering patterns and making inferences, which will be later used to describe the data instances or make predictions. The learning paradigm and the choice of the algorithm strongly depend on the learning purpose, as will be detailed in Sect. 11.2.3. One general aspect affecting most of the ML algorithms is the presence of hyperparameters¹ that should be fine-tuned to understand their influence in the learning process. Good practices in hyperparameter tuning and validation procedures should be considered during this step to guarantee not only high accuracy, but also generalisability of the conclusions derived from the data.
- 5. Evaluation and interpretation.** Evaluation does not only mean computing performance metrics to quantitatively assess how effective the algorithm is when solving the target problem, but also analysing critical areas of the data distribution that might affect the behaviour of the algorithm [33]. Often, evaluation leads

¹ In the ML literature, the term *hyperparameter* refers to a variable whose value is defined by the user and controls the training process (e.g., the number of epochs for training a neural network), whereas the term *parameter* is generally used to refer to those variables whose values are learned during model training (e.g., the weights of each neuron).

to an iterative refinement of the learning process, e.g., revisiting the features and adjusting the algorithm hyperparameters, to produce robust results.

11.2.2 Data Preprocessing

Data preprocessing is a key step in the ML pipeline to ensure the quality of the data and, therefore, it takes most of the efforts [14]. The following sections explain common data preprocessing tasks, divided into data preparation and data reduction. In addition, a brief introduction to NLP is presented, as its methods and tools are useful for preprocessing text data.

11.2.2.1 Data Preparation

Data preparation tasks are needed in almost all applications to ensure that the collected instances represent realistic observations and the specific values can be easily handled by the ML algorithm. If the data set is not well prepared, the algorithm might exhibit a wrong behaviour and the results might not be accurate enough or even incorrect [14]. Specific tasks within this category are illustrated in Fig. 11.2 and described as follows:

- Cleaning, whose objective is filtering certain instance values, handling missing values, and detecting wrong measurements due to noise. For missing values, imputation methods replace the value with an estimation or a representative value. Noise

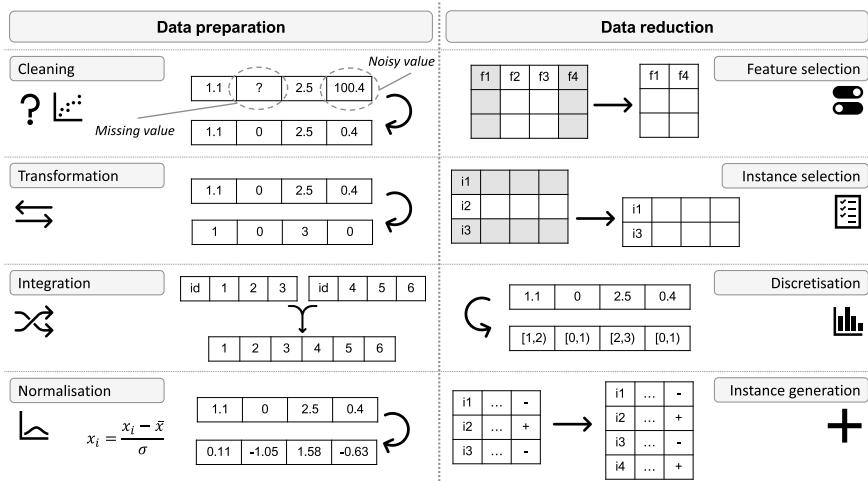


Fig. 11.2 Common tasks in data preparation and reduction

identification and correction might require more sophisticated methods, which can be based on signal smoothing.

- Transformation, whose goal is converting the instance values to new ones so that learning becomes more consistent and efficient. Notice that some preprocessing tasks, such as normalisation and discretisation, can also be viewed as subtasks of data transformation, but they are usually defined separately as they perform specific operations and are frequently applied.
- Integration, which consists in combining values from different data sources to create the instances for learning. As part of this task, we might need to identify and unify variables, remove duplicate values, and check possible inconsistencies.
- Normalisation, which takes the instance values and unifies or scales them to guarantee they lie on a particular range. For instance, instance values can be transformed to follow a normal distribution. Some ML algorithms are sensible to the variable scale, so this task prevents bias in the importance each feature will have.

11.2.2.2 Data Reduction

In contrast to data preparation, the objective of data reduction tasks is to decrease the amount of data to be used for learning, either in the feature or instance dimension [14]. In essence, valid results will be obtained if data reduction tasks are not performed, but they can greatly help reduce computational time and mitigate the effects of feature correlation (two or more features that exhibit the same behaviour in the data distribution) and imbalance (instance values for one feature are far less frequent than others). Learning from imbalanced data does not only require some preprocessing steps, but also applying specific algorithms and assessment metrics able to take the class distribution into account [22, 23].² Figure 11.2 shows examples of the four data reduction tasks described next:

- Feature selection (FS), whose techniques analyse the set of features in the data set to keep only a subset of them. For instance, irrelevant (e.g., autoincrement identifiers in databases) or redundant features (those that are correlated) are removed in this step. Also, features can be combined to reduce the *dimensionality* of the data set. Some popular methods for dimensionality reduction are principal component analysis (PCA), singular value decomposition (SVD), and linear discriminant analysis (LDA) [20]. FS techniques try to keep the performance of the ML algorithm while preserving the original distribution of the data. Techniques for FS are broadly classified into three categories: (1) filters, which only analyse the characteristics and relationships of the features in the data set; (2) wrappers, which evaluate the performance of the ML algorithm under different subsets of features; and (3) hybrid, which combines both approaches. In addition, they can be specifically designed for a learning task, such as classification [38] and clustering [20].

² The specific problem of class imbalance and its implications in the performance evaluation of classifiers is analysed in Sect. 11.3.2.

- Instance selection, which operates at the instance level to choose a representative sample of them. The application of sampling methods is highly recommended in supervised scenarios to internally validate the learning process and avoid *overfitting*. This phenomenon happens when the ML algorithm is excessively well adjusted to the structure of the data used for learning, but performs poorly in unseen data, i.e., it lacks the ability to generalise. Strategies for classifier validation are covered in Sect. 11.3.2.
- Discretisation, which transforms numerical values with a continuous space into a finite number of intervals (called *bins*) [28]. For this reason, it can also be viewed as a data preparation task. However, the underlying idea is to reduce the range of values to a few ones, possibly enhancing the generalisation capability of the ML algorithm. Several strategies exist to define the bins, the most popular are: (1) by frequency, so that each bin has the same number of unique values; and (2) by size, which creates bins of equal size.
- Instance generation, which artificially produces new instances to complement the data set. In particular, oversampling is especially relevant in classification for dealing with an imbalanced data set, since it creates copies of instances associated with the minority class. The opposite approach, undersampling, deletes instances belonging to the majority class. It should be clarified that not all resampling methods are purely random [22]. SMOTE, an oversampling algorithm that has been applied for prediction problems in SE [25], creates the synthetic instances exploring the vicinity of the minority instances.

11.2.2.3 Natural Language Processing

Textual data frequently appear in the artefacts that software engineers manage to make their decisions. Specifications, documentation, and even the code contain words and structures that require specific preprocessing if we want to use them for ML. Text mining and NLP are complementary areas of research that provide methods to manipulate data in text form [13, 26]. Text mining supports the discovery and extraction of knowledge from unstructured text, while NLP is more focused on providing an understanding of the semantic meaning of a text or speech.

The fact that text is a complex structure comprised of inner elements (words and their relations) implies that specific steps have to be executed to build instances from the raw data. Some of the recurrent steps to automatically process text include word tokenisation, removal of punctuation symbols and stop words, and stemming, i.e., reducing the words to their roots to keep only the underlying concepts.

Even after executing these steps, the list of words representing a text can be large, and more importantly, ML algorithms typically work with numerical features or only a few values for categorical ones. We still need a representation of the text that is amenable to learning, closer to the traditional concept of feature set. A first approach, namely, *bag-of-words* (BoW), simply creates a numerical feature for each word, the instance value representing how many times the word appears in the text. However, this approach leads to very sparse data sets and complicates the subsequent learning,

since the total number of words (the so-called *vocabulary*) is huge in a corpus with several texts but very few words appear in each one. Another problem with BoW is that it does not preserve sentence structures and does not take word ordering into account, which are relevant aspects when processing text.

Word embedding is the term used in NLP to define how text data is mapped to a semantically-relevant and compact numerical representation. The choice of the word embedding algorithm might vary depending on the characteristics of the texts under analysis or the ML algorithm to be used. Developed by Google, *word2vec* analyses the semantic similarity between words using neural networks and has already been trained with software-related text from Stack Overflow [12]. Another general-purpose method is *GloVe*,³ based on the analysis of co-occurrence of words. *Doc2vec* is an extension of *word2vec*, which also produces a “paragraph” vector that takes word order into account. Finally, *code2vec* is particularly adapted to work with code in textual form, providing a short list of terms describing the code purpose [6].

11.2.3 Learning from Data

As happens with human learning, ML can follow a variety of strategies to achieve different purposes. Figure 11.3 compiles the most relevant concepts related to this step, inspired by the 2012 ACM Computing Classification System.⁴ Due to space limitations, this chapter covers in detail only those techniques appearing in the rest of the book chapters. However, our intention with this (still) non-exhaustive classification is to highlight the broad scope of ML, giving the reader the basic knowledge to explore the approaches more suitable for different SE problems.

From the perspective of learning purpose, *classification*, *regression*, and *clustering* are the most common tasks appearing in SE. Classification refers to the problem of learning the distinctive features of different types of objects, so that a *class label* can be assigned to unseen instances [10, 27]. The characteristics of this type of task, and the techniques used to build classifiers will be extensively covered in Sect. 11.3. Regression is the process of adjusting a linear or nonlinear model to predict a continuous variable from the input features [21]. This task is the core subject of Sect. 11.4. Finally, the goal pursued in clustering is to discover groups of similar instances, without knowing to which class (if any) the instances belong to [2]. Sect. 11.5 presents the basics of clustering.

The type of task is intrinsically related to the second dimension, which refers to the nature of the data. Regression and classification are *supervised* methods, meaning the true output value or class label is available during learning to fit the curve or build the classifier, respectively. In contrast, clustering does not assume such information is available, i.e., it handles data in an *unsupervised* way. Other tasks solved by following an unsupervised approach are anomaly detection and association rule mining. A third

³ GloVe: <https://nlp.stanford.edu/projects/glove/> (Last accessed March 7, 2022).

⁴ ACM CCS 2012: <https://dl.acm.org/ccs> (Last accessed March 7, 2022).

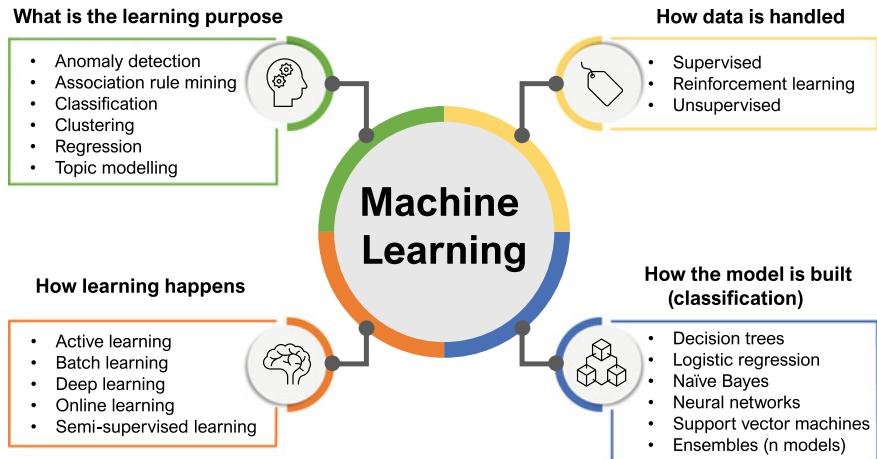


Fig. 11.3 Classification of ML paradigms, adapted from the ACM Computing Classification System

possibility is *reinforcement learning*, which learns from the rewards obtained after taking actions driven by the data, instead of the instances themselves. This type of learning is slightly different in its methodology and techniques, so we refer the reader to dedicated chapters [27, 34] and a specialised book [37].

With a clear task defined and a data set to learn from, the third dimension is focused on the diverse strategies that can be used for learning. In *batch learning*, historical data is available from the beginning and, in the case of supervised methods, partitions are used in different learning stages to build the models and evaluate them, as will be explained for classification in Sect. 11.3.2. If data continuously arrives in a stream fashion, *online learning* is the suitable approach to incrementally analyse the data. *Active learning* and *semi-supervised learning* can be viewed as adaptations of the traditional batch approach to make it more flexible when labelled data is not so easily available. On the one hand, active learning obtains the labels from an oracle (e.g., a human) at an associated cost [35]. On the other hand, semi-supervised learning is suitable in a scenario in which only a small proportion of the data is labelled [11]. The most recent learning paradigm is *deep learning* (DL) [15], which infers complex relations between the objects from simple ones following a layer structure. DL will be briefly covered in Sect. 11.6.

Finally, each type of learning task usually counts with a collection of algorithmic techniques to learn under a particular approach, and the choice will determine the type of decision model (if any) that is generated. For supervised classification, the focus of this chapter, Fig. 11.3 provides a list of popular methods. Section 11.3.1 will cover two of them: decision trees and neural networks.⁵ We refer the reader to ML books for the rest of the techniques, such as [10, 21, 27, 34].

⁵ Support vector machines are introduced in Sect. 11.4.1 in the context of regression tasks.

11.3 Classification

Classification is the action or process of classifying something according to shared qualities or characteristics. In the context of ML, classification is essentially the act of assigning entities—these could be people or objects—to one of a number of categories—usually called *classes*.

When we talk about ML-based classification, we usually refer to the task of teaching machines to classify entities into their appropriate classes. This is attractive because many practical decision-making tasks can be thought as a classification problem: one might want to classify emails as either spam or not-spam; or classify customers into different categories based on their buying patterns; or classify people based on their likelihood of getting a certain illness; or classify documents into different categories. Before a machine can perform any of these tasks, however, it needs to “learn” how to distinguish the entities into different classes.

Classification is a type of supervised learning, i.e., the algorithms are trained—or “supervised”—on a labelled data set to learn how to classify data. The algorithm repeatedly makes predictions on the training data and adjusts for the correct answer, and this is how it learns. By using labelled inputs and outputs, the algorithm can also measure its accuracy and improve it over time.

ML classifiers require initial human intervention to label the data appropriately and accurately. For example, a classification model can distinguish a legitimate email from a spam, but first, it needs to be trained to know what characteristics make an email be considered spam. If, on the one hand, this could be considered as a disadvantage when compared with unsupervised learning models (discussed in Sect. 11.5), on the other hand, supervised learning models tend to be more accurate than unsupervised ones.

In this chapter, we will focus on introducing two commonly used classification algorithms: decision trees and neural networks.

11.3.1 Algorithms

Many classification algorithms have been proposed in the literature [3, 10, 27, 34] and they can be grouped differently depending on the perspective one wants to explore. A popular class of classification algorithms is known as *linear* classification. A linear classifier is one that makes its classification based on a linear function of the features. If it helps the reader to think graphically about this, try to visualise a 2D plot where each point represents an instance that belongs to one of two classes (e.g., positive or negative); now, mentally draw a straight line that perfectly splits the instances into two different groups, each group to one side of the line. If such a line exists, we say that the two classes are *linearly separable*. Of course, this is an oversimplification and, for practical problems, the number of dimensions and classes can be much bigger.

Among popular linear methods for classification, we can cite logistic regression and support vector machines.⁶

For many other problems, however, the data are not linearly separable, i.e., the decision boundaries have a complex shape and a straight line cannot perfectly distinguish the two classes. For those cases, *nonlinear* functions can be applied to separate the instances. Decision trees and artificial neural networks are examples of nonlinear classifiers.

Another popular class is that of *probabilistic* classifiers. Algorithms in this class output a probability of an instance belonging to each of the possible classes; the class with the highest probability is usually selected as the one to label the instance. Naïve Bayes falls into this category. Other algorithms compute the *similarity* between the new instances and previous observations to make the classification, without actually building a model. An example of this type of “lazy” classifier is k-nearest neighbours.

Deciding which algorithm to use will depend on many factors, including the type of classification problem, the desired output, the type of instances, and the number of features, among others. In fact, sometimes it is preferable to use a combination of classifiers, instead of a single one, to improve the predictive performance. Such a combination of multiple classification models is called *ensemble*.

11.3.1.1 Decision Trees

A decision tree is a classification model that uses a flowchart-like structure to represent multiple decision paths. As the name suggests, the model is commonly represented as an upside-down tree structure with the root node at the top and leaf nodes at the bottom. A node that is neither the root nor a leaf node is called an internal (or *test*) node.

To understand how decision trees can be induced from training data, let us take a look at Table 11.1. The data in Table 11.1 comes from a fictitious evaluation of 10 SE projects that are classified as either *successful* or *unsuccessful*. The projects are described according to three categorical features: the quality of the project requirements (column “Req. Quality”), which can be one of *high*, *medium*, or *low*; the dominant experience of the development team (column “Dev. Experience”), which can be either *experts* or *novices*; and the adopted testing strategy (column “Testing Team”), which can be either *in-house*, in case the testing activities are carried out by the same company that developed the project, or *independent*, if the testing tasks are outsourced to an external testing team.

Figures 11.4a and b are examples of decision trees that could be induced from the data in Table 11.1. The basic algorithm to build a decision tree finds a feature and splits the input instances into subsets, each characterised by a different value of the feature. This process is repeated for all non-leaf nodes until all the instances in any given subset belong to the same class. Many strategies exist to select the feature to

⁶ With the help of some tricks, support vector machines, and other linear classifiers, can also deal with non-linearly separable data.

Table 11.1 Training data of (fictitious) SE projects classified as successful or unsuccessful based on three features

Instance	Req. quality	Dev. experience	Testing team	Class
i_1	High	Experts	In-house	Successful
i_2	Medium	Experts	In-house	Successful
i_3	High	Novices	In-house	Successful
i_4	Medium	Novices	In-house	Unsuccessful
i_5	High	Experts	Independent	Successful
i_6	Low	Novices	In-house	Unsuccessful
i_7	High	Novices	Independent	Successful
i_8	Medium	Experts	Independent	Successful
i_9	Medium	Novices	Independent	Unsuccessful
i_{10}	Low	Novices	Independent	Unsuccessful

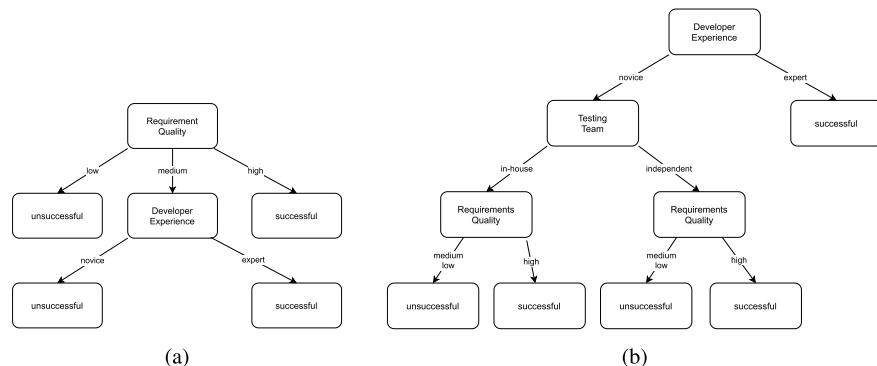


Fig. 11.4 Examples of decision trees induced from the data in Table 11.1

split on at each stage. One common approach is to maximise *information gain*—a measure of how much we know about the class to be predicted after looking at the feature at hand—by minimising the value of entropy.⁷ Other measures such as Gini index and Chi-square are also commonly adopted as splitting strategies. It is beyond the scope of this chapter to explain how each of these measures are used for selecting the feature to split on. For a detailed explanation of this process, please refer to [10, 27, 34].

Notice that, despite being different in terms of size of the tree and the number and order of the test nodes, the trees from Fig. 11.4 are both able to correctly classify all the instances from Table 11.1, i.e., they can correctly distinguish a successful project from an unsuccessful one based on the project characteristics.

⁷ Information gain is the strategy used by the famous C4.5 algorithm, known as J48 in the Weka tool: <https://www.cs.waikato.ac.nz/ml/weka/> (Last accessed March 7, 2022).

Consider the decision tree Fig. 11.4a, for example: The root node asks about the requirements quality. From Table 11.1 we can see that projects i_1 , i_3 , i_5 , and i_7 have *high* quality requirements, which coincide with the right edge *high*. The right edge ends in a leaf node labelled *successful*, which is precisely the cases of projects i_1 , i_3 , i_5 , and i_7 . Projects i_6 , and i_{10} , on the other hand, have *low* quality requirements, which coincide with the left edge *low*. The left edge ends in a leaf node labelled *unsuccessful*. The classification of projects i_2 , i_4 , i_8 , and i_9 is a bit trickier because the centre edge—the one that matches with *medium* requirements quality—does not end in a leaf node; instead, it leads to another test node that asks about the developer experience. The projects with *experts*, i_2 and i_8 , are sent through the right edge that leads to a leaf node labelled *successful*, whereas the projects with *novices*, i_4 and i_9 , are sent through the left edge that leads to a leaf node labelled *unsuccessful*. We invite the reader to perform the exercise of verifying if the decision tree Fig. 11.4b also identifies the correct classes for the projects from Table 11.1.

A remarkable feature of a decision tree is *interpretability*. It is easy to explain to someone that i_7 is classified as *successful* because *the requirements are of high quality*. In fact, a decision tree can be interpreted as a set of rules in an *if-then* format. Each leaf node corresponds to a classification rule. Thus, for the tree Fig. 11.4a the four classification rules are:

```
if Requirement Quality = high then class = successful
if Requirement Quality = medium and Developer Experience = expert then class = successful
if Requirement Quality = medium and Developer Experience = novice then class = unsuccessful
if Requirement Quality = low then class = unsuccessful
```

Such an easy way of interpreting why a given instance has been assigned to a particular class is not always possible (e.g., that is the case of artificial neural networks discussed next in Sect. 11.3.1.2).

As the reader might have noticed, the choice of the root node and the subsequent test nodes can highly influence the size and shape of the induced tree. Generally, smaller trees are preferred because they are easier to interpret and because they avoid irrelevant information. In the examples from Fig. 11.4, for example, *testing team* is not used in tree Fig. 11.4a, but it is a test node in tree Fig. 11.4b, and both trees are equally good at classifying the data from Table 11.1. Besides that, larger trees are more prone to *overfitting*: the induced tree is so specialised to the training data that it performs poorly when applied to previously unseen instances. A common approach for dealing with the problem of overfitting is called *pruning*, which aims to eliminate nodes that are not clearly relevant [34]. Pruning can be applied while inducing the tree to prevent the generation of irrelevant branches (*pre-pruning*); or to remove irrelevant branches after the tree is generated (*post-pruning*). For additional details on overfitting and pruning please refer to Chap. 9 of [10].

11.3.1.2 Artificial Neural Networks

Artificial neural networks (ANNs) embrace a large class of models and learning methods that, as the name suggests, are inspired by a simplification of the biological brain. The *neural* part refers to the units of an ANN, the *neurons*. The *network* part refers to the way those neurons are connected to each other. We will use Fig. 11.5 to visually explore the main structures of an ANN as well as to introduce some key concepts.

Figure 11.5 illustrates a simple ANN. Precisely, it illustrates a Multilayer Perceptron (MLP), which is likely the most classical type of a neural network. The MLP is composed of at least three layers of nodes. The leftmost column of nodes in Fig. 11.5 is the *input layer*, which receives the input data to be processed by the network. The rightmost column is the *output layer*, the one responsible for predicting the final output. In the centre, we have the *hidden layers*, which are responsible for performing most of the computation required for predicting the output. In this example, we have a single hidden layer, but it is quite common to employ a bigger number of hidden layers—different layers may perform different transformations on their inputs. There are many types of specific ANNs proposed in the literature and they are usually grouped into different classes. Other classes of ANNs, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), are discussed in Sect. 11.6.

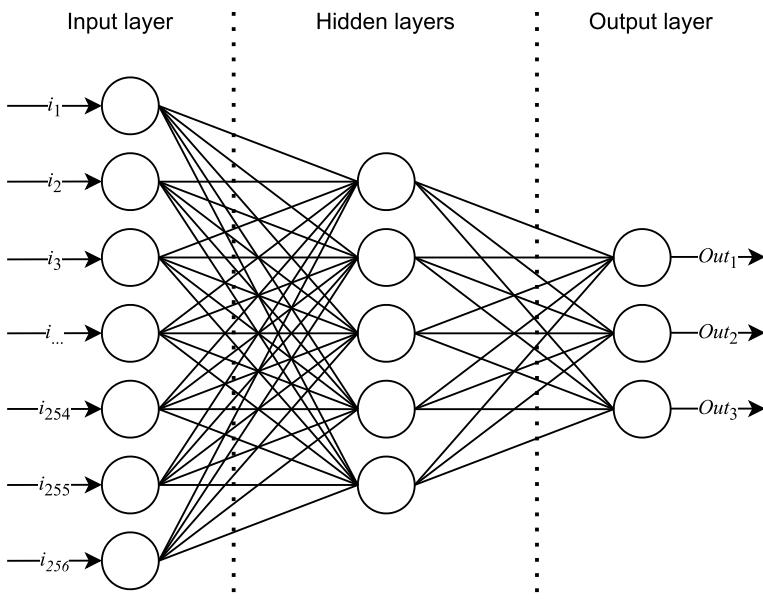


Fig. 11.5 A simple artificial neural network with a single hidden layer

The building block of an ANN is the neuron (all the nodes in the network, except those from the input layer, are neurons). Neurons within the same layer do not communicate with each other, but adjacent layers are fully interconnected through a weighted neuron-to-neuron link, i.e., each neuron in one layer connects, with a certain weight, to every neuron in the next layer. Each neuron is, thus, a computational unit that receives weighted input signals and produces an output signal guided by an *activation function*—each neuron is associated with a numerical value called *bias* that is also considered for producing the output. Different activation functions (sometimes called *transfer function*) can be used. Commonly, sigmoid functions (“S”-shaped curves) are adopted, but we will refrain from providing the equations and other mathematical details in this section. Rather, we invite the reader to focus on the overall process adopted by an ANN. After all the neurons in one layer compute the output, the ANN passes to the next layer. This process is repeated until the values in the last layer are used to determine the final output. The process of propagating the input values up to the network’s output is called *forward propagation*.

The number of neurons and the activation function in the output layer will highly depend on the problem at hand. For example, for a binary classification problem, a single output neuron and an activation function that outputs a value between 0 and 1 could be enough (class 1 or 0 is assigned depending on the output value being above or below a given threshold, say 0.5, for example). For a multi-class classification problem, which is the case illustrated by Fig. 11.5, we may have multiple neurons in the output layer, one for each class.

For training an ANN, a supervised learning technique called *backpropagation* is used. This is how it works: The training instances are presented one by one to the ANN via forward propagation. For each input, the ANN predicts the output class and compares it against the actual class. Now, suppose that the predicted class does not match with the actual one, i.e., the ANN misclassified the input. When this happens, the discrepancy between the predicted and expected values are computed and later used to instruct the ANN on how to modify the weights. Because each neuron contributes differently to the prediction, the idea is to compute the contribution of each neuron to the overall error and adjust the weights accordingly, i.e., links to neurons that have a big influence in the wrong prediction undergo greater weight changes than those with a small influence in the error. After the weights of the ANN are modified, the next input is provided. This process is repeated for all the inputs in the training set. When we talk about the learning process of an ANN what that is referring to is getting the computer to find a valid setting for all of the weights such that it maximises the number of correct predictions in the training set. This is achieved by this combination of forward propagation of inputs and backpropagation.

ANNs are complex enough to justify entire chapters, or even an entire book, on their own. In this section, we tried to address the underlying idea and main concepts of ANNs without going into the mathematics behind them. We invite the interested reader to refer to [1, 21, 27].

11.3.2 Evaluation

The most basic metrics used for estimating the performance of a classifier are *classification accuracy* (also called *predictive accuracy*, or simply *accuracy*) and *error rate* [10, 27]. Before we explain these two metrics, however, let us first analyse the possible outcomes that we can observe when we apply a classifier to entities whose real class are known. The only possible outcomes are the following:

- a positive example is correctly recognised as such by the classifier (*true positive*);
- a negative example is correctly recognised as such by the classifier (*true negative*);
- a positive example is labelled negative by the classifier (*false negative*);
- a negative example is labelled positive by the classifier (*false positive*).

Each of these four outcomes will arise at different number of times when we apply a classifier to a set of instances, and these quantities can be used to evaluate the classifier's performance. These outcomes can be displayed in a tabular format called *confusion matrix* that helps us to visualise how frequently the classifier accurately predicted the class of an example and how frequently the examples were misclassified.

Table 11.2 shows the structure of the confusion matrix where N_{TP} is the number of *true positives*, N_{TN} is the number of *true negatives*, N_{FP} is the number of *false positives*, and N_{FN} is the number of *false negatives*. Because any prediction done by the classifier will fit in one of these four categories, the size of the set provided as input to the classifier equals the sum $N_{TP} + N_{TN} + N_{FP} + N_{FN}$.

The number of correct predictions is the number of times that the class predicted by the classifier is, indeed, the correct class, i.e., the number of *true positives* plus the number of *true negatives* ($N_{TP} + N_{TN}$); the number of errors, on its turn, is the number of times that the predicted class does not correspond to the actual class, i.e., the number of *false positives* plus the number of *false negatives* ($N_{FP} + N_{FN}$).

Classification accuracy is the proportion of correct predictions made by the classifier over the whole set of predictions. It is calculated by dividing the number of correct predictions by the total number of predictions (see Eq. 11.1).

$$\text{Accuracy} = \frac{N_{TP} + N_{TN}}{N_{TP} + N_{TN} + N_{FP} + N_{FN}} \quad (11.1)$$

Table 11.2 Structure of the confusion matrix

		Predicted class
Correct class	Positive	Negative
Positive	N_{TP}	N_{FN}
Negative	N_{FP}	N_{TN}

Error rate, E , is the proportion of wrong predictions made by the classifier over the whole set of predictions. It is calculated by dividing the number of errors by the total number of predictions (see Eq. 11.2).

$$E = \frac{N_{FP} + N_{FN}}{N_{TP} + N_{TN} + N_{FP} + N_{FN}} \quad (11.2)$$

Notice that classification accuracy and error rate are complements of each other. This means that one can compute accuracy as $accuracy = 1 - E$. Analogously, the error rate can be computed with respect to accuracy as $E = 1 - accuracy$.

11.3.2.1 Going Beyond Classification Accuracy and Error Rate

In some application domains, the number of instances on each class is imbalanced, i.e., the number of instances in the negative class outnumbers the positive class, or vice-versa. Consider, for example, a data set that maps the presence or absence of a rare disease to patients. In such a data set, the vast majority of the patients, say 99%, are healthy. Now, suppose that we build a classifier to distinguish future patients between healthy and unhealthy. If we aim for high accuracy and low error rate, we could build a simple model that *always* predicts that the patient is healthy. Such a model would achieve the remarkable figure of 99% accuracy. However, the reader will agree that this is a useless classifier: it never recognises when a patient is unhealthy and this could have serious consequences.

Domains with imbalanced classes are very common. In the context of software engineering, for example, if one looks at the history of execution of all test cases across the project life cycle, one should expect to see—at least for successful projects—that the number of executions that passed outnumber those that failed.

For domains with imbalanced classes, the use of classification accuracy and error rate are not helpful to evaluate the usefulness of a classifier as they cannot distinguish how the individual classes are affected. Rather than averaging the classifier’s performance over all classes, we need to use appropriate metrics that allow us to focus on each class individually. Next, we introduce some metrics that are more appropriate for domains with imbalanced classes.

Precision captures the proportion of positive identifications that are actually correct. In other words, precision captures the probability that the classifier is correct when it labels an instance as positive. It is calculated by dividing the number of true positives (N_{TP}) by the total number of instances predicted as positive ($N_{TP} + N_{FP}$) (see Eq. 11.3).

$$Precision = \frac{N_{TP}}{N_{TP} + N_{FP}} \quad (11.3)$$

Recall captures the proportion of actual positives that are correctly identified. In other words, recall captures the probability that a positive instance will be cor-

rectly labelled as such by the classifier. It is calculated by dividing the number of true positives (N_{TP}) by the total number of positive instances ($N_{TP} + N_{FN}$) (see Eq. 11.4).

$$Recall = \frac{N_{TP}}{N_{TP} + N_{FN}} \quad (11.4)$$

In some contexts, it is desirable to combine precision and recall as a single metric. In those cases, one can use the **F_1 -score**, or simply F_1 . In Eq. 11.5, the parameter “2” comes from a weighting factor equal to “1”, meaning that the user does not know which of the two, precision or recall, is more important for the given context. However, it is possible to change the weighting factor to adjust the relative importance of the two criteria and give more weight to either precision or recall. The interested reader can refer to Chap. 11 of reference [27] for more details.

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (11.5)$$

Let us now see how precision and recall can reflect different aspects of the classifier’s behaviour through an example. Table 11.3 shows the confusion matrices for the same classifier applied over two different input data sets.

When applied to the input data set #1 the classifier suffers from poor precision. $Precision = 50/130 = 0.38$ means that out of the 130 instances predicted as positive by the classifier, only 50 are indeed positive, the remaining 80 being false positives. $Recall = 50/70 = 0.71$ means that out of the 70 positive instances, 50 are correctly predicted as positive by the classifier. When applied to the input data set #2 the values of precision and recall are now inverted, which means that the classifier suffers from poor recall. $Recall = 250/650 = 0.38$ means that out of the 650 positive instances, only 250 are correctly predicted as positive by the classifier. The achieved precision is computed as $Precision = 250/350 = 0.71$.

If classification accuracy was used as the single metric to evaluate the usefulness of such a classifier when applied to the two input data sets, the results would tell us that the classifier is equally useful when applied to either set of instances: Accuracy for the input data set #1 is computed as $Accuracy = 1900/2000 = 0.95$; and accuracy for the input data set #2 is computed as $Accuracy = 9750/10250 = 0.95$. The use of precision and recall, on the other hand, can help us identify how the classifier’s behaviour was drastically affected by the input data sets in this example.

Table 11.3 The effects of an imbalanced data set on classifier’s performance metrics

	Predicted class			Predicted class	
Correct class	Positive	Negative	Correct class	Positive	Negative
Positive	50	20	Positive	250	400
Negative	80	1850	Negative	100	9500

11.3.2.2 On the Importance of Precision and Recall for Different Domains

There is no single combination of precision and recall that can be considered ideal for every application domain. The relative importance of each metric is highly influenced by the domain and by the problem at hand. In some domains, precision is more important than recall. For example, suppose you develop a classifier that analyses the modifications performed by a developer and labels the change as “likely fail” or “likely pass”, trying to anticipate the test suite results. If a change is considered “likely fail” the developer receives a warning and is asked to review the change before they can proceed with committing the changes to a central repository.

In such a scenario, one would like to have high precision even if it comes at the cost of low recall: a low recall in this scenario means that not all of the “likely fail” changes are classified as such—while anticipating bugs before committing the code to a central repository would be highly desirable, if a “likely fail” change is not captured and the code is committed, the test suite would eventually report failing test cases; a low precision in this scenario, on the other hand, means that many “likely pass” changes are mistakenly classified as “likely fail”—this wastes developer’s time by asking them to review changes that are actually correct. Besides wasting developer’s time, in the long term, developers would lose trust in the classifier and could start ignoring future warnings. Finally, high precision means that every time a warning is raised there is a high probability that there is, indeed, something wrong that should be fixed before the code is committed to the central repository.

By contrast, recall can be considered more important than precision in other domains. Let us get back to the example of the classifier able to distinguish between patients with or without a rare disease illustrated at the beginning of this section. A false positive (affects precision) happens when a patient who is not affected by the rare disease is misclassified as unhealthy. A false negative (affects recall) happens when a patient who *is* affected by the rare disease is *not* properly diagnosed as such. In this scenario, we would be willing to accept a high proportion of false positives—healthy patients diagnosed as unhealthy would go through additional evaluations that would eventually identify the misclassification. However, we would like the proportion of false negatives to be as small as possible, ideally, zero—unhealthy patients wrongly diagnosed as healthy would not receive the required treatment.

11.3.2.3 Experimentally Evaluating the Classifier’s Performance

An obvious way of evaluating a classifier’s performance is by measuring the proportion of unseen instances that it correctly classifies. Unfortunately, in practice, for most of the domains, the number of possible unseen instances is potentially very large [10]—e.g., all commits made to GitHub, all Java programs ever written, every possible defect report that will be registered in the future. Thus, the usual way by which one evaluates a classifier’s performance is by estimating its predictive accuracy—or any other performance metric—with respect to a sample of data that

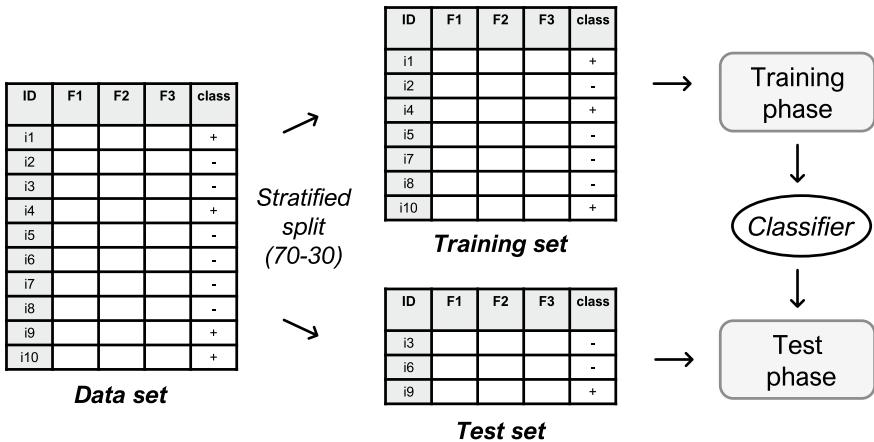


Fig. 11.6 The hold-out strategy

the classifier has never “*seen*”, i.e., not used for training the classifier. Many strategies have been developed for supporting the evaluation of a classifier’s performance. Next, we discuss the most common ones (refer to [10, 27] for more details): The hold-out and the k-fold cross-validation strategies. Finally, we also give some notes on hyperparameter tuning, another concept related to performance evaluation.

The hold-out strategy

The simplest evaluation strategy is to split the set of instances for which the correct label is known into two groups, one for *training* and one for *testing*. The *training set* is used for building the classifier,⁸ and then the classifier is used for predicting the classification of the (unseen) instances from the *testing set*. The classifier cannot use the instances from the testing set to build the model, which is why it is often called a *hold-out* set. This process is depicted in Fig. 11.6. Many approaches can be used for splitting the data into two sets. Commonly, the data is split randomly in a given proportion (e.g., 1:1, 2:1, or 70:30) [10].

Given that the data used for evaluating the classifier is usually limited, a common approach is to repeat the hold-out strategy multiple times (typically 10 or 5), each time with the data being randomly split again into training and testing sets, and report the classifier’s predictive accuracy in terms of average and standard deviation of the results obtained across all repetitions.

An important aspect to take into account when using the hold-out strategy is that the random split of the data could create training and test sets that misrepresent the distribution of classes. In other words, the proportion of instances from the positive and negative classes in the training/test set differs from what one would expect to observe in the real application domain. This is very common when dealing with

⁸ Sometimes it is useful to keep a part of the training data as a separate set, called *validation set*, to tune the hyperparameters and detect overfitting.

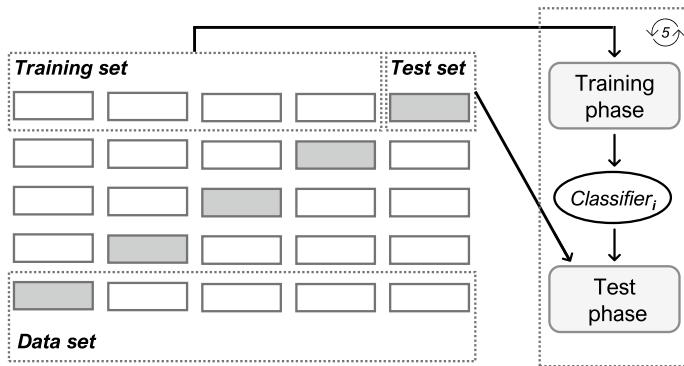


Fig. 11.7 k -fold cross-validation with $k = 5$

imbalanced data sets. In such cases, it might be preferable to use a *stratified* approach to make sure that each set, training or test, provides a good representation of the individual classes.

The code snippet 1 uses Python's scikit-learn library to illustrate the main steps to build and evaluate a Decision Tree model (See Sect. 11.3.1.1) using the hold-out strategy for splitting the known instances into training and testing.

The k -fold cross-validation strategy

In k -fold cross-validation the N instances are split into k equal parts (*folds*)—with k being a small number (typically 5 or 10). Then, a series of k runs are carried out and, in each of the runs, a different fold is retained for *testing* while the remaining $k - 1$ folds are used for *training* the classifier. Such an approach guarantees that a different test set is used in each run. This process is depicted in Fig. 11.7. Because multiple runs are performed, k -fold cross-validation can be costly in terms of both computational resources and time. If the costs are not prohibitive, however, cross-validation should be preferred over the hold-out strategy as it gives the model the chance to be trained and tested multiple times, each time using different training and testing sets.

The overall predictive accuracy of the classifier is reported as the total number of instances correctly classified across all k runs divided by the total number of instances N . When applying k -fold cross-validation, stratified approaches can also be used to guarantee that each *fold* contains a distribution of positive and negative instances that is representative of the entire set of labelled entities. For example, when applying 5-fold cross-validation in a set that contains 10 positive and 90 negative instances, each *fold* should contain 2 positive and 18 negative instances.

code snippet 1: Evaluating a Decision Tree with Python's scikit-learn

A code snippet illustrating the main steps to build and evaluate a Decision Tree model (See Section 3.1.1) using Python's library scikit-learn. In lines 1 to 3 we import the `DecisionTreeClassifier` and other auxiliary functions for training and evaluating the model. The first thing we need to do is to load our data set (line 9). In line 12 we use the helper function `train_test_split` to split our data set into random training and testing subsets (this is the hold-out strategy discussed in Section 3.2.3). In this example, we are using 80% of the instances for training, and 20% for testing. If we wanted to used cross validation instead of the hold-out strategy, we could import `cross_val_score` from `sklearn.model_selection`. In line 15 we create the classifier object and we train our model with `clf.fit` (line 16) by providing the training data (`X_train`) along with its ground truth — correct labels — (`y_train`). After building our Decision Tree model, we can now use it to predict the labels of the unseen instances from the testing set (line 17). Once we have the predictions we can evaluate the performance of our classifier. In the example from Listing 3.2.3 this is done by measuring the model's accuracy. As discussed in Section 3.2.1 different evaluation metrics could be adopted depending on the context and on the characteristics of the data set.

```

1  from sklearn.tree import DecisionTreeClassifier # import the Decision Tree
2  # classifier
3  from sklearn.model_selection import train_test_split # import the helper
4  # function to create a random split into training and test sets
5  from sklearn import metrics # import the metrics module
6
7  # Loading your dataset
8  """Here you should load your data set, which is usually organized into two
9  # different arrays (let us call them <data> and <target>).
10 # <data> is a 2D array with each row representing one sample and each column
11 # representing the features.
12 # <target>, usually a 1D array, contains the class values."""
13 data, target = dummy_load_dataset() # Loading the data set.
14
15 # Splitting the data set
16 X_train, X_test, y_train, y_test = train_test_split(data, target,
17 # test_size=0.2, random_state=0) # splits the data set into training (80%)
18 # and testing (20%)
19
20 # Building the model
21 clf = DecisionTreeClassifier() # the classifier object
22 clf.fit(X_train, y_train) # training the classifier
23 y_pred = clf.predict(X_test) # make the predictions for the instances in the
24 # testing set
25
26 # Evaluation
27 print("Accuracy:",metrics.accuracy_score(y_test, y_pred)) # y_test is the
28 # ground truth (the correct labels); y_pred contains the labels predicted by
29 # a classifier.

```

Hyperparameter Tuning

The classification performance is not only affected by the data used for training, but also the choice of hyperparameters. Identifying the ideal set of hyperparameters to use is a difficult problem, because there is no guarantee that the values working well for one problem will provide the same performance on a different problem. Manually tuning the hyperparameters until one is happy with the model's performance is the first alternative. However, it can be time-consuming, especially if there are many hyperparameters to be considered. Efficient strategies for hyperparameter tuning, thus, become necessary to improve the performance without trying all possible combinations of hyperparameter values. We briefly introduce some “classical” approaches [7], which can be used in combination with either the hold-out and the k -fold cross-validation. For more advanced methods, we refer the reader to works proposing new methods in the context of software analytics [4, 5].

The first approach is *grid search*, which tries a specific number of combinations. The process consists in defining a list of values for each hyperparameter, then run the algorithm with every possible combination of these values. Although the number of combinations is reduced, the problem of combinatorial explosion can appear if the set of hyperparameter is large. A second alternative is *random search*, in which a maximum number of combinations are tried. Random search uses a sampling method to choose each hyperparameter value from its data distribution. This method allows setting a low number of configurations regardless of the number of hyperparameters, but provides less guarantee that a good configuration will be found. Both strategies can be used in combination, the so-called *randomised grid search*. The idea is to run first a random search to locate a promising region of values for each hyperparameter, then use grid search to systematically explore the combination of values in such regions.

11.4 Regression

Regression is another supervised task, whose goal is to fit a model able to predict a continuous variable from one or more inputs. Regression is founded on statistical concepts and can be solved by means of analytical methods, so it is usually included under the umbrella of statistical learning [21]. This might mean changes in notation and terminology, e.g., the inputs are frequently called *predictors* instead of features as we use in classification.⁹ However, regression is the essence of many ML algorithms that, internally, build regression functions to link the features to the class. Even more, classification can be expressed in a regression form where the output represents a probability in the range $[0, 1]$ (recall ANNs in Sect. 11.3.1.2): if the value is ≥ 0.5 , then the positive class is assigned, otherwise the instance belongs to the negative class.

⁹ However, we will continue using the classification terminology here to avoid confusion.

But if we focus on a classical regression problem, we can think of the following situation: could a bank manager predict the savings from the client's salary to give him/her a customised bank planning? To answer this question, we can think of a simple regression model that takes the form of the following equation: $y = \alpha + \beta * x$, where y is the variable to be predicted (income), x is the predictor (salary), β is the slope and α is the intercept (often called *bias* in the ML terminology). If we want to predict y from x , we can collect many points (x, y) and then try to determine the values of α and β (they are usually referred together as the *coefficients*) that provides the optimal adjustment to the points. Once we have such coefficients, we can compute estimations of y (\hat{y}) for any x value just applying the above formula.

However, the proposed example is rather simple and one will argue that an accurate prediction of income might not only depend on the salary, but also on many other factors, such as age, mortgage, etc. As in classification, more input features could help us to make more informed decisions. Indeed, regression from multiple features is also possible [24], we just need to extend the equation to: $y = \alpha + \sum \beta_i * x_i$. Under this formulation, we can interpret that each β_i represents the “weight” to which feature x_i (positively or negatively) contributes to the prediction.

Different methods to estimate the coefficients in a regression problem exist, as well other forms of equations can be used depending on how we assume the features are related between them and with the output [21]. We will briefly present those more relevant to ML and this book in the next subsection. Then, some notes about how we can measure the goodness of fitted models are presented.

11.4.1 Algorithms

The equation presented at the beginning of the section represents the most basic form of regression, known as simple linear regression. By linear, we mean that the model is expressed as a “weighted” sum of terms. These terms could be the features, as in our first equation, or the result of a function over them. Indeed, we could have a distribution of instances for which a curve would fit better than a line. When we use functions like power, multiplication or trigonometric operations over the features, we are defining *basis* functions [21]. By using these nonlinear functions, we search for other (often better) “adjustments”, but the resulting model is still linear. Next, we will describe two types of linear models: the least squares method, based on the basic form of the regression equation; and relevance vector machines, which apply nonlinear basis functions. Other linear models, as well as nonlinear ones, fall outside the scope of this chapter, but the interested reader can find them in [21, 24].

11.4.1.1 The Least Squares Method

The least squares method (LSM) is a simple yet popular way to fit a linear model from a set of training data.¹⁰ LSM seeks to find the coefficients that minimise the squared differences between \hat{y} (the estimations) and y (the actual values): $\sum (y_i - \hat{y}_i)^2$. Such differences are called *residuals* and are the basis of the error measures used to evaluate regression models, as will be explained in Sect. 11.4.2.

Graphically, LSM traces the line that is closer to all the instances in the training set. However, this idea makes a strong assumption: the training data is a good sample of all possible instances. So, the fitted model is highly sensible to variance in the input data and, consequently, it might not generalise well [21]. Another issue is that LSM returns a weight for all the features, even if they have low impact on the prediction. To avoid models with a large number of features, some linear regression methods apply strategies to “shrink” the coefficients, retaining only those that exhibit the strongest effects in the prediction. Ridge regression and Lasso are good examples [24].

11.4.1.2 Relevance Vector Machines

To understand the idea behind relevance vector machines (RVM), we need a brief introduction to support vector machines (SVM) [24]. We will illustrate the concept of SVM for classification, as usually done in textbooks, although SVM has been adapted to regression too [21]. Let us focus on a simplified example with two classes and instances having two features only. If we draw the instances in a 2D space, our classification problem can be viewed as finding a line (the *class boundary*) that splits the plot area in two parts, each one containing instances from one class only. However this is an ideal scenario, often it is not possible to define such a line because instances from the two classes are “overlapped” in the 2D space. SVM becomes useful in this realistic situation, as it is able to find a higher-dimensional space in which the instances can be separated by such kind of line (actually, a hyperplane). Internally, what SVM tries to do is find the hyperplane that maximise the “margins” between the two classes. The hyperplane is defined using *kernels* (basis functions) focused on the instances (the so-called *support vectors*) that are closer to the boundary between classes in the new space defined by the kernel.

RVM is founded on similar ideas, as it builds a linear regression model whose nonlinear kernels are associated with a few number of relevant instances (*relevance vectors*) [9]. However, RVM takes a probabilistic approach. Instead of returning an estimated output for the given instance, it provides a conditional probability which allows modelling uncertainty in the prediction. Combining both characteristics, RVM are able to produce more sparse models compared to SVM, but keeping their generalisation capability. Furthermore, RVM relaxes the mathematical constraints required by SVM, and reduces the number of hyperparameters to be optimised during learning.

¹⁰ In fact, this method is the one behind the implementation of the *LinearRegression* class in scikit-learn: https://scikit-learn.org/stable/modules/linear_model.html (Last accessed March 7, 2022).

11.4.2 Evaluation

While discussing ways for evaluating the performance of classification in Sect. 11.3.2, we saw that *accuracy* is one of the most basic metrics, and it aims at measuring the proportion of correct predictions made by the classifier. The concept of accuracy is not very meaningful for regression and this is justifiable: recall that regression predictions are numeric values. In cases like the one illustrated at the beginning of this section—predicting the savings based on a client’s salary—we should not expect the model to predict the precise values. Instead, we would like to know how close the predictions are to the actual values.

In Sect. 11.4.1.1, we learned that when we perform linear regression we get a line of best fit. As the reader might expect, it is not the case that the regression line will pass through all the instances; instead, they will be scattered around the regression line. The vertical distance between the regression line and an instance is called a *residual* (or *error*). Each instance has one residual. We say the residual is positive if it is above the regression line, negative if it is below the regression line, or zero if it is crossed by the regression line. The residual is thus computed as the difference between the actual and the estimated value with the equation $e = y - \hat{y}$. Residuals are the essence of many metrics that are commonly used to evaluate the performance of regression models. Next, we present three of them: mean squared error, root mean squared error, and mean absolute error.

Mean squared error (MSE) is an absolute measure of the goodness of the fitted model. It is calculated as the mean of the squared differences between the predicted and the actual values (see Eq. 11.6). In essence, *MSE* measures how much the predicted values deviate from the actual ones. When used to compare different regression models, the lower the *MSE*, the better (an *MSE* equal to 0 means a perfect match between predicted and actual values). Notice that, because the errors are squared, larger errors are punished more than smaller errors.

$$MSE = \left(\frac{1}{n} \right) \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (11.6)$$

Root mean squared error (RMSE) for computing the *MSE*, the errors are squared and this means that the unit of the output from *MSE* is not the same as the one from the value being predicted. Because of that, it is common to report the performance of a regression model in terms of the square root of *MSE* as it brings the unit back to the original level (see Eq. 11.7).

$$RMSE = \sqrt{\left(\frac{1}{n} \right) \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (11.7)$$

Mean absolute error (MAE) is similar to *MSE*, but instead of considering the squared error, it uses the absolute value of the error. The value of *MAE* increases

linearly with increases in error. This is because, differently from *MSE*, *MAE* does not give more or less weight to different errors (see Eq. 11.8).

$$MAE = \left(\frac{1}{n} \right) \sum_{i=1}^n |y_i - \hat{y}_i|. \quad (11.8)$$

11.5 Clustering

Many times the data collected come without a “label”, but that does not mean we cannot learn anything from them. We as humans have the capacity of analysing the characteristics of what we see and find similar cases that respond to the same pattern, even if we do not know what that pattern represents. For instance, we can read pieces of texts and identify the common topics underlying them without predefining the list of themes to be found. In ML, this activity is traditionally solved by means of clustering, one of the most common unsupervised learning tasks [27].

Basically, clustering (or cluster analysis) divides the data set of instances into subsets, so that those instances grouped in the same cluster are closely related and, at the same time, are substantially different from the instances assigned to other clusters [21]. In contrast to classification, the clustering process is a unique step, i.e., no training actually happens and a decision model is not built. Given a new instance, we could simply assign it to the closest cluster.

A clustering algorithm uses all the available instances to analyse their distribution, and identify representative examples in each cluster—usually a point that lies in the “centre” of the cluster, the so-called *centroid*. *Similarity* is the key concept in clustering, since the algorithm needs to decide whether two instances are similar enough to be grouped together. For the most simple algorithms, it takes the form of a function that, given two instances, returns their degree of similarity [21]. The number, shape, and composition of each cluster is not known a priori, although some algorithms make assumptions or require to set them as parameters.

Several types of clustering algorithms exist depending on how they measure similarity and how the instances are iteratively grouped until the clusters become stable [2]. In the next section, we will briefly introduce these types and explain two popular algorithms: k-means and agglomerative hierarchical clustering. After that, we address the evaluation of clustering results.

11.5.1 Algorithms

Most of the clustering algorithms usually fall into one of the following categories [2]:

- *Distance-based algorithms*, which use a distance function to measure the similarity of each pair of instances. The cluster creation can happen in two ways:

- Based on partitions, meaning that each instance is assigned to one cluster, the number of partitions being usually specified by the user as a parameter. K-means and k-medoids are popular methods within this category.
- Based on hierarchies, allowing the composition of clusters at different levels of granularity. *Agglomerative* methods follow a bottom-up approach, so they start with one cluster per instance and combine them until all instances belong to the same cluster. The opposite strategy (top-down) is used by *divisive* methods, which use a partition algorithm to iteratively split the previous clusters in smaller ones.
- *Density-based algorithms*, which seek for dense regions with arbitrary shapes to define the clusters, isolating instances that are in areas of low density and might represent noise. DBSCAN is probably the most applied algorithm in this group.
- *Probabilistic algorithms*, which assign each instance a probability to be a member of each cluster. This is a kind of “soft” clustering, since one instance can belong to more than one cluster. The expectation maximisation algorithm is an example of a probabilistic method.

11.5.1.1 The k-Means Algorithm

K-means is a simple clustering algorithm that looks for k clusters, using the average values of the instances in each cluster to iteratively determine the centroids. The steps of the algorithm can be summarised as follows [2]:

1. Choose k instances in the data set as initial centroids. Usually, this process is done at random.
2. For the rest of the instances, assign them to the closest centroid to create each cluster.
3. Recompute the centroid position as the average point in each cluster, i.e., average value of each feature among all instances belonging the cluster.
4. Go to step 2, and repeat until instances do not need to be reassigned.

Three main elements are key for the success of k-means [21, 27]: the choice of k , the centroid initialisation and the definition of the distance between instances. Firstly, k is a user-defined parameter, but for which no general guidelines exist. It is highly dependent on the data distribution and the concepts the clustering is expected to discover. A common approach is to run the algorithm with several values and compare the results. Secondly, the initialisation method clearly influences the convergence of the method. Random initialisation implies that results can change from one execution to another, so alternatives have been proposed to limit its impact. Lastly, several distance functions (Euclidean, Manhattan, etc.) can be considered depending on the type of features. Normalisation of the feature values becomes a necessary preprocessing step here, since distinct value ranges can cause distortions when taking averages. A final aspect is the termination condition (step 4), since instances might continue being

reassigned endlessly. For this reason, some implementations allow stopping the process when the change between iterations is below a threshold (known as tolerance), or allow running the process until a maximum number of iterations is reached.

The simplicity of the algorithm and the shortcomings mentioned above means that k-means might not fit well to many cluster analysis situations. This is especially evident when the data distribution presents areas of different density, shape, and size, because k-means will look for homogeneous clusters. In the next section, the reader will find a clustering algorithm that overcomes some of these limitations.

11.5.1.2 Agglomerative Hierarchical Clustering

If we are not sure about the approximated number of clusters, the hierarchical clustering approach can be of great help to understand the distribution of the instances. This method still relies on distances between instances, but explores several “levels” when grouping them and, therefore, does not assume the optimal number of clusters. The agglomerative process happens as follows [2]:

1. The distance between each pair of instances is computed (known as proximity matrix).
2. Each instance is initially assigned to one cluster.
3. Find the two clusters that are closer and merge them.
4. Update the proximity matrix between clusters.
5. Go to step 3 until the number of clusters is equal to 1.

In this case, the most important factor is how the proximity of two clusters is calculated, since the algorithm is strongly guided towards minimising this “linkage” criterion. Some common approaches are [21]: (1) the minimum distance between any pair of instances belonging to different clusters (*single linkage*); (2) the maximum distance between any pair of instances (*complete*); (3) the average distance between all instances (*group average*); or (4) the variance of the merged clusters (Ward method).

The fact that the number of clusters is iteratively decreased allows inspecting the results at intermediate stages, obtaining solutions from 1 to N clusters (N being the size of the data set). To visually support this analysis, a dendrogram can be depicted showing how the clusters are nested in each step.

11.5.2 Evaluation

The fact that clustering is, by definition, an unsupervised task poses some challenges when it comes to evaluate the quality of the results. As opposed to classification, for which the true class labels are known, we do not have a “ground truth” of clusters to compare the outputs of a clustering method. Evaluation in clustering is known to be

a more subjective process, but some validation criteria can give us some clues about how well clusters describe the underlying data [2].

The clustering validation can be quantitatively performed from three different perspectives [18]: *internal*, if it only assesses the quality based on the produced clusters; *external*, when additional information is available to evaluate the clusters; and *relative*, when we want to compare the results of two or more cluster assignments over a same data set. Both internal and external metrics (a.k.a. validation indices) can be used to compare the results of two clustering algorithms for the relative evaluation approach. Due to space limitations, this section does not include formulations and examples of these metrics as done for classification, so we refer the reader to specialised sources [2, 18] and online material, e.g., the scikit-learn documentation provides a good introduction to most of them.¹¹

Internal validation indices quantify properties of the clusters like their radius and density. However, the results could introduce bias upon comparison if the algorithms under evaluation favour specific cluster shapes [2]. If we want to measure the variation inside a cluster, the **Error sum of squares** computes the sum of the differences between each instance and its cluster centroid. This metric is very useful to set the number of clusters in algorithms requiring such a value as a parameter, or decide the best “level” after executing hierarchical clustering. Other metric is the **Silhouette coefficient**, which does not only consider the proximity of the instances in each cluster, but also the distance to the rest of the clusters.

If the instances can be associated with actual groups (e.g., by a human annotator), then it is possible to measure how similar the clusters are to such groups. External validity indices will use this information to evaluate two desirable properties of any cluster assignment: *homogeneity* and *completeness*. The former indicates that each cluster should be comprised of instances from the same actual group, whereas the latter is achieved when all the instances belonging to an actual group are put together in one cluster. Also, we can simply measure the agreement between the “ground truth” and the cluster assignment for each individual instance, thus following a similar precision-recall strategy as in classification (see Sect. 11.3.2). **Rand index** and **Mutual information** are two metrics based on this idea.

11.6 Deep Learning

So far the reader has been introduced to three common and relatively simple ML tasks: classification, regression, and clustering. This section covers the basics of deep learning (DL), an approach that allows for addressing more complex tasks. We can think of situations in which we, as humans, cannot make decisions solely based on a small set of “flat” variables, but as a consequence of what we observe, how we combine our thoughts and past experiences, and the reactions to our surroundings.

¹¹ <https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation>
(Last accessed March 7, 2022).

A clear example of this is car driving, so the reader will be right in thinking that achieving autonomous driving requires something more than the decision models explained in previous sections.

This is the natural scenario for DL, whose underlying idea is that the computer needs to learn a nested hierarchy of concepts in order to understand the real-world problem [15]. DL is founded on the principles of ANN (see Sect. 11.3.1.2), using multiple layers connected under different architectures to build complex concepts on top of simpler ones. Under this approach, the computer autonomously learns the relevant representations of the data that it needs to fulfil the task, either in a supervised or unsupervised way. Indeed, DL architectures have shown great potential to manipulate many forms of data not easily handled by other ML techniques, such as images and texts, to perform object and speech recognition, respectively.

From this short introduction, the reader can guess that DL is a powerful approach with many potential applications, most of which are yet to come. Also, that understanding the mathematical principles and experimental practices to master DL requires time and specialised sources (we refer the interested reader to [2, 15, 36]). In Sect. 11.3.1.2, the elements and basic structure of an ANN were presented. DL is based on the same concepts (layers comprised of neurons), but it makes use of special functions and defines more complex connection structures. In the next sections, the reader will find the descriptions of the most popular DL architectures, and why they are particularly adequate in some contexts or for some applications. We refer the reader to a recent survey on the application of DL in software engineering to discover the tasks and activities that are currently being automated, and their associated challenges and opportunities [40].

11.6.1 Convolutional Networks

A convolutional neural network (CNN) is a type of ANN specially designed to take inputs with a grid-structure,¹² such as 2D images [15]. In an image, pixels are not independent elements, since adjacent pixels share properties (e.g., colour) and are part of more abstract elements (lines, shapes, and objects). Inputs with this type of spatial dependencies and compositions are suitable for CNNs, since they exploit the information from regions instead of individual units (the pixels).

CNNs owe their name to *convolution*, a mathematical operation that makes a special type of matrix multiplication (dot product) between two arguments: a multidimensional array (e.g., the image) and a *kernel*. The kernel is also a multidimensional array, but it contains the parameters that should be adapted to make proper transformations depending on the target problem. In the DL nomenclature, both arrays are called *tensors*.¹³ The output of the operation is another array, referred as *feature*

¹² But not limited to it, as sound (1D) and video (3D) could be processed as well.

¹³ This is why the Google platform for ML/DL is called TensorFlow: <https://www.tensorflow.org/> (Last accessed March 7, 2022).

map. In our example, this feature map can be a pixel-based representation of a small entity identified in the image, e.g., the eyes of a person, which will be propagated to the next layers as input (until the whole person is recognised in the image).

If the network applies the convolution operation in at least one layer, we can say we have built a CNN. However, it usually appears in multiple layers and is combined with other operations. A CNN often presents three types of layers [1]: convolution, already introduced, acts as a filter that slides across spatial regions (e.g., a group of NxN pixels in the image); pooling, which reduces data dimensionality and helps make the process independent of the data conditions (e.g., the location of the object in the image); and a rectified linear activation function (ReLU in short), instead of the sigmoid function frequently used by ANNs.

The success of a CNN depends on the design of the network structure, since layers are not usually fully connected. Instead, they are specialised in local regions analysed in the previous layer and the spatial relationships traced through them.

11.6.2 Recurrent Networks

A recurrent neural network (RNN) is a type of DL architecture specialised in processing sequences of data [15]. We can think of two types of inputs: a time series in which values are produced at some frequency, and text data, viewed as sequences of words. Both kinds of data have characteristics that, if we opt for traditional ML algorithms, will be lost. We can omit timestamp information in a time series and just learn with all the instances collected in an amount of time, but often a point in a series is related to the previous one.¹⁴ As for text, we could use a BoW representation (see Sect. 11.2.2.3), but we lose syntactic structures and understanding the semantics becomes difficult. Another issue with sequential data that DL overcomes is the fact that the example sequences can be large and of different lengths, but an ANN is designed to receive a fixed (ideally small) number of values in the input layer.

To handle sequences more efficiently, a RNN uses a variable number of layers, each one having a unique input. In other words, a one-to-one correspondence between the layers and the positions in the sequence exists [1]. The layers receive a sequence of input instances with one or more features and produce one or more outputs. Each layer has the same parameters, so the global architecture seems to be repeated in time and thus termed as “recurrent”. If convolution was the core concept in CNNs, *unfolding* is the keyword for understanding RNNs [15]. Unfolding a function that involves a temporal variable, i.e., the function takes the previous value of the variable as a parameter to calculate the next one, consists of repeatedly applying the function to all the previous results in the variable sequence, thus removing the recursion. RNNs apply this concept to compute the state of the hidden units in each time instant, taking as input all (or a subset of) the previous values in the sequence and

¹⁴ Online (a.k.a. stream) learning is more suitable for this type of situation, but still presents limitations compared to DL such as the lack of memory.

the history of past states. Graphically, it can be seen as an operation by which a recurrent graph, comprised of an input node and its state (a node that is recursively updated), is transformed into a chain (unfolded graph) of input and state nodes that are traversed by the repeated application of one function. This network structure has two main advantages: (1) the learned model does not depend on the input length, since it learns from the state transitions, and (2) only one transition function with the same parameters is needed.

The long short-term memory (LSTM) model is a popular RNN with good results in many applications [15]. LSTM adapts the rules guiding how the states are propagated through the hidden units, allowing to keep or forget parts of the information inferred from the sequence. This process is controlled by special filters that take the form of *gates*, which perform addition, multiplication, and nonlinear operations over the data, to influence the hidden units [36]. More specifically, the LSTM architecture can be viewed as a network of *memory blocks*, each one comprising of one or more self-connected *cells* [16]. Each block contains three gates: the *input* gate, the *output* gate, and the *forget* gate. These gates controls how the information is managed by the cells, as a kind of “write”, “read”, and “reset” operations over the cells, respectively. If the input gate is “open” (i.e., activated), new inputs are allowed to enter the memory block. Otherwise, it keeps the last saved input. Such input is made available to the network when the output gate is opened, and is not forgotten until the forget gate is also opened. This mechanism ensures the relevant information is not lost as the time goes by, a problem that appears in other recurrent architectures.

11.6.3 Autoencoders

Autoencoders are feed-forward ANNs with an arbitrary number of layers, like those presented in Sect. 11.3.1.2, but with a special characteristic: the output layer produces the same values as those fed into the input layer [36]. This could be seen as a useless process, but actually represents a form of unsupervised data preprocessing able to find alternative representations of the data. When building an autoencoder, the interest does not lie in the last layer, but in the output of one hidden layer. If we know that the output values are equivalent to the input ones, then the transformations done by the hidden layer provide an intermediate representation of the data using other features. Actually, the result is similar to what a feature selection method like PCA (mentioned in Sect. 11.2.2.2) returns: a new set of features which are obtained as combinations of the original ones.

Depending on how the autoencoder is defined, the new representation of the data will take different forms. Assuming the simplest scenario with only one hidden layer, the number of hidden units in the middle layer should be distinct from the number of inputs. Otherwise, it will simply learn an identity function that copies the inputs to the outputs. If the hidden layer contains less neurons than the number of inputs, we are defining a *simple autoencoder* that will produce a representation with fewer elements. The opposite case is possible too, and is referred to as *sparse autoencoder*.

The result is, therefore, a larger representation of the data, in which the original data features are “diluted” and, in some scenarios, can make learning easier [36]. Other autoencoders exist, e.g., denoising autoencoders to remove injected noise in the inputs, and even their middle layers can be combined to build more complex preprocessing methods.

11.6.4 Other Deep Learning Models

The field of DL is evolving fast, and so does its application to software engineering [40]. In this last section, we briefly introduce some other DL architectures. Interested readers can deepen in these architectures with dedicated reviews [17, 31, 32].

- **Attention models.** These models are inspired by a cognitive process we as humans do. When we are learning, we pay more attention to some parts of what we read, or tend to focus on some objects in the scene we are observing. This idea is translated to deep learning by means of a specific mechanism added to the network architecture that allows it to assign different “levels of importance” (attention weights) to parts of the inputs. Attention models were conceived for translation tasks with long input sequences as input, e.g., translators from language A to language B. The attention mechanism improves the prediction of the output sequence (sentence in language B), mapping each value (word in language B) with specific parts of the input sequence (one or more words in language A). In short, a “good translation” is not a word-to-word conversion process, it requires some context. Attention mechanisms are key components of **transformers** [39], a DL architecture for sequence-to-sequence processing. Transformers follow an encoder-decoder structure, each part containing several attention mechanisms and a feed-forward neural network. BERT and GPT are transformers created by Google and OpenAI respectively, that have achieved notorious relevance in NLP tasks.
- **Generative adversarial networks (GANs).** In contrast to attention models, GANs are not conceived to transform data, but to generate data. The architecture is actually comprised of two models, a *generator* and a *discriminator*. The former “learns” from data samples in order to produce new examples as similar as possible to the real ones. The discriminator has to learn how real samples can be distinguished from synthetic ones. The training process can be viewed as a competition since each element acts as the “adversary” of the other. In other words, the generator has to produce sophisticated examples that makes it difficult for the discriminator to discern from the original data. As the discriminators increase their performance in the classification task, the generator has to improve its generation process. GANs are widely applied in image and video processing, but are also used in NLP to process text or in machine learning to augment data.

11.7 Conclusions

We conclude this chapter with a list of “takeaways” for the reader, as a way to highlight relevant ideas and practicalities surrounding the application of ML:

1. The learning process actually starts with a clear identification of the problem, collecting the evidence (in form of variables) that could help us to solve it.
2. Data preprocessing is as important as training the model, the expression “garbage in, garbage out” strongly applies in the ML case.
3. Depending on how we model the problem, we will define a different learning task (classification, regression, clustering, etc).
4. Each learning task makes its own assumptions and has its own training and evaluation methods.
5. When training, problems like overfitting and imbalance can appear. One should follow good experimental practices to deal with them.
6. Each algorithm builds a different type of decision model, some of which might be easier to interpret. We should not look for perfect accuracy only.
7. The experience says that relatively simple algorithms work well in many situations. We should not adopt complex techniques just because they are trendy.

This chapter covers just the essential part of the vast and fascinating ML world. We invite the reader to dig into the presented techniques and explore additional ones guided by the provided references. There may be more than one technique to solve a problem, just as new problems will emerge that require novel techniques. Our final thought for the reader is: ML is a fast-developing area and the best is yet to come.

References

1. C.C. Aggarwal. *Neural Networks and Deep Learning: A Textbook* (Springer, 2018)
2. C.C. Aggarwal, C.K. Reedy, (eds.), *Data Clustering: Algorithms and Applications*, 1st edn. (CRC Press, 2013)
3. C.C. Aggarwal, C. Zhai, A survey of text classification algorithms, in *Mining Text Data* (Springer, 2012), pp. 163–222
4. A. Agrawal, W. Fu, D. Chen, X. Shen, T. Menzies, How to “dodge” complex software analytics. *IEEE Trans. Softw. Eng.* **47**(10), 2182–2194 (2021)
5. A. Agrawal, X. Yang, R. Agrawal, R. Yedida, X. Shen, T. Menzies, Simpler hyperparameter optimization for software analytics: why, how, when. *IEEE Trans. Softw. Eng.* **1** (2021)
6. U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2Vec: learning distributed representations of code. *Proc. ACM Program. Lang.* **3**(POPL), 40:1–40:29 (2019)
7. J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* **13**(2) (2012)
8. C. Bird, T. Menzies, T. Zimmermann, (eds.), *The Art and Science of Analyzing Software Data*, 1st edn. (Morgan Kaufmann, 2015)
9. C.M. Bishop, *Pattern Recognition and Machine Learning*, 1st edn. (Springer, 2006)
10. M. Bramer, *Principles of Data Mining*, 3rd edn. (Springer, 2016)
11. O. Chapelle, B. Schölkopf, A. Zien, *Semi-Supervised Learning* (MIT Press, 2006)

12. V. Efstatiou, C. Chatzilena, D. Spinellis, Word embeddings for the software engineering domain, in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)* (ACM, 2018), pp. 38–41
13. J. Eisenstein, (ed.), *Introduction to Natural Language Processing*, 1st edn. (MIT Press, 2019)
14. S. García, J. Luengo, F. Herrera, *Data Preprocessing in Data Mining*, 1st edn. (Springer, 2015)
15. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016)
16. A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*, 1st edn. (Springer, 2012)
17. J. Gui, Z. Sun, Y. Wen, D. Tao, J. Ye, A review on generative adversarial networks: algorithms, theory, and applications. *IEEE Trans. Knowl. Data Eng.* **1** (2021)
18. M. Halkidi, Y. Batistakis, M. Vazirgiannis, On clustering validation techniques. *J. Intell. Inf. Syst.* **17**, 107–145 (2001)
19. J. Han, M. Kamber, J. Pei, *Data Mining: Concepts and Techniques*, 3rd edn. (Morgan Kaufmann, 2011)
20. E. Hancer, B. Xue, M. Zhang, A survey on feature selection approaches for clustering. *Artif. Intell. Rev.* **53**, 4519–4545 (2020)
21. T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd edn. (Springer, 2009)
22. H. He, E.A. Garcia, Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.* **21**(9), 1263–1284 (2009)
23. H. He, Y. Ma, (eds.), *Imbalanced Learning: Foundations, Algorithms, and Applications* (Wiley-IEEE Press, 2013)
24. G. James, D. Witten, T. Hastie, R. Tibshirani, *An Introduction to Statistical Learning with applications in R*, 1st edn. (Springer, 2017)
25. J. Jiarpakdee, C. Tantithamthavorn, C. Treude, The impact of automated feature selection techniques on the interpretation of defect models. *Empir. Softw. Eng.* **25**, 3590–3638 (2020)
26. A. Kao, S.R. Poteet, *Natural Language Processing and Text Mining*, 1st edn. (Springer, 2007)
27. M. Kubat, *An Introduction to Machine Learning*, 2nd edn. (Springer, 2017)
28. H. Liu, F. Hussain, C.L. Tan, M. Dash, Discretization: an enabling technique. *Data Min. Knowl. Disc.* **6**, 393–423 (2002)
29. T. Menzies, L. Williams, T. Zimmermann, (eds.), *Perspectives on Data Science for Software Engineering*, 1st edn. (Morgan Kaufmann, 2016)
30. T. Mitchell, *Machine Learning*, 1st edn. (McGraw Hill, 1997)
31. Z. Niu, G. Zhong, H. Yu, A review on the attention mechanism of deep learning. *Neurocomputing* **452**, 48–62 (2021)
32. S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M.P. Reyes, M.-L. Shyu, S.-C. Chen, S.S. Iyengar, A survey on deep learning: algorithms, techniques, and applications. *ACM Comput. Surv.* **51**(5) (2018)
33. D. Pyle, *Data Preparation for Data Mining*, 1st edn. (Morgan Kaufmann, 1999)
34. S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th edn. (Pearson, 2020)
35. B. Settles, *Active Learning* (Morgan & Claypool Publishers, 2012)
36. S. Skansi, *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence* (Springer, 2018)
37. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction*, 2nd edn. (MIT Press, 2018)
38. J. Tang, S. Aleyani, H. Liu, Data classification: algorithms and applications, in *Chapter Feature Selection for Classification: A Review*, 1st edn. (Chapman and Hall/CRC, 2014)
39. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)* (Curran Associates Inc, Red Hook, 2017), pp. 6000–6010
40. Y. Yang, X. Xia, D. Lo, J. Grundy, A survey on deep learning for software engineering. *ACM Comput. Surv.* (2021)