Università degli studi di Napoli "Federico II"

Dottorato in Scienze Computazionali e Informatiche

Ciclo XXV

Improving Software Maintenance using Unsupervised Machine Learning techniques

a dissertation presented

by

Valerio Maggio

to

The Department of Mathematics and Applications

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Software Engineering

Naples, Italy

March 2013

Thesis advisors: Sergio Di Martino and Anna Corazza Valerio Maggio

Improving Software Maintenance using Unsupervised Machine Learning techniques

Abstract

Software maintenance is an essential step in the evolution of software systems and represents one of the most expensive, time consuming, and challenging phases of the whole development process. In particular, the cost and the effort necessary for both the maintenance and the evolution operations (e.g., corrective, adaptive, etc.) are mainly related to the effort necessary to comprehend the system and its source code. As a consequence many *reverse engineering* tools and solutions have been proposed to support the maintainers in their activities.

An important resource for maintainers is represented by the architectural information of the system. However such information is usually not documented, or the documentation is outdated. Therefore, the existing code remains the most updated source of information to exploit in order to automatically retrieve and reconstruct the architecture of a system.

Many research efforts are being devoted to support this task, in order to define solutions that are able to *re-modularise* a given software application. The main purpose of re-modularisation techniques is to automatically partition the system into meaningful subsystems, in order to locate and group together software components that are in some way related, e.g., they implement the same functionalities.

A number of these approaches generally attempt to discover these groups (or clusters) by exploiting the lexical information provided in the source code, such as terms in comments, as well as names of identifiers (e.g., variable, methods and classes).

Nevertheless, the source code lexicon has some specific peculiarities that make it conceptually different from a typical textual resource: identifiers are often created by con-

catenating multiple words (e.g. `getAttribute`, `MINHEIGHT`), which may be additionally shortened (e.g., `getAttr`, `MINHGT`) to avoid long names. As a consequence, tools and techniques that analyse the source code lexicon must integrate algorithms to *normalise* its vocabulary.

Another well known and largely investigated issue in software maintenance is *clone detection*: it is focused on the identification of source code duplications. Software clones might affect the reliability and the maintainability of large software systems. For example, errors affecting a fragment of code must be fixed in everyone of its possible duplications.

Clones are usually not documented, and their identification is usually complicated since programmers adapt software copies by applying multiple modifications (e.g., adding new statements and renaming variables). Therefore, automatic and reliable approaches are required in order to tackle this problem.

In this thesis we proposed new Machine Learning (ML) based approaches that mine the relevant information directly from the source code to cope with the three introduced issues, namely the *software re-modularisation*, the *source code vocabulary normalisation*, and the *clone detection*. In particular, proposed contributions leverages the benefits of ML algorithms, which have been properly tailored and customised in order to make them suitable for the considered domain.

All the presented approaches have been extensively assessed with empirical evaluations conducted on large software systems, and results have been compared with other related techniques, whenever possible. Achieved results outperform the state-of-the-art solutions for all the three considered problems, thus confirming the benefits derived from the definition and the application of ML algorithms to maintenance tasks.

Contents

Listing of figures

List of Tables

List of Algorithms

To V., my deepest love and my reason to smile (without a reason why)

Acknowledgments

First of all, I would like to express my gratitude to my tutors, Dr. Sergio Di Martino and Dr. Anna Corazza, for their guidance, advice, encouragement and support.
I would like to thank all of my friends and other members of the KnomeLab, who have helped me in one way or another along the way.
I also thank my family for its constant support and inspiration over these years.
Finally I would like to thank everybody that has been on my side and helped me whenever I needed, and whenever I did not.

If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts he shall end in certainties.

Sir Francis Bacon

I Introduction

The development and the *maintenance* of large software systems in a changing environment is one of the major challenge for software engineering [193]. This issue has been eloquently discussed in Brooks' classical paper *No Silver Bullet* [30], where the author affirms that developing large software involves essential difficulties related to its *complexity*, *conformity*, *invisibility* and *changeability*.

In fact, changes are inherent to software [92]: they may be necessary to satisfy requests for performance improvements, or to deal with errors discovered in the system [79, 122].

In his famous *Laws of Software Evolution* [120], Lehman declares that software applications must necessarily evolve and grow to remain satisfactory (Lehman's first law), but maintenance operations are required in order to reduce the complexity of the systems (Lehman's second law).

However, *software maintenance* arguably represents one of the most expensive and time consuming software activities. Companies spend more time (and money)

to maintain and evolve existing software than on developing new one [92].

A number of studies have been undertaken to investigate the costs of software maintenance [60, 62, 88, 143, 151], and many of their findings declare these could account up to the 85-90% of the total software costs [79].

The cost and the effort necessary for maintenance activities (e.g., corrective or adaptive), are mainly related to the effort necessary to comprehend the system and its source code [135]. In fact, it is argued that up to the 60% of the total maintenance effort is spent on such activity [6, 108].

The main reasons are: (I) some pieces of knowledge on the specific domain covered by the application are not explicitly stated in the documentation [108]; or (II) the documentation is missing or not up-to-date.

Conversely, the source code represents a valuable source of information for program comprehension [126]. On the one hand it intrinsically provides the most up-to-date information about the system; on the other hand, its lexicon (i.e., identifiers and comments) embeds the domain knowledge provided by developers, thus bridging the gap with the lack of a reliable documentation.

As a consequence, any solution that can improve maintenance productivity is bound to have a dramatic impact on software costs [92]. To this aim, in the last decades several approaches have been proposed in the literature to support maintainers, giving rise to the establishment of the *software maintenance (and evolution)* field as an accepted research area in the software engineering community.

I.A Thesis Motivations

Understanding the software is the fundamental and necessary activity that precedes any type of change to the system. The comprehension process requires a great deal of the total time spent on analysing and applying changes to the system to maintain. This is mainly caused by the complexity of the system, the lack of sufficient domain knowledge, and it is usually aggravated by an incorrect, outdated or non-existent documentation [79].

To reduce this effort, one possible solution should be to gather relevant information about the system from the source code (e.g., the design or the architectural model).

Reverse engineering techniques provides effective solutions to face this kind of issues, aiding the comprehension of the system and easing the implementation of the desired modifications [79].

Reverse engineering approaches are distinguished according to the specific kind of information they exploit during the analysis process. However, such techniques share the same goal, namely to support maintenance activities by allowing a large and usually complex system to be comprehended in terms of *what it does*, *how it works* and its *architectural representation* [79].

In fact, an important resource for software maintainers is represented by the *architectural information* [73]. Software architectures provide *models* and *views* representing the relationships among the different software components, according to a specific set of concerns [57, 155].

Several approaches have been proposed in the literature to support this task, known as *software architecture recovery* [57]. The greater part of the approaches for architecture recovery [103, 132] aim at partitioning the system into meaningful subsystems by automatically locate and group together software components that are in some way related, e.g., they implement the same functionalities.

This specific task is usually referred as *software re-modularisation* (or *software clustering*). In particular, a re-modularised version of the system produces an architectural model that is easier to comprehend and to maintain [11, 184].

A number of these approaches generally attempt to discover these groups (or clusters) by analysing structural dependencies between software artefacts [11, 28, 141, 186]. However, if the analysis is based on the sole structural aspect, a key source of information about the analysed software system may be lost, i.e., the domain knowledge that developers embed in the source code lexicon. In fact, developers usually communicate their intent and their domain knowledge by means of significant terms in comments, as well as names of identifiers (e.g., variable, methods and classes). However, the source code vocabulary has some specific peculiarities that make it different from a typical plain text: identifiers are often created by concatenating multiple words, separated by some special characters (e.g. `to_string`), by capitalising their first letters, or by any special convention (e.g. `IRDocument`, `MAXSIZE`) [61, 118]. In addition, a common programming habit is to shorten words of compound identifiers using abbreviations and/or acronyms

to avoid long names. As a result, tools and techniques that analyse the lexical information provided by the source code must integrate algorithms and solutions that are able to *normalise* the source code [113, 115, 117, 118], namely splitting compound identifiers and expand possible occurring abbreviations. To this aim, in recent years many research efforts are being devoted to deal with the structure of the identifiers aiming at improving the performance of the so-called lexical-based software analysis/maintenance tools [61, 66, 80, 85, 113, 115, 117, 126].

Another issue that characterise large software systems is the problem of *redundancies*. Programs are often polluted by redundant code, namely similar code structures are often replicated in many places of the program.

Duplicated source code is a phenomenon that occurs frequently in large software systems [20]. Reasons why programmers duplicate code are manifold. The most well-known is a common bad programming practice, the so-called *copy and paste* [159], that gives rise to *software clones*, or simply *clones*.

Code clones may increase the difficulties in analysing and applying changes and thus they significantly contribute to increase maintenance costs [79, 92].

To this aim, this problem is a well-known and largely investigated issue in software maintenance, usually referred as the problem of *clone detection*.

The clone detection task is mainly focused on the analysis and the identification of code duplications, aiming at documenting the different redundancies detected in a system. In fact, maintenance problems aggravate during the evolution: the different duplications are merged with existing code and it becomes unclear which part of the code relate to which source of change, thus leaving duplications mostly undocumented.

However, programmers usually adapt the copies to the new context by applying multiple modifications such as adding new statements, renaming variables, and so forth. This aspect further complicates the detection process: on the one hand, it could likely happen that some clones are not detect; on the other hand this affects the *reliability* of the system under investigation. As a consequence, maintainers need to rely on automatic tools and techniques that are able to support them in their detection duties, especially in case of large and complex systems.

The work presented in this thesis provides research contributions to all the three maintenance issues previously described, namely *software re-modularisation*,

source code vocabulary normalisation, and *clone detection*.

In particular, the presented contributions combine different methods gathered from the *machine learning* (and information retrieval) field to automatically mine information from the source code.

Machine learning (ML) algorithms have proven to be of great practical value in a variety of application domains, providing flexible solutions able to analyse large data set with an affordable computational efficiency.

Not surprisingly, the field of software engineering turns out to be a fertile ground where many software development and maintenance tasks could be formulated as learning problems and handled in terms of learning algorithms [192, 193].

As a matter of fact, machine learning techniques have been applied by researches to support a variety of software engineering activities, such as prediction of software development effort, program transformation, or reuse library construction [131], producing some good results [3, 24, 82, 123, 125, 192, 193].

In the case of source code analysis, the aforementioned flexibility of ML approaches allows to exploit the different kind of information provided by the source code. In particular, the proposed approaches apply the so-called *Kernel Methods* [87, 146] to combine the *structural/syntactical* information of the source code with the *lexical* one, based on the assumption that the different kind of information could produce several significance to the considered maintenance issues. However, a great effort is required to adapt and modify these approaches in order to make them suitable to analyse the source code.

ML techniques are founded on a learning method that is usually referred as *inductive learning* [142], since conclusions are derived generalising from "observed" examples, namely the analysed data. These examples may be "labeled" or "unlabeled", giving rise to the so-called *supervised* and *unsupervised* learning strategies respectively.

Nevertheless, the availability of a reliable set of labeled data is usually difficult or impossible in some extreme cases. As a consequence, unsupervised machine learning techniques are typically preferred over supervised ones [131], especially in case of software engineering problems. In particular, some of the advantages of unsupervised learning techniques are [59, 131]:

- There is no cost of collecting and labelling data.
- Unsupervised techniques may be used to identify characteristics of the samples which are useful for differentiating between them.
- Unsupervised techniques may be used for exploring the data and analysing its structure.

On the other hand, a trade-off is imposed on their effectiveness as they solely rely on the quality of the analysed data.

I.B Outline of the Thesis

This section describes the contents of the thesis, highlighting its original contributions.

The thesis is divided into two parts. The first part outlines background concepts related to both software maintenance and machine learning.

Chapter 1 provides a general description of software maintenance and evolution research themes, specifically focused on the description of the three maintenance issues related to the research contributions presented in this thesis, namely the *software re-modularisation* (Section 1.3), the *source code (vocabulary) normalisation* (Section 1.2), and the *clone detection* (Section 1.4). Each of these Sections, provides a detailed specification of the analysed problem, together with an extensive description of corresponding related work.

Furthermore, Chapter 2 introduces the notation and basic concepts of Machine Learning used throughout the remaining chapters. Section 2.1 gives basic definitions about the structures used in the following chapters. Section 2.2 provides a general description of (machine) learning problems and definitions, outlining the differences between *supervised* and *un-supervised* learning problems. Sections 2.3 give an overview of existing *clustering* approaches and of *probabilistic models* respectively. Section 2.4 introduces the *kernel functions*, the convolution kernel framework and provides a description of existing approaches for building kernel functions. Section 2.4.2 gives an overview of kernel functions for structured data as trees and graphs.

The second part of the thesis is devoted to the presentation of the original contributions.

I.c Origin of the Chapters

Chapter 3 is based on the articles [43, 45] (Section 3.1) and contains some unpublished work (Section 3.2).

The material presented in Chapter 4 is based on the article [4].

Chapter 5 is based on articles [44, 46] (Sections 5.1 and 5.3) and contains some unpublished work (Section 5.2).

Part A

Background and Related Work

Know how to solve every problem that has ever been solved.

Richard Feynman

1

Software Maintenance and Evolution

The classical view on software engineering, founded upon the well-known *waterfall life-cycle* process for software development [160], delegates to *maintenance* activities only bug fixing and minor adjustments operations [92, 139]. This view has long governed the industrial practice in software development and it even became a part of the IEEE 1219 *Standard for Software Maintenance* [69], which defines software maintenance as "the modification of a software product *after delivery* to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [139].

However, today we know that what actually happens to software after its first delivery its much more complicated than simply fixing errors, and the different changes to the system may involve corrective as well as adaptive modifications.

For example, a system may need functional enhancements to take advantages of some new technological changes in the production environment (e.g., new platforms or architectures), or it may require several modifications to fix errors or to improve performances.

In more details, the ISO/IEC standard on software maintenance proposed a categorisation of maintenance activities based on four different classes [139]:

- *Perfective Maintenance* is any modification of a software product after delivery to improve performance or maintainability.
- *Corrective Maintenance* is the reactive modification of a software product performed after delivery to correct discovered faults.
- *Adaptive Maintenance* is the modification of a software product performed after the delivery to keep a computer program usable in changed or changing environment.
- *Preventive Maintenance* refers to software modifications performed for the purpose of preventing problems before they occur.

For the sake of completeness, it is worth mentioning that the above classification has been further extended by Chapin et al. [39] based on objective evidence of maintainers' activities [139].

As a result, the new term *software evolution* has been coined to better reflect this wide range of post-release processes on software systems [92].

The term was originally coined by Manny Lehman, who in the late seventies formulated his, now famous, *Laws of Software (or Program) Evolution* [119, 120], emphasising the importance of maintenance activities in the software life-cycle, and its great impact on the quality and the complexity of the systems.

In particular, the first two Lehman's laws (out of eight) declare that:

> *Law of Continuing Change*: systems must be continually adapted or they become progressively less satisfactory to use.
>
> (*First Law of Software Evolution* - Lehman, 1980 [120])

> *Law of Increasing Complexity*: as a system evolves, its complexity increases unless work is done to maintain or reduce it.
>
> (*Second Law of Software Evolution* - Lehman, 1980 [120])

Nevertheless, the term *software evolution* gained its widespread acceptance only in the nineties [139], and the research on software evolution started to become popular [15, 150].

The overall software evolution process is depicted in Figure 1.1 by means of an UML Activity diagram. The picture represents the different activities included in the process to satisfy a new *Change Request* to the system. In particular, some of these activities, such as the *Change Analysis* (Figure 1.2) and the *Change Implementation* (Figure 1.3), are further specified to emphasise that they are far from being trivial tasks.

Nowadays, software evolution has become a very active and well-respected field of research in software engineering [139]. In fact, the 2004 ACM/IEEE Software Engineering Curriculum Guidelines list software evolution as one of the ten key areas of software engineering education [139].

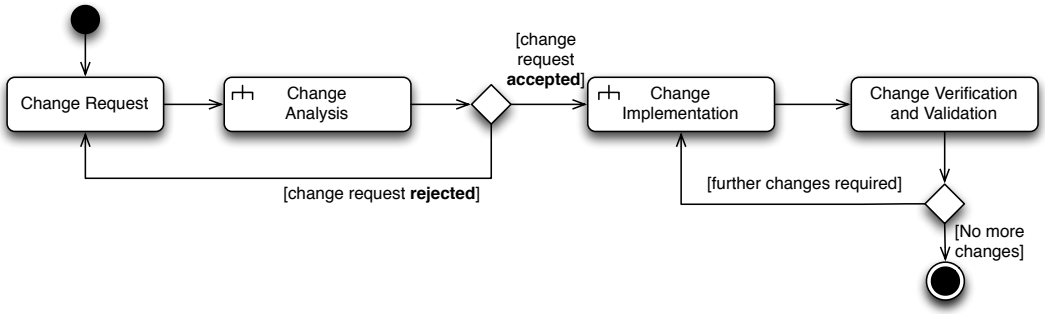

Figure 1.1: UML Activity Diagram of the staged process model for evolution (adapted from [190])

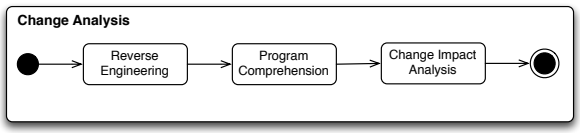

Figure 1.2: *Change Analysis* Activity Diagram

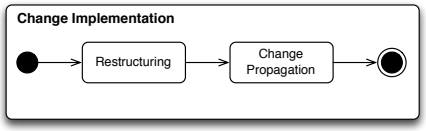

Figure 1.3: *Change Implementation* Activity Diagram

1.1 Issues in Maintaining Large Software Systems

Software maintenance is about change. It may be a small change to fix a bug or enhance software requirements to better satisfy users [92]. However, the size and complexity of programs make applying changes a very hard activity. In particular, the complexity of understanding and maintaining a program is proportional to its size and complexity [92]. In his famous paper, "No Silver Bullet: Essence and Accidents of Software Engineering" [30], Frederick Brooks argues that programming is inherently complex. In particular, he affirms that whether we adopt a machine language or a high-level programming language, we are not able to simplify a program below a certain threshold that he calls an *essential program complexity*. Brooks continues declaring that the two main factors that determine the program complexity threshold are: (I) the complexity of a problem and its solution at the conceptual level, and (II) the fact that not only do programs express problem solution, but also must address a range of issues related to solving a problem by a computer.

These two factors cannot be clearly separated one from the other in program components, which constrains maintainers' effort in using the "conventional decomposition" strategy (i.e., *divide and conquer*) to handle the software complexity issue.

In other words, what makes changes hard is the ripple effect of modifications [92], coupled with the difficulties in correctly identifying what are (and will be) all the different part of the system to modify. As a matter of fact, a single change to the system usually affects multiple software components. For instance, a single change in user requirements may likely affect the design and the component architecture. Furthermore, any change that affects modules interfaces or global system properties may unpredictably impact many other different components [92].

Therefore, any solution that could alleviate these problems is inevitably bound to improve the effectiveness of maintenance activities.

To this aim, in recent years many research efforts have been devoted to the definition of tools and techniques aimed at supporting maintainers in their duties. This effort fed the growing interest of the research community in these topics, and has provided successful solutions for the industry. In fact, many adopt automated

maintenance tools to improve the quality of their software development process.

1.1.1 Reverse Engineering Techniques

An important theme within the research domain of software maintenance and evolution is the *reverse engineering* [40].

Reverse engineering approaches build higher-level, more abstract *software models* in an automatic fashion, gathering information from the source code or any other available document [23, 36, 40, 92, 168].

In particular, the reverse engineering is the process of analysing a subject system to [40, 131]:

- Identify the system's components and their interrelationships.
- Create a representation of the system in another form, or at a higher level of abstraction.

As a matter of fact, reverse engineering solutions are usually employed to recover lost information, to improve and/or provide new documentation, to extract reusable components or to reduce the overall maintenance effort [79]. A summary description of these goals and their corresponding benefits for the maintenance are reported in Table 1.1. Moreover, within the overall software evolution process (Figure 1.1), the application of reverse engineering solutions immediately precedes program comprehension (or program understanding) [63] tasks (Figure 1.2). In fact, program comprehension approaches are specifically devoted to make sense of the different information produced by reverse engineering techniques [139].

Recovering lost information: The documentation of requirements and design of large systems is usually not up-to-date or, in extreme cases, may even not exist. As a consequence, the program code is the only reliable source of information about the system. To this aim, several reverse engineering tools have been developed to exploit the precious information embedded in the source code by developers such as their domain knowledge.

Re-documentation: The re-documentation of a system is the recreation of a semantically equivalent representation within the same relative abstraction level [79]. The main goals of this process are to create alternative views of the system to enhance the understanding, to improve the current available documentation, and to generate the documentation for a newly modified program [79].

Extracting reusable components: The source code of large software systems is polluted by a lot of redundancies and duplications. Indeed programmers usually duplicate the code to overcome some possible limitations of the programming language or the pressures of an upcoming deadline.
Duplications inevitably increase the size and the complexity of the code, as well as the total effort necessary to comprehend and maintain it.
To this aim, several approaches have been defined to automatically identify the so-called "cloned code" [21, 159]. In particular, these techniques aim at documenting all the discovered redundancies and at refactoring possible reusable components, improving the design and the maintainability of the analysed system.

Reducing the maintenance effort: One of the main driving force behind the increasing interest in reverse engineering has ever been the intent to reduce the maintenance effort [79]. A large percentage of the total time required to make a change is spent in comprehending the system and its source code. This is mainly due to the lack of an appropriate and accurate documentation and of sufficient domain knowledge [36]. Reverse engineering techniques have the potential to alleviate these problems and thus reduce maintenance effort because they provides a means of obtaining the missing information [13, 79].

In conclusion, it is worth noting that reverse engineering techniques represent the first and the most important step within the *re-engineering* process [139]. In particular, such process [13, 14] encompasses a two-step approach that aims at examining and altering a target system in order to implement the desired modifications [79].

In more details, firstly reverse engineering techniques are used to the target system in order to understand it and represent it in new forms [79, 139]. After-

	Objectives	Benefits
1	To Recover lost information	Enhances understanding and aids the identification of errors.
2	Re-documentation	Improves the documentation of the system; Provides alternative views of the system.
3	To extract reusable components	Supports identification of duplications and refactoring of reusable knowledge.
4	To reduce maintenance effort	Improve the quality and the comprehension of the system.

Table 1.1: Summary of the objectives and benefits of reverse engineering (adapted from [79])

wards, *forward engineering* [154, 169] methods are applied to actually implement and integrate the modifications, thus leading to the "new" enhanced system [79].

The overall re-engineering process is typically represented in the literature by the well-known *horseshoe model* [98].

1.1.2 Typical Approaches to Reverse Engineering

When analysing a program fro understanding, maintainers create multiple mental models of the system, aiming at identifying the *concepts* and their relationships [31, 92]. However, if such system models are not available in the documentation, maintainers repeatedly recover them from the source code and from other sources.

Automated reverse engineering can ease a tedious and error-prone manual process of recovering these information about the system [92], giving rise to the definition of several approaches aimed at supporting the practitioners in their duties.

Reverse engineering techniques may be roughly classified in two distinct categories, according to the kind of analysis they apply on the *software artefacts*, namely the *Static* and the *Dynamic Program Analysis*.

On the one hand, *static analysis*-based techniques are performed without actually executing the program, but all the relevant information are gathered directly from the source code or the object code [185].

Conversely, approaches based on *dynamic analysis* methods require the execution of the analysed program on "real" or "virtual" processors, in combination with some preliminary code *instrumentation* and an accurate selection of input data, in

order to produce interesting behaviours.
The *execution traces analysis* [22] and the *code coverage* [140] are examples of approaches belonging to this second category.
Furthermore, there are also techniques belonging to both classes, since they could be based on static or dynamic analysis methods as well.
Program slicing is a good representative for this kind of approaches, whose definition considers two different application variants [102, 183].

Nevertheless, despite the different kind of analyses performed on the program, all the reverse engineering approaches try to infer and to understand the behaviour and the architecture of a large software system [139].

As a matter of fact, an important discipline of reverse engineering is the *architecture recovery*, which deals with recovering the subsystems of a software and the dependencies between them [131]. In particular, the architecture of a software involves the organisation of a software system as a composition of artefacts, mainly focused on their interaction and relationships.

The comprehension of a software system at an architectural level is required in many cases [131] including:

- determining whether a system has the ability to fulfil its requirements;
- adapting a system to changing requirements;
- estimating the costs and risks of a change;
- enabling the re-use of components across several projects;
- defining product family architectures.

The different approaches proposed in the literature for the recovery of architectural information exploit structural dependencies between software artefacts to group related artefacts [11, 28, 141, 186], or rely on the lexical information provided by programmers in the source code [42, 108, 163]. In particular, the source code lexicon represents a key source of information, since developers usually embed in its terms, namely identifiers and comment, their domain knowledge about the system. As a matter of fact, to date more and more automatic reverse engineering tools exploit the lexical information of the source code to support different

maintenance tasks, such as locating concepts [54], or recovering the traceability links [12, 49].

Most of these lexical-based tools [108] usually complement typical static code analysis techniques with Information Retrieval (IR) approaches [130], with the often implicit assumption that the same words are used whenever describing a particular concept [115].

Nevertheless, programmers usually name the concept they want to represent by creating identifiers with multiple words, called *multi-word* identifiers (e.g., `toString`, `DynamicTable`) [61]. Therefore, lexical-based software maintenance tools require more advanced techniques that are able to *normalise* [25, 26, 115] the source code lexicon. In particular, such normalisation process implies that identifiers composed by multiple words are correctly split and that all possible occurring abbreviations are mapped to the corresponding (dictionary) words.

1.2 Source Code Vocabulary Normalisation

It is widely recognised that source code identifiers play a fundamental role in supporting software analysis/maintenance tasks [25, 37, 52, 54, 86, 116, 118]. Indeed, since the documentation of many software systems is often limited and/or outdated, the lexical information provided within the code represent one of the most valuable sources for program comprehension [126], since developers usually communicate their intent and their domain knowledge by means of significant names of identifiers. In addition, the code intrinsically provides the most up-to-date information about the latest changes of an evolving system [80]. As a matter of fact, this precious information is exploited by a number of tools suited to support analysis activities, such as software clustering [42, 108], concept location [153], source code summarisation [81, 170], or recovering traceability links [12]. Popular examples include ADAMS [49] or FLAT3 [162].

The common idea is to infer the "concepts" covered by a software artefact, given the lexical information provided in its code. Unfortunately, the effectiveness of these tools is strongly dependent on the way identifiers have been defined by programmers. In fact identifiers are usually created by concatenating multiple words according to some naming conventions, such as capitalising first letters of

each word (e.g. `toString`) and/or using special characters, like the underscore (e.g. `to_string`) [61, 116]. Thus, a *splitting* step is required by these tools in order to get all the basic words constituting an identifier [126]. However, in many cases programmers break down these conventions (e.g. `IRDocument`, `MAXSIZE`) [61], making an automatic splitting algorithm difficult to implement.

Moreover, another typical programming habit is to shorten identifier names by using abbreviations and/or acronyms. This is so diffuse that Hill et. al., in [85], report that within the source code abbreviated forms of words are more frequent than the expanded ones. This phenomenon heavily challenges the effectiveness of software maintenance tools in exploiting lexical information, and consequently word expansion approaches could be very suitable in this context.

1.2.1 Problem Statement

To split multi-word identifiers, most existing software maintenance techniques usually employe trivial algorithms that rely on coding conventions [61]. Typical examples are the *Camel Case* [2]) (e.g., `buildTree`), and the *Snake Case*[*] naming conventions [1] (e.g., `build_tree`). When simple conventions are used, the splitting of multi-word identifiers is straightforward. However, there are many cases where conventions break down [61], thus affecting the effectiveness and the applicability of these kind of algorithms. In fact, developers may decide to alternate the case of entire words instead of single characters in order to improve the readability of the identifiers (e.g., `UTFtoASCII`) [61], or to not alternate letter cases at all, i.e., the *same-case* identifiers (e.g., `MAXINT`).

In these cases, no cues are available to identify the different composing words, and smarter and more advanced algorithms are required to split this kind of identifiers

Another important aspect to consider is that developers make a heavy use of abbreviations when composing identifiers names (e.g., `getNextElem`). Therefore, lexical-based tools must embed an additional step in their processing to map every possible occurring abbreviations, i.e., the *short forms*, to the corresponding original (dictionary) words, i.e., the *long forms*.

[*]This name derives from the snaky shape assumed by identifiers written using this style.

Hill et al. [85] group the *short forms* in two distinct categories: *single-word* and *multi-word* short forms, based on whether they can be mapped to a single or to multiple words.

Single-word short forms are further distinguished in (I) *prefix* short forms if the abbreviations are obtained by dropping the last part of the corresponding expansion (e.g., `obj` standing for **obj**ect); and (II) *dropped-letters*, when some letters are removed, not including the first one (e.g., `msg` as for **M**es**S**a**G**e).

Empirical studies conducted in [126] indicate that vowels are the most typical removed letters.

On the other hand, *multi-word* abbreviations are distinguished in *acronyms* and *combinations* respectively when only or at least one letter of each composing word is considered. Examples of these two types of short forms are `awt` for **A**bstract **W**indows **T**oolkit) (acronym) and `OID` for **O**bject **ID**entifier) (combination).

However, it is worth noting that often there is no unique or obvious way to expand a given abbreviation, but the correct expansion could likely depend on the particular domain of the system. For instance, the `pnt` abbreviation could be expanded in terms such as `paint`, `pointer` or `point`, which are equally correct possible solutions.

This observation also holds for the splitting of identifiers, where several consistent decompositions could be produced, due to the intrinsic ambiguity of the natural language. For example, `findent` may be split in `f|indent` or `find|ent`, both being valid, but only one being correct for the given context. As a result, additional disambiguation strategies are needed in order to consider the different domain of the system.

To this aim, in recent years some research efforts are being devoted to the definition of new *source code vocabulary normalisation* techniques [61, 66, 80, 85, 117, 126].

1.2.2 Related Work

The important role of identifiers in program comprehension tasks motivates the large body of relevant work proposed in literature.

The work presented in [66] by Feild et al. represents the first approach to deal

with the problem of automatically splitting identifiers. In this work two approaches to same-case identifier splitting are provided, based on a greedy optimisation algorithm and on some different identifier metrics respectively. The former exploits an English dictionary (*ispell*[†]) and a list of known abbreviations to recursively search for the longest prefix and/or suffix of a term, to identify splitting markers. On the other hand, the latter relies on the combination of different metrics (e.g., word length), to determine splitting results. The approach has been evaluated on 4,000 randomly chosen identifiers gathered from 746,345 C, C++, Java and Fortran systems and results have been compared with the ones achieved by a random splitting algorithm intended as baseline.

Another splitting approach, known as *Samurai*, has been proposed by Enslen et al. [61]. It is based on the assumption that an identifier is composed by words that should appear elsewhere in the code. Thus the algorithm determines likely identifier splittings according to the different frequencies of words mined from the source code. The technique has been evaluated on over 8,000 identifiers extracted from open source Java systems. Interestingly, Samurai only relies on the domain knowledge embedded by programmers in the code, without exploiting any external dictionary to identify the different words composing a multi-word identifier.

The first works focusing on the expansion of abbreviations (*short forms*) occurring in source code to their corresponding *long forms* are the ones by Lawrie et al. [117] and by Hill et al. [85]. The former exploits different lists of potential terms, gathered from code, comments, and keywords of the programming language and an English dictionary. On the other hand, the latter proposes an approach called *AMAP*, based on a set of multiple regular expressions applied on different scopes. In particular, the AMAP approach has been evaluated considering 250 abbreviations randomly selected from five open source Java systems. Another remarkable contribution provided by Hill et al. [85] regards the definition of a detailed taxonomy of short forms grouped in the two main categories described in the previous Section (Section 1.2.1)

The abbreviation expansion problem has been also investigated by Madani et al. [126], who, to the best of our knowledge, provides the first technique to focus on both splitting and expansion of identifiers. In particular the proposed technique

[†] http://www.gnu.org/software/ispell/ispell.html

follows a two-step process: firstly it determines the splitting of identifiers whose compound terms are dictionary words. Then it tries to reconstruct the considered identifier by inferring the set of potential word transformations originally applied by the developers. The approach is based on the adaptation of the Dynamic Time Warping (DTW) algorithm aiming at finding near optimal matchings between identifier's substrings and words in an English dictionary. The overall approach has been evaluated using a manually-built oracle gathered from two open source software systems written in C and Java respectively.

Another splitting and expansion approach, named *TIDIER*, has been proposed in [80]. TIDIER applies the DTW matching algorithm in combination with a set of different dictionaries of terms, representing contextual information and specialised knowledge. Authors conducted an extended case study where they assess the effectiveness of their approach in both splitting and expanding identifiers by using a sample of more than 1,000 identifiers randomly selected from a set of 340 open source C projects.

Finally, Lawrie et al. propose the *Normalize* algorithm [115] that applies a two-step approach to split identifiers into their constituent parts and to expand possible occurring abbreviations. The first step is accomplished by the "generate and test algorithmic strategy", named *GenTest*: the algorithm *generates* all the possible splittings of an identifier compound name and then it *tests* a scoring function for each splitting, returning the best results. The definition of such scoring

Technique	**Identifier Splitting**	**Abbreviations Expansion**
Greedy Splitting Algorithm [66]	✓	✗
Neural Networks Splitting [66]	✓	✗
Samurai [61]	✓	✗
Abbreviation Expansion Alg. [117]	✗	✓
AMAP [85]	✗	✓
Normalize [113, 115]	✓	✓
DTW-based Algorithm [126]	✓	✓
TIDIER [80]	✓	✓

Table 1.1: State-of-the-art Identifier Splitting and Abbreviation Expansion Techniques

function is obtained by the combination of multiple metrics, whose weights are statistically estimated by a *Generalized Linear Multiple Model* [115]. On the other hand, the expansion of abbreviations is achieved by using wild card string matching and phrase finder tools. The same authors refines this abbreviation expansion technique in [113], where the *Normalization* algorithm integrates a strategy that determines the most likely expansion of a term. In particular, this strategy considers the co-occurrences of terms with (I) terms appearing in the analysed short forms, and (II) terms appearing in the function from which the identifier has been extracted.

Table 1.1 summarises the list of the related works, together with an indication of the considered problem.

1.3 Software Re-modularisation

Software architecture plays an important role in at least six aspects of software development: understanding, reuse, construction, evolution, analysis and management [57].

Specifying the architectural structure of a system is a significant issue, especially in case of large and complex system [131]. As a result, several approaches have been proposed in the literature to support the *software architecture recovery* (SAR) [57]. Many of these techniques derive *architectural views* of the subject system from the source code by applying some clustering analysis techniques[‡] to software artefacts, considered at different levels of granularity (e.g., at *classes* level) [57].

1.3.1 Problem Statement

Architectural information represents an important resource for software maintainers to aid the comprehension, the analysis, and the maintenance of large and complex systems [73]. In fact, software architectures provide *models* and *views* representing the relationships among different software components according to a particular set of concerns [57, 155]. However, unlike classes or packages, this information do not have an explicit representation in the source code. Moreover, the

[‡]A more detailed description of Un-supervised machine learning algorithms in general and of clustering algorithms in particular is reported in Chapter 2

external documentation is usually not present or outdated. Therefore, the existing code remains the most updated source of information to exploit in order to automatically retrieve and reconstruct the architecture of a software system [131, 165].

One of the typical tasks for the maintainers is to locate groups of software artefacts that deals with a specific topic, in order to modify them. For instance, a maintainer could be interested in grouping all the classes that handle a given concept in the application domain, or that provide related functionalities. As a result, the analysed system is *re-modularised* into meaningful subsystems that are easier to maintain and comprehend.

In the literature this particular analysis (sub)task of SAR approaches is usually referred as the *software re-modularisation* process.

The greater part of the approaches for software re-modularisation apply clustering algorithms to large software systems, to partition them into meaningful subsystems [103, 132].

A number of these approaches generally attempt to discover clusters by analysing structural dependencies between software artefacts [11, 28, 141, 186]. However, if the analysis is based on the sole structural aspect, a key source of information about the analysed software system may be lost, i.e., the domain knowledge that developers embed in the source code lexicon. As a consequence, some effort is being devoted to investigate the use of lexical information, namely source code comments and identifiers, for software re-modularisation [42, 108, 164, 165].

Typical steps in a clustering-based re-modularisation process are [103]:

1. Select entities (artefacts) to be modularised and their corresponding granularity level (e.g., classes). The overall modularisation process is based on *features* (attributes) possessed by the entities, namely the previously cited structural dependencies or the source code lexicon.

2. Identify the similarity measures and algorithms to be employed in the clustering analysis.

3. Evaluate the different partitions of software artefacts generated by the selected clustering algorithm.

1.3.2 Related Work

The definition of effective solutions for documenting software architectures is a longstanding and relevant research topic in the field of software maintenance [57, 103, 108, 176, 179].

The greater part of the approaches for SAR applies clustering algorithms to large software systems [103, 132], whose most relevant aspects will be described in the remainder of this Section. A complete and extensive survey of SAR techniques is proposed by Ducasse et al. [57] where authors provide an accurate taxonomy of proposed approaches based on the analysis of five distinct aspects, namely the *goals*, the *process*, the *inputs*, the *techniques* and the *outputs*.

Table 1.1 summarises the state-of-the-art regarding software clustering for the recovery of software architectures.

To better provide a detailed overview of different approaches, in the following the related literature is presented with respect to the information exploited in the clustering process, namely *structural information*, *lexical information*, and their combinations.

Structural-based approaches: The works proposed by Wiggerts [186] and by Anquetil and Lethbridge [11] represent the first two contributions to semi-automatic approaches for the clustering of software entities. In particular, in [11] authors present a comparative study of different hierarchical clustering algorithms based on structural information. However the proposed solutions require human decisions (e.g., cutting points of the dendrograms) to get the best partition of software entities into clusters. Similarly, Tzerpos and Holt [177] present a comparative study of a number of software clustering algorithms aiming at investigating the *stability* of modularisation results obtained on a set of different software systems. The comparison is conducted generating randomly "perturbed" versions of an example system. Differences between the partition identified by the clustering algorithms and the original partition of the system are measured by using the MoJo distance [178].

Maqbool and Babri in [132] highlight the features of hierarchical clustering research in the context of SAR. Special emphasis is posed on the analysis of different

Approach Type	Authors and Reference	Clustering Algorithm	Technique
Structural	Anquetil and Lethbridge [11]	Hierarchical Clustering	Semi-automatic
	Mitchell and Mancoridis [141]	BUNCH	Automatic
	Doval et al. [55]	Genetic algorithms	Automatic
	Mahdavi et al. [127]	Genetic algorithms	Automatic
	Bittencourt and Guerrero [28]	Edge Betweenness; K-means; Module quality; Design Matrix.	Semi-automatic
	Wu *et al.* [187]	Hierarchical Clustering; Prog. Comp. Patterns; BUNCH.	Semi-automatic
	Tzerpos and Holt [177]	Hierarchical Clustering	Semi-automatic
	Sartipi and Kontogiannis [161]	Data mining Techniques	Semi-automatic
Lexical	Kuhn et al. [108]	K-Means	Semi-automatic
	Risi et al. [156]	K-means	Automatic
	Scanniello et al. [165]	K-means	Automatic
Lexical and Structural	Maqbool and Babri [132]	Hierarchical Clustering	Semi-automatic
	Maletic and Marcus [128]	Minimum Spanning Tree	Semi-automatic
	Adritsos and Tzerpos [10]	*LIMBO*	Semi-automatic
	Scanniello et al. [163, 164]	K-means	Automatic

Table 1.1: Overview of architecture recovery approaches

similarity and distance measures that could be effectively used in clustering software artefacts. The main contribution of the paper is, however, the analysis of two clustering based approaches and their experimental assessment. The discussed approaches try to reduce the number of decisions to be taken during the clustering. They also conducted an empirical evaluation of the clustering-based approaches on four large software systems.

Mitchell and Mancoridis in [141] present a novel clustering algorithm, named *Bunch*. Bunch produces system decompositions applying search based techniques in combination with several heuristics, such as the *coupling* and *cohesion* of produced partitions, specifically designed for the clustering of software artefacts. In particular, the coupling and the cohesion heuristics are defined in terms of *intra-* and *inter-clusters* dependencies respectively. The evaluation of the produced partitions has been conducted according to qualitative and quantitative empirical investigations. Similarly, Doval et al. [55] propose a structural approach based on genetic algorithms to group software entities in clusters. A search based approach is also proposed in [127]. In order to automate the software partitioning, the authors use dependencies between modules to maximise cohesion within each cluster and to minimise coupling between clusters.

Clustering algorithms based on structural information have also been used in the analysis of the software architecture evolution [28, 187]. Wu et al. in [187] present a comparative study of a number of clustering algorithms: (a) hierarchical agglomerative clustering algorithms based on the Jaccard coefficient [89] and the single/complete linkage update rules [130]; (b) an algorithm based on program comprehension patterns that tries to recover subsystems that are commonly found in manually-created decompositions of large software systems; and (c) a customised configuration of an algorithm implemented in Bunch [141]. Similarly, Bittencourt and Guerrero [28] present an empirical study to evaluate four widely known clustering algorithms on a number of software systems implemented in Java and C/C++. The analysed algorithms are: Edge betweenness clustering, k-means clustering, modularisation quality clustering, and design structure matrix clustering.

Sartipi and Kontogiannis [161] present an interactive approach composed of four phases to recovery cohesive subsystems within C systems. In the first phase

relations between C programs are extracted. In the second phase these relationships are used to build an attributed relational graph, while in the third phase the graph is manually or automatically partitioned using data mining techniques. A case study is conducted to assess the validity of the approach.

Lexical-based approaches: Software clustering approaches exploiting lexical information are based on the idea that the lexicon provided by developers in the source code represent a key source of information. In particular, such techniques mine relevant information from source code identifiers and comments based on the assumption that related artefacts are those that share the same vocabulary.

The approach proposed by Kuhn et al. [108] constitutes one of the first proposals in this direction defining an automatic technique based on the application of the Latent Semantic Indexing (LSI) method [51]. The approach is language independent and mines the lexical information gathered from source code comments. In addition, the approach enables software engineers to identify topics in the source code by means of labelling of the identified clusters. To identify how the clusters are related to each other a correlation matrix is used. The authors perform a qualitative analysis of the clustering results, while no quantitative analysis is executed.

Similarly, Risi et al. [156] propose an approach that uses the LSI and the k-means clustering algorithm to form groups of software entities that implement similar functionality. A variant based on fold-in and fold-out is introduced as well. Furthermore this proposal provides an important contribution on the analysis of computational costs necessary to assess the validity of a clustering process.

Scanniello et al. [165] present an approach to automate the software system partitioning. This approach first analyses the software entities (e.g., programs or classes) and uses LSI to get the dissimilarity between the entities, which are grouped using iteratively the k-means clustering algorithm. The approach is implemented in a prototype of a supporting software system to partition Java and C/C++ software systems. To assess the validity of the approach a case study on open source software systems has been conducted.

Structural and Lexical-based approaches: Maletic and Marcus in [128] propose an approach based on the combination of lexical and structural information to support comprehension tasks within the maintenance and reengineering of software systems. From the lexical point of view, they consider problem and development domains. On the other hand, the structural dimension refers to the actual syntactic structure of the program along with the control and data-flow that it represents. Software entities are compared using LSI, while file organisation is used to get structural information. To group programs in clusters a simple graph theoretic algorithm is used. The algorithm takes as input an undirected graph (the graph obtained computing the cosine similarity of the two vector representations of all the source code documents) and then constructs a Minimal Spanning Tree (MST). Clusters are identified pruning the edges of the MST with a weight larger than a given threshold. To assess the effectiveness of the approach some case studies on a version of Mosaic are presented and discussed.

Andritsos and Tzerpos in [10] present *LIMBO*, a hierarchical algorithm for software clustering. The clustering algorithm considers both structural and non structural attributes to reduce the complexity of a software system by decomposing it into clusters. The authors also apply LIMBO to three large software systems.

Scanniello et al. [163] present a two phase approach for recovering hierarchical software architectures of object oriented software systems. The first phase uses structural information to identify software layers [164]. To this end, a customisation of the Kleinberg algorithm [100] is used. The second phase uses lexical information extracted from the source code to identify similarity among pairs of classes and then partitions each identified layer into software modules. The main limitation of this approach is that it is only suitable for software systems exhibiting a classical tiered architecture.

1.4 Clone Detection

To date code duplications (also called *software clones* or simply *clones*) occur frequently in large systems: ad-hoc reuse through copy-and-paste is a common habit among developers that affects the maintainability of software systems [139]. Several authors report that within large software systems, duplicated code usually accounts from the 7% to the 23% of the total code [18, 19].

The presence of clones in a software system is considered risky in the execution of maintenance operations [21]. However, clones in software are not usually documented, and their identification is not a trivial task in case of large systems. Therefore, automatic approaches are required to support the software maintainer.

As a result, in recent years the identification of code clones has become a very active research area [21, 159]. In particular, the different approaches proposed in the literature are more or less automated and require different levels of expertise to let maintainers effectively use them. These approaches generally take into account either the syntactic structure (e.g., abstract syntax tree) or lexical information (e.g., the signature of a function) [104].

However, code duplications represent an issue not only for the maintenance but also for the evolution of a system. In fact, the different duplications inevitably merge with existing code during the evolution and it becomes unclear which part of the code relate to which source of change [92]. To this aim, many research effort are being devoted to maintain software clones during the evolution [149], tracking the different changes in the code in order to distinguish copies from the originals [16, 56, 107]. This particular task is referred in literature as the problem of *code provenance* [107].

1.4.1 Problem Statement

Reasons why programmers duplicate code are manifold. The most well known is a common bad programming practice: copying and pasting [129, 172].

Duplications and redundancies make hard to understand the many code variants as they increase the size of the code, and the effort necessary to maintain it [139]. In particular, software clones should be known and well documented in order to apply consistent changes.

The main issue in the management of clones is that errors in the original version must be fixed in every clone. Furthermore, clones make difficult also the executions of other kinds of maintenance operations, e.g., extensions or adaptations. Thus, the presence of code clones is one of the factors that complicates the maintenance and evolution of a software system [58]. To make things worse, code clones are usually not documented and so their location in the source code is not known. This implies that maintainers have to detect them [77] in order to properly perform maintenance operations. In case of small-size software systems the detection of clones may be manually performed but on large software systems it can be accomplished only by means of automatic or semiautomatic approaches [104].

More recently a numbers of studies pointed out that the presence of clones may be not so risky for the software maintenance and evolution [16, 96, 173]. Despite the controversial point on the risks related to the presence of code clones, there is a large consensus on the need of detecting them [104].

There is no agreement in the research community on the exact notion of redundancy and cloning [139]. Ira Baxter's definition of clones express this vagueness [20]:

> Clones are segments of code that are similar according to some definition of similarity. (Baxter, 1998)

In particular, this definition allows different notions of similarity, which can be based on the program text or on some notions of semantic aspects. Furthermore, similarities on program text may be based on lexical or syntactic structures.

On the other hand, semantic similarities relates to observable behaviour: a fragment of code A is "semantically" similar to another fragment B if B subsumes the functionality of A, i.e., they have "similar" pre and post conditions [139].

Nevertheless, it is worth considering that the two different notions of similarity are not necessarily equivalent. In fact, it is **not** always true that "two fragments of code are semantically similar if and only if their program text is similar".

In other words, even if the program text of two fragments of code is similar, their behaviours are not necessarily equivalent or subsumed. For instance, two pieces of code may be identical at the textual level including all variable names that occur within, but the variable names are bound to different declarations in the different

contexts. Then, the execution of the code changes different behaviour [139].

```
1  int sum = 0
2  void a_function(Iterator iter){
3      for (Item item = iter.first(); iter.has_next(); item = iter.next()){
4          sum = sum + item.value()
5      }
6  }
7  void b_function(Iterator iter){
8      int sum = 0
9      for (Item item = iter.first(); iter.has_next(); item = iter.next()){
10         sum = sum + item.value()
11     }
12 }
```

Listing 1.1: Example of code clones (adapted from [139])

The two functions reported in Listing 1.1 represent an example of two textually equivalent clones in the line range of 3-5 and 9-11, respectively. The two functions iterate on a collection of numbers, summing the value of each element to the variable `sum`. However, while the former (i.e., `a_function`), sets the *global* variable `sum`, the latter (i.e., `b_function`) sets a *local* variable with the same name. As a consequence, even if their program text is exactly the same, they are not semantically equivalent at a concrete level.

1.4.2 Clone Terminology and definitions

A first step towards a more formal definition of software clones is provided in [92], where authors defines *software clones* as:

> *program structures of considerable size that exhibit significant similarity. The actual size and similarity (which can be measured, for example, in terms of replicated lines of code) vary depending on the context. Clones may represent similar program structures of any kind and granularity.*

In particular, two different *types* of clones are distinguished:

1. *Simple clones*: the same or similar segments of contiguous code, such as class methods or fragments of method/function implementation.

2. *Structural clones*: patterns of interrelated components (e.g., collaborating classes) representing design solutions repeatedly applied by programmers to solve similar problems and similar program modules or subsystems comprising many components.

These two definitions give to software clones a broader meaning as they take into account not only duplications at a code level, but also at design or architecture levels (e.g., common design patterns or duplicated subsystems). However, even if these definitions are too general, they emphasise the difficulties of defining the concept of similarity (already expressed in Baxter's definition [20]) in precise and descriptive terms. In fact, the notion of similarity necessarily involves the human judgement and, therefore, is inherently subjective [92].

To this aim, Kapser et al. [96] elicited judgements and discussions from world experts regarding what characteristics define a code clone. Their experiments concluded that less than a half of the clone candidates they presented to these experts had 80% of agreement amongst the judges. However, judges appeared to differ primarily in their criteria for judgement rather than their interpretation of the clone candidates [139].

More recently, a more precise definition of code clones has been proposed that is now largely accepted in the research community. This definition categorises the different kind of duplications in a set of four different *Types* [21, 159]. Terms and definitions reported below refer to the survey on clone detection research provided by Roy et al. in [159].

Definition 1.1. Code Fragment. *A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition,* `begin-end` *block, or sequence of statements.*
A CF is identified by the name of its source file and its corresponding line numbers in the original code. Thus, a code fragment is denoted as a triple

$$CF = (CF.filename, CF.beginline, CF.endline)$$

Definition 1.2. Code Clone. *A code fragment CF_2 is a clone of another code fragment CF_1 if they are similar by some given definition of similarity. Two fragments that are similar to each other form a* clone pair *(CF_1, CF_2), and when many fragments are similar, they form a* clone class *or* clone group.

This definition is invariant to the different possible transformations applied to the two code fragments, i.e., if CF_1 and CF_2 form a clone pair, $f(CF_1)$ and $f(CF_2)$ still remain clones.

Definition 1.3. Clone Types. *There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (i.e., independent of their text). In particular, there are four different* Types *of clones based on textual (Type 1 to 3) [21] and functional (Type 4) [71, 101] similarities:*

> **Type 1***: Identical code fragments except for variations in whitespace, layout and comments.*
>
> **Type 2***: Syntactically identical fragments except for variations in identifiers, literals, whitespace, layout and comments.*
>
> **Type 3***: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, whitespace, layout and comments.*
>
> **Type 4***: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.*

The taxonomy of textual-based clones (Type 1 to 3) has been further detailed and extended by Evans et al. [64] and also reported in [174, 175]:

- **Exact Clone (Type 1)** is an exact copy of consecutive code fragments without modifications. That is, the transformation to the code is the identity.
- **Parameter-substituted clone (Type 2)** is a copy where only parameters (identifiers or literals) have been substituted. Given a suitable structure substitution, the transformed copy is a Type 1 clone [64].

- **Structure-substituted clone (Type 3)** is a copy where program structures (complete subtrees in the syntax tree) have been substituted. Given a suitable structure substitution, the transformed copy is a Type 1 clone [64]. For parameter-substituted clones, we can replace one leaf in the syntax tree by another leaf. For structured-substituted clones, larger subtrees can be replaced.
- **Modified clone (Type 3)** is a copy whose modifications go beyond structure substitutions by added and/or deleted code.

Similarly, Kim et al. [99] define subtypes for Type 4 clones, i.e., functional clones:

- Control replacement with semantically equivalent control structures (Listing 1.2).
- Statement re-ordering without modifying the semantics (Listing 1.4).
- Statement insertion without changing the computation (Listing 1.5).
- Statement modification with preserving memory behaviour (Listing 1.3).

```
PyObject *PyBool_FromLong(long ok) {
    PyObject *result;
    if (ok)
        result = Py_True;
    else
        result = Py_False;
    Py_INCREF(result);
    return result;
}
```

```
static PyObject * get_pybool(int istrue) {
    PyObject *result = istrue? Py_True : Py_False;
    Py_INCREF(result);
    return result;
}
```

Listing 1.2: Type 4 clone: *control replacement* from Python system. The `if-else` statement is substituted by the ternary conditional operator (adapted from [99])

```
void appendPQExpBufferChar(PQExpBuffer str, char ch) {
    /* Make more room if needed */
    if (!enlargePQExpBuffer(str, 1))
        return;
    /* OK, append the data */
    str->data[str->len] = ch;
    str->len++;
    str->data[str->len] = ''\0;
}
```

```
void appendBinaryPQExpBuffer(PQExpBuffer str, const char *data, size t datalen) {
    /* Make more room if needed */
    if(!enlargePQExpBuffer(str, datalen))
        return;
    /* OK, append the data */
    memcpy(str->data + str->len, data, datalen);
    str->len += datalen;
    str->data[str->len] = ''\0;
}
```

Listing 1.3: Type 4 clone: *preserving memory behaviour* from PostgreSQL system. (adapted from [99])

```
static const char *set_access_name(cmd_parms *cmd, void *dummy, const char *arg) {
    void *sconf = cmd->server->module config;
    core_server_config *conf = ap_get_module_config(sconf, &core_module);
    const char *err = ap_check_cmd_context(cmd, NOT_IN_DIR_LOC_FILE|NOT_IN_LIMIT);
    if (err != NULL) {
        return err;
    }
    conf->access name = apr_pstrdup(cmd->pool, arg);
    return NULL;
}
```

```
static const char *set_protocol(cmd_parms *cmd, void *dummy, const char *arg){
    const char *err = ap_check_cmd_context(cmd, NOT_IN_DIR_LOC_FILE|NOT_IN_LIMIT);
    core_server_config *conf = ap_get_module_config(cmd->server->module_config,
                                                    &core_module);
    char *proto;
    if (err != NULL) {
        return err;
    }
    proto = apr_pstrdump(cmd->pool, arg);
    ap_str_tolower(proto);
    conf->protocol = proto;
    return NULL;
}
```

Listing 1.4: Type 4 clone: *statement reordering* from Apache system (adapted from [99])

```c
const char * GetVariable(VariableSpace space, const char *name) {
    struct _variable *current;
    if (!space)
        return NULL;
    for (current = space->next; current; current = current->next) {
        if (strcmp(current->name, name) == 0) {
            return current->value;
        }
    }
    return NULL;
}
```

```c
const char * PQparameterStatus(const PGconn *conn, const char *paramName) {
    const pgParameterStatus *pstatus;
    if(!conn || !paramName)
        return NULL;
    for (pstatus = conn->pstatus; pstatus != NULL; pstatus = pstatus->next) {
        if (strcmp(pstatus->name, paramName) == 0)
            return pstatus->value;
    }
    return NULL;
}
```

Listing 1.5: Type 4 clone: *statement insertion without changing computation* from PostgreSQL system (adapted from [99])

1.4.3 Related Work

The different research contributions for clone detection could be grouped according to the particular kind of information exploited in the analysis of source code fragments. In particular, in Table 1.1 the different proposals are grouped according to the information they exploit and the types of clones they are able to detect.

Note that our goal here is not to provide an extensive analysis of the clone detection approaches presented in the literature but to provide an overview of most relevant techniques together with a general background on the problem, necessary to introduce the proposal presented in Chapter 5.

An exhaustive survey of clone detection tools and techniques is provided in [159].

Approach Type	Technique	Author(s) and Reference	Clone Types			
			1	2	3	4
Textual	String matching	Ducasse et al. [58]	✓	✓	✗	✗
		Johnson [94]	✓	✗	✗	✗
Token	Pattern matching	Baker [18]	✓	✓	✗	✗
	Suffix-tree matching	Kamiya et al. [95]	✓	✓	✗	✗
Metrics	Metric vectors distance	Balazinska et al. [19]	✓	✓	✓	✗
Syntax Tree	Dynamic Programming	Yang [189]	✓	✓	✗	✗
	Tree matching	Baxter et al. [20]	✓	✓	✗	✗
	Suffix-tree AST	Koschke et al. [105]	✓	✓	✗	✗
	Anti-unification	Bulychev et al. [33]	✓	✓	✗	✗
	LSH	Jiang et al. [93]	✓	✓	✓	✗
Dependency Graph	Program Slicing	Komondoor et al. [101]	✓	✓	✓	✗
	PDG matching	Gabel et al. [71]	✓	✓	✓	✓
		Krinke [106]	✓	✓	✓	✗
Other	Software metrics	Leitão [121]	✓	✗	✓	✗
	Frequent Item-sets	Wahler et al. [182]	✓	✓	✗	✗
	LSI	Marcus and Maletic [133]	✓	✓	✗	✗
	Code Transformations	Roy and Cory [157, 158]	✓	✓	✓	✗
	Count Matrix	Yuan et al. [191]	✓	✓	✓	✓
	Memory state matching	Kim et al. [99]	✓	✓	✓	✓

Table 1.1: Overview of clone detection techniques

Textual-based approaches: Ducasse et al. [58] propose a language-independent approach to detect code clones, based on the following three steps: line-based string matching, visual presentation of the cloned code, and detailed textual reports of the clones.

A different approach has been proposed by Johnson [94] where the author presents a prototype based on fingerprints to identify exact repetitions (i.e., Type 1 clones) of text in the source code of large software systems.

Both these techniques leverage the efficiency and the scalability properties of string-matching algorithms that make them perfectly suitable for the analysis of large software systems. However, their detection capabilities are very limited and restricted to very similar textual duplications (Clones of Type 1 and 2).

For the sake of completeness, it is worth mentioning that textual approaches are also used to detect cloned web pages, such as in [50]. Clone detection approaches for web applications have not been included in Table 1.1 as the types of identified clones do not follow the classification proposed in the literature.

Token-based approaches: Baker [18] suggests an approach to identify duplications and near-duplications (i.e., copies with slight modifications) in large software systems. The proposed approach is able to identify source code copies that are substantially the same except for global substitutions.

Similarly, Kamiya et al. [95] use a suffix-tree matching algorithm to compute token-by-token matching among source code fragments. The authors adopt optimisation techniques that normalise token sequences. This is due to the fact that the underlying algorithm may be expensive when used on large software systems. To assess the validity of their approach, the authors also propose a prototype of a supporting system and applied it on software systems implemented in C, C++, Java, and COBOL.

The main drawback of these approaches is that they completely disregard the syntactical information of the source code, similarly to textual-based techniques. As a consequence, these solutions may detect a large number of false clones that do not correspond to any actual syntactical unit.

Metrics-based approaches: Balazinska et al. in [19] suggest a classification schema based on software metrics to detect code clones. This approach builds metric vectors and then compute the distances between these vectors as a clue for similar code fragments. The approach detects clones of Types 1, 2, and 3. A similar approach to identify scripting functions within web pages is presented by Calefato *et al.* in [34] .

Syntax Tree-based approaches: Syntactic-based approaches exploit the information provided by Abstract Syntax Trees (AST) to identify similar code fragments. These techniques are more robust than the previous ones and are able to deal with larger degrees of modifications among the cloned code fragments. Nevertheless, they may possibly fail in case modifications concerns the inversion or the substitution of entire code blocks: the so-called *gapped-clones* [106].

Yang [189] defined a dynamic programming-based algorithm to detect differences between two versions of the same source file. Clones of Types 1 and 2 can be identified.

A similar approach is presented by Baxter et al. [20]. It is based on a tree matching algorithm that compares the different sub-trees of the AST of a given software system.

On the other hand, Koschke et al. [105] describe an approach to detect clones based on suffix trees of serialised ASTs. The main contribution of this work concerns the computational efficiency of the proposed solution. In fact, the approach is able to identify software clones in linear time and space. This approach has been extended and improved in [65] to improve clone detection effectiveness for both Java and C software systems. A case study based on an extension of the Bellon's benchmark [21] is also conducted to assess the validity of the approach.

A different approach is presented by Bulychev et al. [33], where authors propose a clone detection technique based on the *anti-unification* algorithm, widely used in natural language processing tasks.

A novel technique for detecting similar trees has been presented by Jiang et al. [93]. Authors proposed an automated algorithm and corresponding prototype tool named *Deckard*. This algorithm defines specialised characteristic vectors of each code fragment to approximate the structure of ASTs in a Euclidean space.

Locality Sensitive Hashing (LSH) is then used to cluster similar vectors using the Euclidean distance metric.

Dependency Graph-based approaches: Dependency graph-based approaches generally use algorithms to identify isomorphic sub-graphs within a graph built considering control and data flow dependencies (i.e., the program dependence graphs, PDG) of the software system to analyse. The main advantage of these techniques is that they do not depend on the particular textual representation of the code, allowing to detect also "functional" duplications (i.e., Type 4 clones), in addition to the textual based ones considered by previous approaches (Types 1-3). However the identification of isomorphic sub-graphs is a NP-hard problem and only approximated solutions may be provided.

Komondoor and Horwitz [101] propose an approach based on program slicing techniques, applied on a program dependence graph.

Moreover, a heuristic to identify isomorphic sub-graphs is proposed by Krinke in [106].

More recently, Gabel et al. [71] propose a dependency graph-based technique that maps slices of PDGs to syntax subtrees and applies the Deckard clone detection algorithm [93].

Other Approaches: In the literature techniques that combine some of the aspect discussed so far are presented as well. For example, Leitão [121] combines syntactic and semantic techniques using functions that consider various aspects of software systems (e.g., similar call sub-graphs, commutative operators, user-defined equivalences). This approach is defined to detect cloned source code of Types 1 and 2 in large software systems implemented in Lisp.

An approach based on an information retrieval technique is presented by Marcus and Maletic [133]. This approach detects clones of Types 1, and 2. To this end, similarities among source code (treated as plain text) are computed using a measure based on Latent Semantic Indexing [130].

Differently, Wahler et al. [182] present an approach based on data mining techniques to detect clones of Types 1 and 2. This approach uses the concept of frequent item-sets on the XML representation of the the software system to be

analysed.

On the other hand, Roy and Cordy [157, 158] present an approach based on source transformations and text line comparison to find clones of Types up to 3. The validity of the approach is empirically verified some C, Java and C# software systems.

Yuan et al. [191] propose a language-independent approach named *Count Matrix* Clone Detector (CMCD). The key concept behind CMCD is the so-called *Count Matrix* algorithm that is able to represent the characteristics of a code segment by counting the occurrence frequencies of every variable in pre-determined counting conditions [191]. The approach has been applied on the 16 clones scenarios proposed in and on JDK 1.6 project. Empirical results show that the approach is able to detect clones up to Type 4.

Kim et al [99] propose a *semantic* clone detection technique that is able to compare programs' abstract memory states, which are computed by a semantic-based static analyser. The approach has been evaluated in an experimental study on three large open source projects written in C that assess the ability of the technique in detecting clones up to Type 4. Another important contribution of this paper concerns the proposal of a more refined characterisation of the Type 4 clones.

I don't fear computers. I fear the lack of them.

Isaac Asimov

2

Machine Learning and Pattern Matching Techniques

MACHINE Learning (ML) is defined as "the systematic study of algorithms and systems that improve their knowledge or performance with experience." [68]. As a matter of fact, machine learning comprises algorithms and techniques that are able to gain insight from a (usually large) data set, in order to turn data into *meaningful information* [83]. This *data driven methodology* [167] represent one of the most important characteristic of ML approaches, which differentiates them from classical Artificial Intelligence techniques. On the one hand, ML approaches do not require that all the necessary knowledge (i.e., facts and rules) must be specified in the knowledge base from the very beginning [131]. On the other hand, ML algorithms are able to *generalise*, namely they are able to adapt their behaviour according to the input data and try to infer a solution. Conversely, classical knowledge base system are not capable to adapt their behaviour to new data, unless the necessary information

is incorporated into the knowledge base itself [131].
Therefore, such generalisation capabilities practically define the concept of *learning though experience* that is a specific peculiarity of ML techniques. More formally, learning may be defined as [142]:

> "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"

Moreover, Flach in [68] declares that:

> "Machine learning is concerned with using the right *features* to build the right *models* that achieve the right *tasks*".

Differently from classical definitions mainly focused on the concept of learning, the one proposed by Flach emphasises the fundamental elements (called "ingredients" by the author) constituting ML approaches. In particular, such ingredients "may come in many different forms, and need to be chosen and combined carefully to create a successful *meal*, namely the machine learning application" [68]. Indeed, the development of a ML application encompasses the following steps [83, 193]:

1. *Problem formulation*: The first crucial step concerns the formulation of a give problem in terms of the selected ML approach. Actually there are several learning methods that are based on different theoretical backgrounds and adopt different algorithmic strategies. All these aspects characterise the *task* of different ML techniques, which must be taken into consideration during the problem formulation [193]. In fact, as observed in [112], the power of ML methods does not come from a particular induction, but instead from the problem formulation.

2. *Problem representation*: The next step is to select an appropriate representation for both the data and the knowledge to be learned (i.e., the *model*). Different learning methods require different formalisms [193], e.g., some approaches are based on a vectorial representation of data, while others require to process structured data such as trees or graphs.

3. *Data collection*: Data represent the main "ingredient" for ML algorithms. In particular, ML approaches strongly depend on the quality and the quantity of the data available to carry out the learning process. Therefore, data must be properly sanitised to remove spurious values and uniformed to a unique format. To this aim, several techniques of *data manipulation* and *data crunching* are required to tackle such problems.
   Furthermore, data are fundamental for the definition of *features*, which determine much of the success of ML applications, because a model is only good as its features [68]. Roughly speaking, a feature can be thought as a function that map the data from one domain to another, i.e., the *feature space*.
   This definition will be further detailed in Section 2.4 where *Kernel methods* will be presented.

4. *Perform the learning process*: Once the data have been collected, the learning process can be accomplished. This step represents the *core* of ML algorithms [83], where the data are actually analysed. Depending on the specific strategy, data may be separated in two different sets, namely the *training* and the *test* sets, exploited in an iterative process to train and evaluate the learning, respectively (Section 2.2).

5. *Analyse and Evaluate learned Knowledge*: The final step of ML applications concerns the evaluation of performance of the learning process. Depending on the type of application, i.e., automatic or semi-automatic, this step could also be an integral part of the learning process itself. Besides performance evaluations, this step is also devoted to the resolution of practical problems, such as the *overfitting*, or the *local optima* [167], that may affect learning methods. Some possible causes could be data inadequacy, noise or irrelevant attributes in data.

A summary representation of a classical ML application is depicted in Figure 2.1. In particular, a task (red box) requires an appropriate mapping, i.e., a model from data described by features to outputs. Obtaining such a mapping is what constitutes a learning model (blue box). Depending on the particular learning

strategy, such mapping could possibly leverage of some *training data* (dashed blue line).

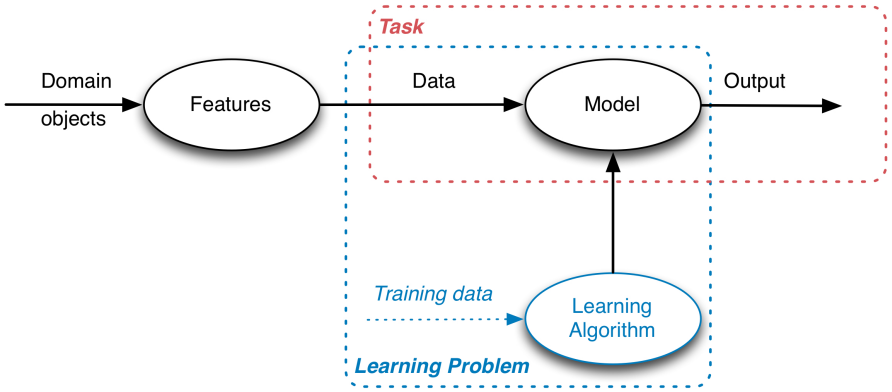

Figure 2.1: An overview of how the different components of a machine learning applications are organised (adapted from [68]).

Machine learning lies at the intersection of computer science, engineering, and statistics [83], and it can be applied to many problems. As a matter of fact, ML algorithms have proven to be of great practical value in a variety of application domains [193]: any field that needs to interpret and act on data can benefit from machine learning techniques [83].

Nevertheless, the application of ML approaches to a software engineering (SE) problem requires to address certain issues [193]: on the one hand it is necessary to (re-)formulate the SE tasks in terms of a *learning problem*, in order to allow the application of ML algorithms. On the other hand, it is important to select a technique which is appropriate to solve the problem under consideration [131]. Moreover, it is required to adapt and modify selected ML techniques in order to make them suitable to analyse software data. An extensive survey of SE approaches that leverage ML techniques is provided in [193] and in [192].

2.1 Definitions and Notations

This section recalls basic definitions and notation that will be used in the following chapters. In particular, main linear algebra notions, such as dot product and norm,

are recalled in Section 2.1.1, while basic definitions for trees and graphs structures are reported in Section 2.1.2.

2.1.1 Dot Products and Norms

Definition 2.1. (Field)*: Given a set F and two operations $*$ and $+$, the (algebraic) structure $F, *, +$ is a field, iff:*

1. $(F, +)$ *is an Abelian group with identity element equals to* $\mathbf{0}$*, namely:*
   $\forall\ f \in F,\ f + \mathbf{0} = \mathbf{0} + f = f$*;*

2. $(F \setminus \{0\}, *)$ *is an abelian group with identify element equals to* $\mathbf{1}$*, namely:*
   $\forall f' \in F, f' * \mathbf{1} = \mathbf{1} * f' = f'$*;*

3. *(*Distributivity of $*$ operator over $+$ operator*):*
   $\forall\ a, b, c \in F,\ a * (b + c) = (a * b) + (a * c)$.

Remark 2.1. Real Numbers*:*
The set of all real numbers $\mathbb{R}$ *is a Field, whose operations correspond to the standard numerical* addition *and* multiplication.

Definition 2.2. (Vector Space)*: A vector space over a field F is a set V together with two binary operations, i.e., $(V, +, *)$, s.a.:*

(Notation)*: Elements of V are called* vectors*, and elements of F are called* scalars.

(Vector Addition)*: The first operation, vector addition, takes any two vectors v and w and outputs a third vector $v + w$.*

(Scalar Multiplication)*: The second operation, scalar multiplication, takes any scalar a and any vector v and outputs a new vector $a * v$*

Therefore, the structure $[V, +, *]$ corresponds to a vector space *over the field F.*

The operations of *additions* and *multiplication* in a vector space satisfy the following axioms:

- Associativity of scalar multiplication:

$$\forall a, b \in F,\, v \in V, \quad a * (b * v) = (a * b) * v \tag{2.1}$$

- Distributivity of scalar multiplication w.r.t. vector addition:

$$\forall a \in F,\, u, v \in V,\; a * (u + v) = a * u + a * v \tag{2.2}$$

- Distributivity of scalar multiplication w.r.t. field addition:

$$\forall a, b \in F,\, v \in V,\; (a + b) * v = a * v + b * v \tag{2.3}$$

- Identity element of scalar multiplication:

$$\forall v \in V,\; 1 * v = v, \text{where 1 denotes the multiplicative identity in } F \tag{2.4}$$

- Identity element of addition (Null vector):

$$\exists\, \mathbf{0} \in V,\; \text{s.t.:}\; v + \mathbf{0} = v, \forall v \in V \tag{2.5}$$

From now on, without loss of generality, a generic vector space $[V, +, *]$ over a field F will be synthetically referred by V. In particular, we will write $\mathbb{R}^N$ instead of $[\mathbb{R}^N, +, *]$ over the field $\mathbb{R}$ to indicate the vector space of real numbers, whenever this does not lead to confusion.
Moreover, vector elements will be typed in **boldface**, i.e., $\mathbf{v}$, and $\mathbf{0}$ will indicate the *null vector*.

Definition 2.3. (Linear Combination)*: Let F be a Field and V a vector space model over F. If $\mathbf{v_1}, \ldots, \mathbf{v_n}$ are vectors and $a_1, \ldots, a_n$ are scalars, the* linear combination *of those vectors with those scalars (as* coefficients*) is defined as:*

$$a_1 * \mathbf{v_1} \; + a_2 * \mathbf{v_2} + \ldots a_n * \mathbf{v_n} = \sum_{i=1}^{n} a_i * \mathbf{v_i}$$

where $*$ correspond to the scalar multiplication *operation.*

Definition 2.4. (Dot Product): *Let V be a vector space. A* dot product *(or* scalar product*) is an application $\cdot : V \times V \to \mathbb{R}$ satisfying the following conditions:*

$$\begin{array}{lcl}
a) & \mathbf{v}\cdot\mathbf{w} = \mathbf{w}\cdot\mathbf{v} & \mathbf{v},\mathbf{w} \in V \\
b) & (\mathbf{v}+\mathbf{w})\cdot\mathbf{z} = (\mathbf{v}\cdot\mathbf{z}) + (\mathbf{w}\cdot\mathbf{z}) & \mathbf{v},\mathbf{w},\mathbf{z} \in V \\
c) & (a\mathbf{v})\cdot\mathbf{w} = a(\mathbf{v}\cdot\mathbf{w}) & \mathbf{v},\mathbf{w} \in V,\ a \in \mathbb{R} \\
d) & \mathbf{v}\cdot\mathbf{v} \geq 0 & \mathbf{v} \in V \\
e) & \mathbf{v}\cdot\mathbf{v} = 0 & \mathbf{v} = \mathbf{0} \\
f) & \mathbf{v}\cdot(\mathbf{w}+\mathbf{z}) = (\mathbf{v}\cdot\mathbf{w}) + (\mathbf{v}\cdot\mathbf{z}) & \mathbf{v},\mathbf{w},\mathbf{z} \in V
\end{array} \iff$$

Some remarks follow from the Definition 2.4:

Remark 2.2. Notation *The inner products of two vectors* $\mathbf{v}$*, and* $\mathbf{w}$ *may be also indicated using the symbols* $\langle \mathbf{v}, \mathbf{w} \rangle$*. Thus, from now on, the two notations will be used interchangeably selecting the one that aid the readability and the comprehension of the different formulations.*

Remark 2.3. Algebraic definition of the inner product*:*
The inner products of two vectors $\mathbf{v} = (v_1, v_2, \ldots, v_n)$*, and* $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ *is defined as:*

$$\langle \mathbf{v}, \mathbf{w} \rangle = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{n} v_i * w_i = v_1 * w_1 + \ldots + v_n * w_n \tag{2.6}$$

Remark 2.4. *Axioms (a) and (b) imply (f).*
Moreover, using the axiom (d), it is possible to associate to the inner product, a quadratic form, called norm*,* $\|\cdot\|$*, such that:* $\|\mathbf{v}\|^2 = \langle \mathbf{v}, \mathbf{v} \rangle$*.*

More formally:

Definition 2.5. (Norm): *The norm* $\|\cdot\| : V \to \mathbb{R}; \mathbf{v} \mapsto \|v\|$ *is a function with the following properties:*

$$\begin{array}{cc}
\|\mathbf{v}\| \geq 0 & \forall\ \mathbf{v} \in V \\
\|\mathbf{v}\| = 0 \iff \mathbf{v} = \mathbf{0} & \forall\ \mathbf{v} \in V \\
\|a\mathbf{v}\| = \|a\|\|\mathbf{v}\| & \forall\ \mathbf{v} \in V, a \in \mathbb{R} \\
\|\mathbf{v}+\mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\| & \forall\ \mathbf{v},\mathbf{w} \in V \quad \textit{(Triangular Inequality)}
\end{array}$$

Remark 2.5. *Following from Remarks 2.3 and 2.4, an equivalent algebraic formulation of the norm could be provided:*

$$\|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} = \sqrt{\sum_{i=1}^{N} v_i^2} \tag{2.7}$$

Remark 2.6. Cauchy-Schwarz's Inequality
Norms and Inner Products are connected by the Cauchy-Schwarz's Inequality:

$$|\langle \mathbf{v}, \mathbf{w} \rangle| \leq \|\mathbf{v}\| \|\mathbf{w}\|$$

The definitions of dot product (Definition 2.4), and norm (Definition 2.5) allow to formally define the notion of (internal) *angle* between two vectors $\mathbf{v}$ and $\mathbf{w} \in V$.

Definition 2.6. (Angle between two vectors)*: Let* $\mathbf{v}$ *and* $\mathbf{y}$ *be vectors in a vector space* V*. The* angle between $\mathbf{v}$ and $\mathbf{w}$*, indicated with* θ*, corresponds to:*

$$\theta = \arccos\left(\frac{\langle \mathbf{v}, \mathbf{w} \rangle}{\|\mathbf{v}\| \|\mathbf{w}\|}\right) \tag{2.8}$$

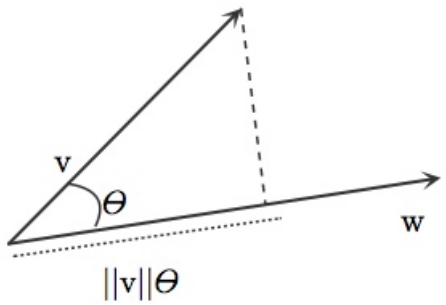

Figure 2.1: Scalar projection.

Remark 2.7. Geometric definition of the inner product*:*
Equation 2.8 leads to the following formulation:

$$\langle \mathbf{v}, \mathbf{w} \rangle = \|\mathbf{v}\| \|\mathbf{w}\| \cos\theta \tag{2.9}$$

Therefore, $\|\mathbf{v}\|\cos\theta$ *corresponds to the* scalar projection *of vector* $\mathbf{v}$ *onto the vector* $\mathbf{w}$ *(Figure 2.1)*

Definition 2.7. (Distance)*: Let* V *be a vector space. A function* $\rho : V \times V \to \mathbb{R}$ *is called a* distance *on* V *if:*

a) $\rho(\mathbf{v}, \mathbf{w}) \geq 0 \qquad \forall\ \mathbf{v}, \mathbf{w} \in V$

b) $\rho(\mathbf{v}, \mathbf{w}) = \rho(\mathbf{w}, \mathbf{v}) \quad \forall\ \mathbf{v}, \mathbf{w} \in V$

c) $\rho(\mathbf{v}, \mathbf{v}) = 0 \qquad \forall\ \mathbf{v} \in V$

The (V, ρ) *is called a* **distance space***. In addition, If* ρ *also satisfies the* **triangle inequality**

d) $\rho(\mathbf{v}, \mathbf{w}) \leq \rho(\mathbf{v}, \mathbf{z}) + \rho(\mathbf{w}, \mathbf{z}) \quad \forall\ \mathbf{v}, \mathbf{w}, \mathbf{z} \in V$

then ρ *is called a* **semimetric** *on* V*.*

Besides, if

e) $\rho(\mathbf{v}, \mathbf{w}) = 0 \quad \Rightarrow \quad \mathbf{x} = \mathbf{y}$

also holds, then ρ *is called a* **metric***, and* (V, ρ) *is a* **metric space***.*

Remark 2.8. Cosine Similarity *From Definition 2.6 and Equation 2.9, an important conclusion can be derived.*

$$\text{(Equation 2.9)} \qquad \cos\theta = \frac{\langle \mathbf{v}, \mathbf{w} \rangle}{\|\mathbf{v}\|\|\mathbf{w}\|} \tag{2.10}$$

$$\text{(Equation 2.7 and 2.6)} \qquad = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{2.11}$$

$$= \frac{\sum_{i=1}^{N} v_i \sum_{i=1}^{N} w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{2.12}$$

$$= \frac{\sum_{i=1}^{N} v_i}{\sqrt{\sum_{i=1}^{N} v_i^2}} * \frac{\sum_{i=1}^{N} w_i}{\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{2.13}$$

$$= \frac{\mathbf{v}}{\|\mathbf{v}\|} * \frac{\mathbf{w}}{\|\mathbf{w}\|} \tag{2.14}$$

$$= \langle \nu(\mathbf{v}), \nu(\mathbf{w}) \rangle \tag{2.15}$$

where the element $\nu(\mathbf{v}) = \frac{\mathbf{v}}{\|\mathbf{v}\|}$ *represent the vector* $\mathbf{v}$ *normalised by its corresponding* unit vector*, called* versor*.*

In the literature, this formulation is usually called the **cosine similarity** *[130] as it represents a similarity computation (the dot product) normalised by the norms.*

In conclusion,

Definition 2.8. (Euclidean Space)*: A set V is an* Euclidean Space *iff. V is a* metric space *that is linear and finite-dimensional.*

Remark 2.9. Euclidean Distance*: It is easy to show that*

$$\rho(\mathbf{v}, \mathbf{w}) = \|\mathbf{v} - \mathbf{w}\| = \sqrt{\sum_{i=1}^{N}(v_i - w_i)^2}$$

is a metric*.*

Remark 2.10. Inner Product*: In the context of Euclidean Space, the dot product is sometimes indicated also by the term* inner product*. Thus the two terms will be used as synonyms in case of Euclidean spaces.*

2.1.2 Graphs and Trees

Definition 2.9. (Directed Graph)*: A directed graph is a pair of sets $G = (V_G, E_G)$, where $V_G = \{v_1, \ldots, v_n\}$ is an ordered set of* nodes *and $E_G = \{e_{ij} = (v_i, v_j), \ldots, e_{kl} = (v_k, v_l)\}$ is an ordered set of pairs of nodes, called the* edges*.*

From now on, a generic graph will be indicated by $G = (V, E)$ (without the subscript G), to aid the readability of the notation. Moreover, the short form G may be used instead of $G = (V, E)$, whenever this does not lead to confusions.

Proposition 2.1. Undirected graph*:*
An undirected graph is a graph for which the following property holds:

$$e_{ij} \in E \leftrightarrow e_{ji} \in E.$$

In other words, the edge set E consists of unordered *pairs of vertices, rather than ordered pairs [47].*

Many definitions for directed or undirected graphs are the same, although certain terms have slightly different meaning in the two contexts [47]. Thus, unless differently specified, the following definitions hold for both directed and undirected graphs.

Definition 2.10. Labelled Graph: *Let $G = (V, E)$ be a graph, and Σ an alphabet of characters. If there exist functions*

$$\mathcal{L}_V : V \to \Sigma^*; v_i \mapsto \mathcal{L}_V(v_i)$$

and

$$\mathcal{L}_E : E \to \Sigma^*; e_{ij} \mapsto \mathcal{L}_E(e_{i,j})$$

then, G is a labelled graph.

Definition 2.11. (Directed Graph Product)
Let $G = (V, E, \mathcal{L})$ and $G = (V', E', \mathcal{L}')$ be two labeled graphs, with labelling functions equals to $\mathcal{L}$ and $\mathcal{L}'$,respectively. Thus, the **directed graph product** *$G_\times = G \times G'$ is defined as:*

$$V_\times = \{(v_i, v_i') \ : \ v_i \in V \wedge v_i' \in V' \wedge \mathcal{L}(v_i) = \mathcal{L}'(v_i')\} \tag{2.16}$$

$$E_\times = \{((v_i, v_i'), (v_j, v_j')) : (v_i, v_j) \in E \wedge (v_i', v_j') \in E' \wedge \mathcal{L}((v_i, v_j)) = \mathcal{L}'((v_i', v_j'))\} \tag{2.17}$$

Definition 2.12. *If (u, v) is an edge in a* directed graph *G, then we may say that the edge (u, v) is* **incident from** *node u, and it is* **incident to** *the node v [47]. Conversely, if G is* undirected*, the edge (u, v) is simply said* **incident***, w.r.t. the considered node.*

Definition 2.13. (Neighbourhood): *Let $G = (V, E)$ be a (directed) graph, and $v \in V$ a node. There exists two functions δ^+, δ^- that defines the* **neighbourhood** *of a given node.*
In particular:

$$\delta^+ : V \to E; v \mapsto \delta^+(v) = \{(v, u) \in E\} \tag{2.18}$$

$$\delta^- : V \to E; v \mapsto \delta^-(v) = \{(u, v) \in E\} \tag{2.19}$$

Therefore, $\delta^+(v)$ *represents the set of* all *edges* **incident from** *the node* v*, while* $\delta^-(v)$ *is the set of* all *edges* **incident to** *the node* v*.*

Remark 2.11. *In case* v *is a node of an undirected graph* G*,* $\delta^+(v) = \delta^-(v) = \delta(v)$

Definition 2.14. (Degree of a Node) *[47]: The* **degree** *of a node* v *in an* undirected graph *is the total number of edges incident on it, i.e.,* $|\delta(v)|$*.*
Conversely, in case of directed graphs*, there exist the notions of* **in-degree** *(*$|\delta^-(v)|$*) and* **out-degree** *(*$|\delta^+(v)|$*), related to the number of edges that are incident* to *and* on *the node* v*, respectively.*

Remark 2.12. *A node whose* degree *(or* in/out-degree*) is equal to zero is* **isolated***.*

Remark 2.13. Maximal (In/Out) Degree of a graph*: The* **maximal in-degree** *and* **out-degree** *of a directed graph* $G = (V, E)$ *are defined as:*

$$\Delta^-(G) = \max\{|\delta^-(v)|, v \in' V\} \tag{2.20}$$

$$\Delta^+(G) = \max\{|\delta^+(v)|, v \in' V\} \tag{2.21}$$

Definition 2.15. Path*: Given a graph* G*, a* path p *in the graph* G *is defined as a sequence of nodes for which there exists an edge connecting to the consecutive node. Thus:* $p(v_i, v_j) = \langle v_i, \ldots, v_j \rangle$ *s.t.* $e_{k,k+1} = (v_k, v_k + 1) \in E, i \leq k \leq j$
The **length** *of the path is defined in terms of nodes that compose the path, and it is usually indicated with* l_p*.*
If all the nodes composing a path are distinct, the path is said to be **simple***.*

Definition 2.16. (Cycle) *[47]:*

(Directed graph)*: A path* $p = \langle v_o, \ldots, v_l \rangle$ *forms a* **cycle** *if* $v_0 = v_l$ *and the path contains at least one edge.*

(Undirected graph)*: A path* $p = \langle v_o, \ldots, v_l \rangle$ *forms a* **cycle** *if* $l_p \geq 3$ *and* $v_0 = v_l$*.*

In both cases, the cycle is **simple** *if, in addition, the nodes* $v_1, \ldots, v_l$ *in the path are distinct.*

Remark 2.14. *A cycle of length $l_p = 1$ is called a* **self-loop**

Definition 2.17. (Connected Graph) *[47]:*

(Directed graph)*: A directed graph G is* **strongly connected** *if every two nodes are reachable from each other.*

(Undirected graph)*: A undirected graph G is* **connected** *if every node is reachable from all other nodes.*

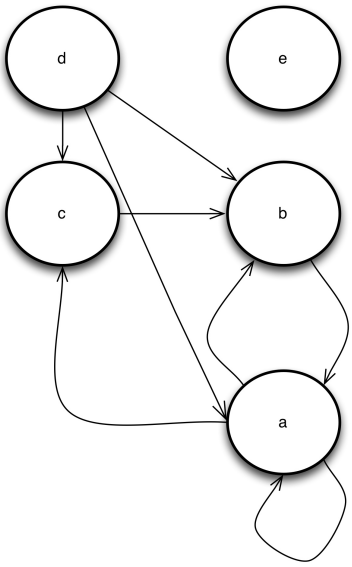

Figure 2.2: Example of a labelled directed graph

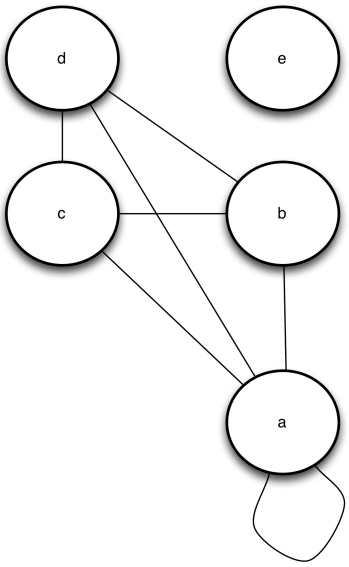

Figure 2.3: Example of a labelled undirected graph

Figures 2.2 and 2.3 represents two variants of the same labeled graph, namely directed and undirected respectively. In particular, in both graphs the node labeled with e is *isolated*, i.e., its (in/out-)degree is zero, and a *self-loop* is present on the node a.

Definition 2.18. Isomorphic Graphs *[47]:*
Two graphs $G = (V, E)$, and $G' = (V', E')$ are called **isomorphic** *if there exists a* bijection $f : v \to V'$ *such that* $(u, v) \in E$ *iff* $(f(u), f(v)) \in E'$.

In other words, two graphs G and G' are *isomorphic*, if and only if it could be possible to define a biunique correspondence between nodes in the two graphs such that nodes in G could be mapped to nodes in G'.

Definition 2.19. (Rooted Tree):
Let $G = (V, E)$ be a graph. G is a rooted tree iff:

a) *G is an undirected acyclic graph;*

b) $\exists!\ r \in V\ :$ *r has no incoming edges.* [*]
The node r is called the **root** *of the Tree.*

Proposition 2.2. *Let $G = (V, E)$ be an undirected graph. The following statements are equivalent [47]:*

1. *G is a Tree.*
2. *Any two nodes in G are connected by a* unique simple *path.*
3. *G is connected, but if any edge is removed from E, the resulting graph is disconnected.*
4. *G is connected, and $|E| = |V| - 1$.*
5. *G is* acyclic, *i.e., it does not contain cycles, and $|E| = |V| - 1$.*
6. *G is* acyclic, *but if any edge is added to E, the resulting graph contains a cycle.*

Proposition 2.3. Tree Properties and Terminology:
Let n be a node in a rooted Tree T with root r [47]:

- *Any node m on the* unique simple *path from r to n is called an* **ancestor** *of n.*
  Conversely, if m is an ancestor of n, then n is a **descendant** *of m.*
- *If the last edge on the unique simple path from r to n is (m, n), then m is the* **parent** *of n, and similarly n is a* **child** *of m.*
  In particular $\texttt{Ch}(m)$ *indicates the set of all the children nodes of the given node m.*

[*]The notation $\exists!$ stands for "There exists a unique"

- *The* **degree** *of a node n corresponds to the number of its children, i.e.,* $|\texttt{Ch}(n)|$
- *The unique node with no parent in T is the root r.*
- *Every node $l_i \in T$ with no children is a* **leaf** *or* **external node**. *A non-leaf node is called an* **internal node**.
- *The length of the unique simple path from the root node r and the node n is the* **depth** *of n in T.*
- *The* **height** *of a node n in T corresponds to the number of edges on the longest simple path from the node itself down to a leaf.*

This Section concludes with a formal definition of substructures, namely *subgraphs* and *subtree*, that will be frequently recalled in the description of *Kernel methods* for structured data (Section 2.4).

Definition 2.20. (Subgraph)*: Let $G = (V, E)$ be a graph. The graph $G' = (V', E')$ is a* **subgraph** *of G if $V' \subseteq V$ and $E' \subseteq E$.*

In other words, given a set $V' \subseteq V$, the subgraph of G **induced** by V' is the graph $G' = (V', E')$, where $E' = \{(u, v) \in E \ : \ u, v \in V'\}$ [47].

For example, considering the Figure 2.3, if we consider the set of nodes $V' = V \setminus \{e\}$, the **inducted subgraph** is $G' = (V', E')$, where $E' = E$ as the excluded node is isolated. It is worth noting that, in this case, the considered subgraph G' is *connected*.

Similarly, a **Subtree** T' of a tree T, could be defined as:

Definition 2.21. (Subtree)*:*
Let $T = (V, E)$ be a tree. $T' = (V', E')$ is a **subtree** *of the tree T iff $V' \subseteq V$ and $E' \subseteq E$, where $E' = \{(u, v) \in E \ : \ u, v \in V'\}$*

Moreover, for tree structures, the following definition may be applied:

Definition 2.22. (Subset Tree)*:*
Let $T = (V, E)$ be a tree. $T' = (V', E')$ is a **subset tree** *of the tree T iff:*

1. T' *is a subtree of* T.

2. $\forall\ n \in V'$*: only one of the two following conditions may hold:*

   (a) $\texttt{Ch}(n) \cap V' = \texttt{Ch}(n)$;

   (b) $\texttt{Ch}(n) \cap V' = \emptyset$

In other words, a *Subset tree* is a subtree with an additional constraint that imposes to every node in the subtree that either all or none of its children must belong to the subtree.

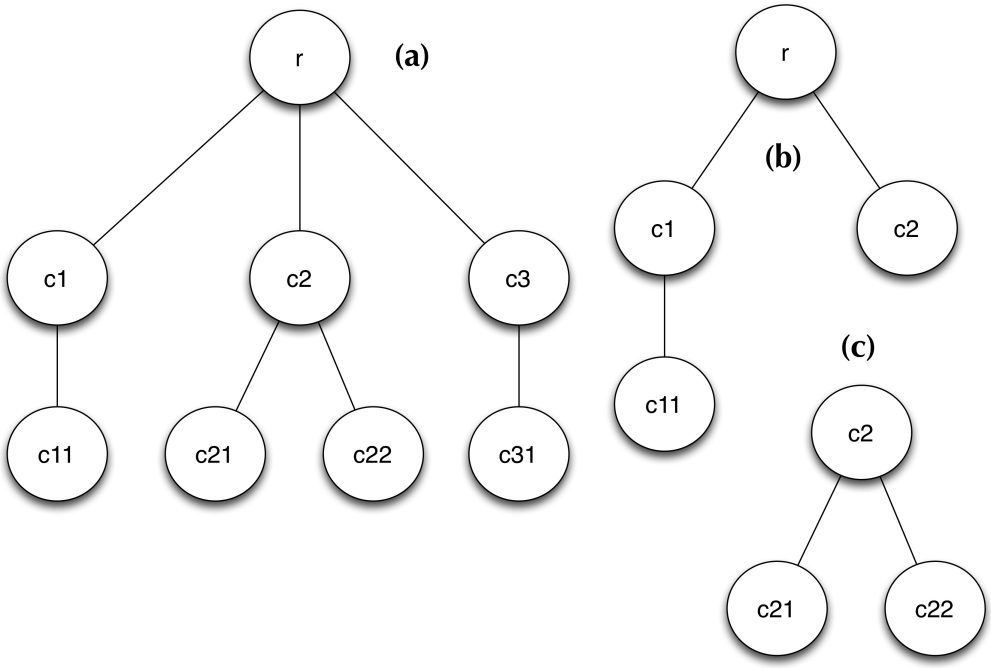

Figure 2.4: Example of a Tree (a), and two corresponding possible subtrees (b) and (c). In particular, (b) is a Subtree, while (c) is a Subset tree.

2.2 Learning from examples

Although ML systems may be classified according to different view points [38], a common choice is to classify ML approaches according to the *underlying learning*

strategy [35]. In particular, in every learning situation, the learner transforms information provided by the environment into some new form in which it is stored for future use [9, 142]. The nature of this transformation determines the type of learning strategy used. Several basic strategies have been distinguished: *rote learning*, *learning by instruction*, *learning by deduction*, *learning by analogy*, and *learning by induction* [9]. The latter is further distinguished in *learning by observation* and *learning from examples* [9, 38].

For the sake of brevity, only the latter strategy will be highlighted in the remainder of this Section, as it is the one closely related to the topics presented in this thesis. For a complete description of the different learning strategies and their corresponding features, the interested reader may refer to [9, 35, 38, 142].

Learning from examples is one of the most popular and widely employed strategy in ML approaches that is often simply called *learning* [35]. Similarly, the term "example" is usually treated as a synonym for "data". Thus, from now on, the two terms will be used interchangeably.

The *learning problem* that this strategy involves can be described as "finding a general rule that explains the data give only a sample of limited size" [35]. In particular, this strategy comprises a set of different learning techniques that are distinguished in three big families: *Supervised Learning*, *Reinforcement Learning*, and *Unsupervised Learning* [134].

2.2.1 Supervised Learning

In supervised learning, data are represented as tuples in the form of $\langle input, output \rangle$ patterns. This problem is called supervised learning because the objects under considerations are already associated with the target values [35]. In more details, in the problem of supervised learning, a so-called *training set* of examples with the correct responses (*targets*) are provided and, based on such training set, the learning algorithm generalises to respond correctly to all possible inputs [134].

More formally, the training set is usually written as a set of pairs $(x_i, \mathbf{t_i})$, where the inputs are x_i, the targets are $\mathbf{t_i}$*, and the i index suggests that there are lots of pairs, ranging from 1 to an upper limit $N \in \mathbb{N}$ [134].

*The boldface denotes a vector of values.

Typical examples of supervised learning approaches are the *Classification* and the *Regression*, which are distinguished according to the type of the outputs.

Classification : In the *classification* learning problem, the output space is composed by a finite number of discrete *classes*, and the corresponding learning algorithm is called the *classifier*. In particular, the learning problem is to assign to each input data its corresponding class or set of classes.

Regression : If the output space of the learning problem corresponds to values of continuous variables, then the learning task is known as the problem of *regression*. Typical examples of regression include predicting the value of shares in the stock exchange market, or estimating the value of a physical measure in a section of a thermoelectric plant [35].

2.2.2 Reinforcement Learning

Reinforcement learning correspond to a strategy that lies between supervised and unsupervised learning [134]. In fact, the algorithm get told when the output is wrong, but does not know how to correct it [134]. Thus, the algorithm tries to explore different possibilities until the final correct output has been discovered.

In other words, the problem of reinforcement learning is to learn what to do, i.e., how to map situations to actions in order to maximise a given *reward* [35]. Such maximisation is what guide the learning algorithm through the different possible solutions. A comprehensive survey of reinforcement learning can be found in [171]

2.3 Unsupervised Learning

If the input data to the learning algorithm comprise a set of samples without associated target values, the problem is classified as *unsupervised learning*. In unsupervised learning, data do not contain any indication to the correct target, instead the algorithm tries to identify *similarities* between the inputs so that inputs that are in some way related are *categorised* together [134].

Duda et al. in [59] indicate some of the advantages of unsupervised learning techniques with respect to supervised ones:

- There is no cost of collecting and labelling samples.
- Unsupervised techniques may be used to identify characteristics of the samples which are useful for differentiating between them.
- Unsupervised techniques may be used for exploring the data analysing its structure.

Clustering algorithms represent a rich subclass of unsupervised learning techniques [35].

Nevertheless, even if the difference between supervised and unsupervised techniques apparently concerns the sole input examples (labelled and unlabelled respectively), their corresponding learning strategies lead to conceptually different problems. For example, let us consider classification and clustering algorithms as for representatives of the two learning strategies. On the one hand, classification algorithms perform a two-step learning, namely the *training* and the *test*, aiming at identifying similarities between inputs that belong to the same class. On the other hand, clustering algorithms aim at identify similarities by directly loooking at the data in order to discover *patterns* and *relationships* among objects [167]. In particular, the main goal of clustering algorithm is to create groups of input data that are coherent internally, but clearly different from each other [130].

In the literature several clustering algorithms have been proposed [29], that have been classified according to different view points. Clustering techniques are generally classified as *partitional clustering* (Section 2.3.2) and *hierarchical clustering* (Section 2.3.3), based on the properties of the generated clusters [188]. Partitional clustering directly divides data points into some pre-specified number of clusters without any structure. As a consequence, partitional clustering algorithms are sometimes referred as *flat clustering* [130].
On the other hand, hierarchical clustering groups data with a sequence of nested partitions, either from singleton clusters to a clusters including all individuals (the so-called *bottom-up strategy* [130]), or vice versa (*top-down strategy* [130]) [188].

Another typical criterion for the classification of clustering algorithms considers the *nature* of the learning problem. Clustering strategies that allow a single object to belong to more than one clusters, give rise to the so-called *soft-clustering* problem [130]. Conversely, the *hard clustering* problem constrains objects to belong

to exactly one cluster. The interested reader could find many other classification criteria for clustering approaches in [72, 90, 91, 188].

2.3.1 Clustering Terminology

Roughly speaking, by data clustering, we mean that for a given set of *data points* and a *similarity* measure, such data are regrouped in a way that points in the same group (i.e., *cluster*) are similar and points in different groups are dissimilar [72]. Thus, the clustering analysis relies on such three main concepts. However, in the literature of data clustering, different terms may be used to express the same thing [72]. To this aim, in the remainder of this section, a brief description of such concepts will be provided, in order to clarify the terminology used in the following sections.

The terms *data point*, *object* [72], *item* [188], and *pattern* [91] are used to denote a single element in the data set. Moreover, for data points in a high-dimensional space, terms such as *attributes* [91] or *features* [68, 72, 134] are used to indicate the different *scalar components* characterising the data points [90]. More formally:

Definition 2.23. (Data) *[72]:*
Let X be a finite set, representing the set of **input data points**, *or simply* **data**, *s.t. $X = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$, where $\mathbf{x_i} = (x_{i1}, x_{i2}, \ldots, x_{if})^T$ is a vector denoting the* i*th object $\mathbf{x}_i \in X$, described by a set of f different* features.

Distances and similarities play an important role in clustering analysis [72], as they guide the learning strategy to generate groups of data. As a matter of fact, every clustering algorithm is based on the index of similarity (or dissimilarity) between pairs of data points [91].
The two most employed measures are the *Euclidean Distance* (Definition 2.7) and the so-called *cosine similarity* [130]. However, in the literature several distance and similarity functions have been proposed, which are employed according to the specific nature of the analysed data [72].

In the literature, the terms *group*, and *cluster* are typically used interchangeably and in an essential intuitive manner [72]. Nevertheless, the common sense of the term *cluster* combines various plausible criteria and require that all objects in a cluster satisfy the aforementioned properties imposed by clustering analysis [72].

For example, objects in the same cluster must share the same or closely related properties, and they must be clearly distinguishable from the rest of the objects in the data set [72, 90].

2.3.2 Partitional Clustering

The problem of partitional clustering can be formally stated as follows:

Definition 2.24. (Partitional Clustering) *[91]: Given N data points $\mathbf{x_i} \in X^d$, where X^d is a* d-dimensional *metric space (see Definition 2.7).*
Partitional clustering algorithms intend to determine a partition of the input data into a set of k clusters, $\{C_1, \ldots, C_k\}$, which optimises a given criterion.

To date, one of the most commonly adopted strategy relies on the *minimisation* of the *square-error* [91].

In more details, let X be a data set with N data points, and let $C_1, \ldots, C_k$ be the k disjoint clusters of X (i.e., *hard clustering*). Then, the error function is defined as [188]:

$$E = \sum_{i=1}^{k} \sum_{\mathbf{x_j} \in C_i} \rho(\mathbf{x_j}, \mu(C_i)) \tag{2.22}$$

where $\mu(C_i)$ represents the *centroid* of cluster C_i. $\rho(\mathbf{x}, \mu(C_i))$ denotes the distance between the input data $\mathbf{x}$ and $\mu(C_i)$ (see Definition 2.7).

As a matter of fact, such optimisation criterion properly characterises a family of partitional clustering algorithms usually called *centroid-based* algorithms [91].

The basic idea of these family of algorithms is to start with an *initial partition* and assign data to clusters with respect to their corresponding centroids, so as to reduce the square-error [91]. In particular, the general algorithmic framework of these techniques is reported below [91]:

> **Step 1.** Select an initial partition with k clusters. Repeat steps 2 through 5 until the cluster membership stabilises.
>
> **Step 2.** Generate a new partition by assigning each data point to its closest cluster centre.
>
> **Step 3.** Compute new cluster centres as the centroids of the clusters.

Step 4. Repeat steps 2 and 3 until an optimum value of the criterion function is found.

Step 5. Adjust the number of clusters by merging and/ splitting existing clusters, or by removing the outliers.

Nevertheless, different implementations of some crucial decisions such as the choice of the *initial partition*, the *update strategy* and the *halting criterion* lead to different clustering strategies [90].

The most famous centroid-based partitional clustering is certainly the k-means algorithm, that tries to minimise the *total within-cluster variance*. In other words, recalling the Equation 2.22, the optimisation criterion is defined as:

$$E_{kmeans} = \sum_{i=1}^{k} \sum_{\mathbf{x_j} \in C_i} (\mathbf{x_j} - \mu_j)^2 \tag{2.23}$$

The algorithmic description of the k-means clustering is reported in Algorithm 1.

The algorithm starts by initialising the centroids (Line 2) by picking a set of k different random points in the input space (Algorithms 2). Then the algorithm continues by assigning each item in the data set to the closest centroid, according to the used distance measure (Line 9). Afterwards, the new clusters configuration is computed, and corresponding centres are updated accordingly (see Algorithm 3). The iteration ends when clusters' centres stop moving or a maximum number of iterations have been performed (Line 16).

The asymptotic computational complexity of the k-means algorithm is $O(NkT)$, where N is the total number of data points, k the number of clusters and T the number of iterations.

An example clustering result of the k-means algorithm is reported in Figure 2.1

One possible variant of the k-means algorithm is the so-called k-medoids [97]. Differently from classical k-means, the k-medoids applies different initialisation and update strategies: instead of selecting random points in the input space as centroids, this algorithm select random points in the data set, called *medoids*, that

Algorithm 1 The k-Means Algorithm

Input: $\mathtt{k}$: The total number of partitions to generate.
Input: $\mathtt{X}$: The input data set, namely $\mathtt{X} = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$
Input: $\mathbb{T}$: Total number of Iterations allowed.
Output: $\mathtt{C}$: The set of the generated k clusters, namely $\mathtt{C} = \{\mathtt{C}_1, \ldots, \mathtt{C}_k\}$

```
1: function k-Means(k, X)
2:     μ ← InitialiseCentroids(k)                          ▷ see Algorithm 2
3:     C ← {∅_1, ..., ∅_k}                                 ▷ Initialisation
4:     iterCount ← 0
5:     repeat                                              ▷ Learning
6:         μ' ← μ
7:         for each: x_i ∈ X do
8:             for each: μ_j ∈ μ' do
9:                 d_ij ← ρ(x_i, μ_j)        ▷ Calculate the distance ρ to each cluster centre
10:            end for
11:            l ← argmin_{1≤j≤k} d_ij
12:            C_l ← C_l ∪ {x_i}              ▷ Assign the data to the closest cluster centre
13:        end for
14:        μ ← UpdateCentroids(C)                          ▷ see Algorithm 3
15:        iterCount ← iterCount + 1
16:    until μ' = μ ∨ iterCount = T
17:    return C
18: end function
```

Algorithm 2 k-Means Initialisation Strategy

Input: $\mathtt{k}$: The total number of partitions.
Output: μ : The set of k different centroids

```
1: function InitialiseCentroids(k)
2:     μ ← ∅
3:     for i ← 1 to k do
4:         μ_i ← choose a random position in the input space
5:         μ ← μ ∪ {μ_i}
6:     end for
7:     return μ
8: end function
```

are considered as representatives for each generated clusters. Therefore, the whole partitioning strategy is performed with respect to the selected medoids, that are changed in the update step, selecting for each clusters the new elements that lead to the configuration with the lowest cost.

The main advantage of the k-medoid algorithm over the k-means is that it is

Algorithm 3 k-Means Update Strategy

Input: $\mathtt{C}$: Partitions generated so far during clustering.
Output: μ : The set of k different centroids

1: **function** UpdateCentroids($\mathtt{C}$)
2: $\quad \mu \leftarrow \emptyset$
3: $\quad$ **for** $i \leftarrow 1$ to k **do**
4: $\quad\quad \mathtt{N_i} \leftarrow |\mathtt{C_i}|$
5: $\quad\quad \mu_\mathtt{i} \leftarrow \frac{1}{\mathtt{N_i}} \sum_{j=1}^{\mathtt{N_i}} \mathbf{x_j}$
6: $\quad\quad \mu \leftarrow \mathtt{mu} \cup \{\mu_\mathtt{i}\}$
7: $\quad$ **end for**
8: $\quad$ **return** μ
9: **end function**

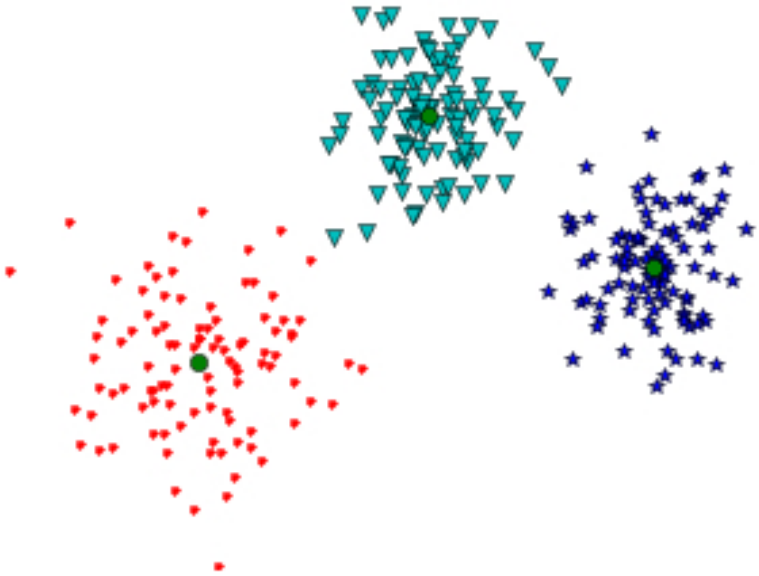

Figure 2.1: An example of *k*-means clustering of 2D points organised in three clusters. Cluster centroids are marked as large green rings, while elements in the different clusters are dots, triangles, and stars respectively.

more robust to noise in the data and to *outliers* [97].

2.3.3 Hierarchical Clustering

Differently from partitional "flat" clustering, *hierarchical clustering* algorithms impose a structure on data. Indeed, instead of generate different *disjoint* partitions of input data, these algorithms organise data into a sequence of *nested partitions* [90]. More formally:

Definition 2.25. (Nested Partition)*:*
Let $X = \{\mathbf{x_1}, \ldots, \mathbf{x_N}\}$ *be the set of* N *input data, and* $C = \{C_1, \ldots, C_M\}$ *be the clustering partition of* X*. A partition* B *is said to be* **nested** *in* C *iff.* $\forall\ B_i \in B, B_i, \exists C_j \in C\ :\ B_i \subseteq C_j$

In other words, the partition B is *nested* in another partition C if and only if every component of the former is a subset of a component of the latter.

Therefore, a hierarchical clustering is a sequence of partitions in which each partition is nested into the next partition in the sequence [91]. In particular, the so-called *agglomerative* hierarchical clustering (HAC) [72, 130] algorithms start with disjoint singleton clusters, one for each element of the data set X, and continue by repeatedly nesting the different partitions until a single cluster containing all the N elements remains. On the other hand, the so-called *divisive* hierarchical clustering algorithms perform the same task in reverse order [91].

In the following, the description will be limited only to HAC algorithms as they are closely related to the research contributions presented in the following Chapters.

Hierarchical clustering algorithms are usually represented by means of a tree structure, called *dendrogram* [72, 188]:

Definition 2.26. (Dendrogram) *[72]:*
A dendrogram T *is a tree in which each* internal node *is associated with a* height *satisfying the following condition:*

$$height(A) \leq height(B) \leftrightarrow A \subseteq B$$

for all possible subset of data points A *and* B *if* $A \cap B = \emptyset$

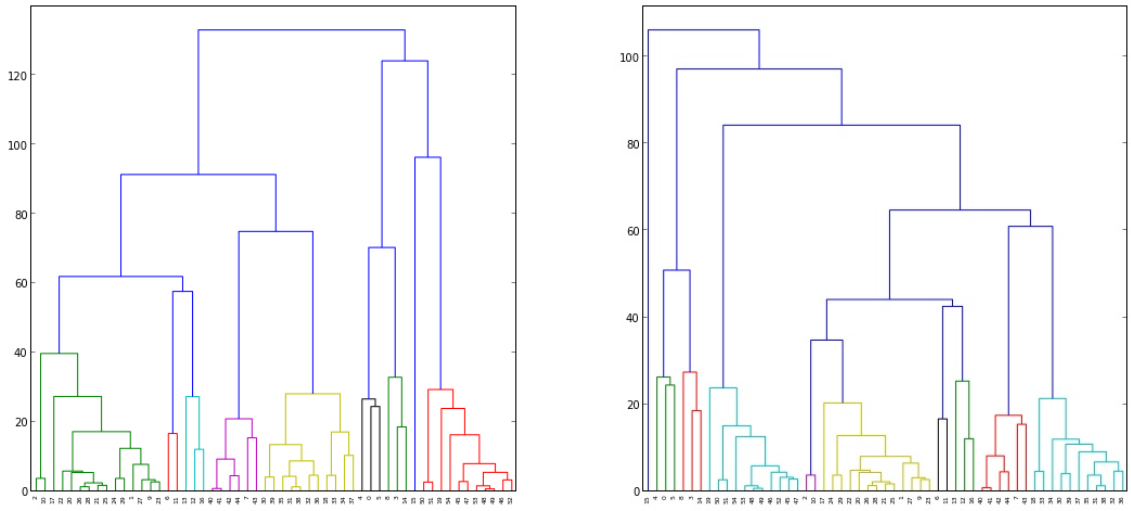

Figure 2.2: Two examples of dendrograms representing the clustering results of the same set of data applying two different *linkage* strategy, namely the *complete linkage* (left) and the *group average linkage* (right).

Two examples of dendrograms are represented in Figure 2.2.

The algorithmic description of the HAC method is reported in Algorithm 4. The algorithm takes in input the data set X and the so-called *proximity matrix* [72, 188], which is a triangular matrix containing the distances between each pair of distinct points in X. As already mentioned, the clustering procedure starts with a set of singleton clusters (Line 2), that will be iteratively updated: at each step, the algorithm looks for the two closest clusters to be joined. Afterwards, the two matched clusters are merged and the similarities between all other clusters are updated accordingly. Indeed, the similarity update strategy is actually applied by invoking the *Linkage* function (Line 20), which represents one of the most important aspects of HAC algorithms. In more details, the so-called *linkage method* [130] defines the strategy to apply in updating the similarities between the new cluster and remaining ones: different linkage strategies lead to different HAC algorithms.

In the literature, several linkage methods have been proposed [148]. The most important and widely used ones are described below, according to the parametric formulation provided by Lance and Williams [110].

Proposition 2.4. Linkage Rule Parametric Formulation *Let C be the set of clus-*

ters, let i, j be the indexes of the two clusters already joined, and let k be the index of another cluster C_k. Thus:

$$sim(i+j,k) = a(i)sim(i,k) + a(j)sim(j,k) + bsim(i,j) + c|sim(i,k) - sim(j,k)| \tag{2.24}$$

where the values of parameters a, b, and c vary depending on the specific linkage method *to apply.*

Table 2.1 summarises the different values of such parameters, according to the

Algorithm 4 HAC algorithm

Input: $\mathtt{X}$: The input data set, namely $\mathtt{X} = \{\mathbf{x_1}, \ldots, \mathbf{x_N}\}$
Input: $\mathtt{M}$: The *proximity* Matrix, namely a triangular matrix containing the distances among all the data in $\mathtt{X}$
Output: $\mathtt{P}$: The list of nested partitions.

```
1: function HAC(k)
2:     P ← {{x_i}, ..., {x_N}}          ▷ Initialise partitions with singleton clusters.
3:     while |P| > 1 do                  ▷ Look for the closest clusters
4:         lowestPair ← (1, 2)
5:         closest ← M(1, 2)
6:         for i ← 1 to N do
7:             for j ← i + 1 to N do
8:                 d ← M(i, j)
9:                 if d < closest then
10:                    lowestPair ← (i, j)
11:                    closest ← d
12:                end if
13:            end for
14:        end for                       ▷ Join Clusters and Update distances
15:        left, right ← lowestPair
16:        leftCluster ← P_left
17:        P ← P \ P_left
18:        rightCluster ← P_right
19:        P ← P \ P_right
20:        newCluster ← Linkage( P_left, P_right, M)
21:        P ← P ∪ newCluster
22:    end while
23:    return P
24: end function
```

Linkage Strategy	Parameter Values
Single linkage	$a(i) = a(i) = 0.5$ $b = 0$ $c = -0.5$ (**shortly**: $max\{sim(i,k), sim(j,k)\}$)
Complete linkage	$a(j) = a(i) = 0.5$ $b = 0$ $c = 0.5$ (**shortly**: $min\{sim(i,k), sim(j,k)\}$)
Group average linkage	$a(i) = \|i\|/(\|i\| + \|j\|)$ $a(j) = \|j\|/(\|i\| + \|j\|)$ $b = 0$ $c = 0$

Table 2.1: Linkage update strategies, according to the parametric formulation proposed by Lance and Williams [110]

particular linkage strategy:

The computational complexity of the HAC algorithm (Algorithm 4) is $\Theta(N^3)$ because the function exhaustively scan the $N \times N$ `M` for the closest clusters in each of the $N - 1$ iterations.

Finally, it is worth mentioning that the two dendrograms shown in Figure 2.2 correspond to the clustering results obtained applying two different linkage methods, namely complete and group average linkage, to the same set of input data. The two resulting partitions are almost different, thus confirming the key importance of linkage methods for HAC algorithms.

2.4 Kernel Methods

The notion of *similarity* is crucial for the definition and the application of machine learning techniques. In particular, this consideration holds regardless the underlying learning strategy applied, e.g., supervised or unsupervised learning.

Roughly speaking, the most intuitive way to characterise the similarity computation between pairs of objects could be simply to count the number of their common features, and actually this is not very different from what is actually done.

Nevertheless, more formal definitions are required in order to integrate a proper

notion of similarity into the computational framework of ML techniques. Thus, let X be the (usual) d-dimensional set of input data as formalised in Definition 2.23. Moreover, let us consider a *similarity measure* of the form

$$k : X \times X \rightarrow \mathbb{R}; (\mathbf{x}, \mathbf{x}^{'}) \mapsto k(\mathbf{x}, \mathbf{x}^{'}) \tag{2.25}$$

that is a function that, given two patterns $\mathbf{x}$ and $\mathbf{x}^{'}$, returns a real number characterising their similarity [166].
Without loss of generality, let us assume that k is *symmetric*, that is, $k(\mathbf{x}, \mathbf{x}^{'}) = k(\mathbf{x}^{'}, \mathbf{x}), \forall\ \mathbf{x}, \mathbf{x}^{'} \in X$. For reasons that will become clearer later, the function k is called a *kernel* [27, 166, 167].

A very straightforward and simple type of similarity measure that is of particular mathematical appeal is the *dot product* (Definition 2.4). However, this measure strongly relies on a vectorial representation of the input data. In other words, the underlying assumption necessary to employ the dot product as similarity measure is that input data must exist in a dot product space (Definition 2.2). Unfortunately, this assumption is not robust enough for the general case: for example, structured data, such as trees or graphs, do not have an *explicit* vectorial representation, so the learning algorithm may not leverage the dot product computation to compare them.

As a result, to tackle this kind of issues, *kernel* functions, or simply *kernels*, have been proposed to make computations [8], and applied in combination with learning algorithms, i.e., *Support Vector Machines* (SVM) [48], that use kernels to make inner products between data vectors. To date a number of learning algorithms, called *Kernel Methods*, have been proposed in the literature [147, 166, 167].

As a matter of fact, the class of Kernel Methods comprises all those (learning) algorithms that do not require an explicit representation of the examples but only information about the similarities among them.

Any Kernel Method can be decoupled into two different components:

1. A problem-specific kernel function
2. A general purpose learning algorithm.

In particular, kernel functions have some interesting (mathematical) features:

- The space of kernel function is closed under operations such as addition and multiplication, thus allowing the composition of different kernel functions to remain a kernel.

- Considering a finite set of input data of n objects, kernel functions are represented by a $n \times n$ matrix, regardless the size of each individual object. This property could be very useful when a small dataset of large size objects (in terms of *features*) has to be analysed.

In the following Sections, some mathematical background and properties of kernels will be described (Section 2.4.1), together with a general overview of kernel methods defined for structured data (Section 2.4.2).

2.4.1 Dual Representation

As briefly emphasised above, the dot product approach is not really sufficiently general to deal with many interesting problems [87, 166].

However, in order to be able to use the dot product as a similarity measure, we first need to represent the input data as vectors in some dot product space $\mathcal{H}$ (which need to coincide to $\mathbb{R}^N$ [166]).

Therefore, let $\mathcal{X} = \{x_1, \ldots, x_m\}$ be a metric space (Definition 2.7). We may define:

Definition 2.27. (Mapping Function)*:*

$$\Phi : \mathcal{X} \to \mathcal{H}; x \mapsto \mathbf{x} \tag{2.26}$$

That is a function that maps *a given input object to its* vectorial *representation.*

In particular, the space $\mathcal{H}$ is called the *feature space* [166, 167]. To summarise, embedding the data into $\mathcal{H}$ by Φ has the following advantages [166]:

- It lets us defined a similarity measure from the dot product in $\mathcal{H}$, such that

$$k(x, x^{'}) = \langle \mathbf{x}, \mathbf{x}^{'} \rangle = \langle \Phi(x), \Phi(x^{'}) \rangle \tag{2.27}$$

- No assumption nor limitation has been imposed in the definition of the mapping function. So this freedom allows to choose mappings that may be used in a large variety of cases.

Before formally defininig what a kernel function actually is, some preliminary definitions are required.

Definition 2.28. (Positive definite matrix)*: An $n \times n$ matrix $A = (a_{ij}), a_{ij} \in \mathbb{R}$ is called a* **positive definite matrix** *iff:*

$$\sum i = 1^n \sum_{j=1}^{n} c_i c_j a_{ij} \geq 0 \tag{2.28}$$

for all $n \in \mathbb{N}, c_1, \ldots, c_n \in \mathbb{R}$

The basic property of positive definiteness are confirmed by the following results.

Proposition 2.5. *Let A be a matrix. The following two statements are equivalent:*

- *A is positive definite iff A is symmetric.*
- *A is positive definite iff A has all eigenvalues ≥ 0*

Definition 2.29. Mercer Kernel*:*
Let $\mathcal{X}$ be a non empty set. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a **Positive Definite Kernel** *(or* **Mercer Kernel***) iff:*

$$\sum_{i=1}^{N} \sum_{j=1}^{n} c_i c_j k(x_i, x_j) \geq 0 \tag{2.29}$$

for all $n \in \mathbb{N}, x_1, \ldots, x_n \in \mathcal{X}, c_1, \ldots, c_n \in \mathbb{R}$

In the following, some important results underlying the basic properties of positive definite matrices are reported. For the sake of brevity, the corresponding proofs will be omitted. Further details are provided in [166] and [167].

Proposition 2.6. *Let k be a kernel on $\mathcal{X} \times \mathcal{X}$. The following properties are equivalent:*

- *k is positive definite iff k is symmetric.*
- *k is positive definitive iff for every finite subset $\mathcal{X}_0 \subseteq \mathcal{X}$, the restriction of k to $\mathcal{X}_0 \times \mathcal{X}_0$, namely $k^{[\mathcal{X}_0 \times \mathcal{X}_0]}$, is positive definite.*

Moreover, if k is positive definite, then $k(x,x) \geq 0 \forall\ x \in X$.

Remark 2.15. *The* inner product *is a* Positive Definite Kernel*, namely it is a* Mercer Kernel.

Proposition 2.7. (Cauchy-Schwarz's Inequality)*: For any* Positive Definite Kernel *k, the Cauchy-Schwarz's inequality holds:*

$$|k(x, x^{'})|^2 \leq k(x,x)k(x^{'}, x^{'}) \tag{2.30}$$

The following result, provided by Mercer, allows the use of kernel functions to **make dot products**.

Proposition 2.8. *Let K be a symmetric function such that:*

$$\forall\ x, y \in \mathcal{X} \subseteq \mathbb{R}, K(x,y) = \langle \Phi(x), \Phi(y) \rangle \tag{2.31}$$

where $\Phi : \mathcal{X} \to \mathcal{H}$, and $\mathcal{H}$ is the (dot product) **feature space**.
K can be represented in terms of Equation 2.31 iff

$$K = (K(x_i, x_j))_{i,j=1}^{N} \tag{2.32}$$

is semidefinite positive, namely K is a Mercer Kernel.
Moreover, K defines an **explicit** *mapping if Φ is known in advance, otherwise the mapping is* **implicit**.

Equation 2.32 defines a kernel function in terms of its corresponding matrix representation, namely the so-called *Gram matrix*. In fact, some alternative definitions of kernel functions in terms of such Gram matrix are also provided [167]:

Definition 2.30. (Gram Matrix)*:*
The Gram *matrix G^K related to kernel K with respect to the input data set $\mathcal{X}$ is defined as:*

$$G_{i,j}^{K} = K(x_i, x_j) \tag{2.33}$$

Remark 2.16. *K is a* valid *kernel function if and only if it is symmetric and positive semidefinite, namely if* any *of its* Gram *matrices are symmetric and positive semidefinite.*
In particular, a matrix is symmetric *iff.* $\forall\ i, j,\ K(x_i, x_j) = K(x_j, x_i)$. *Besides, it is* semidefinite positive *if all of its eigenvectors are nonnegative.*

In conclusion, it is worth repeating that kernel functions are closed under operations such as the addition and multiplication. Therefore, the sum, the product or a linear combination (Definition 2.3) of kernels, still produces a valid kernel. More formally:

Proposition 2.9. *Let* $K_1, K_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ *be two kernel functions on a metric space* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. *Then, the following properties holds [167]:*

1. (Additive Property)
   $K^+(x, x') = K_1(x, x') + K_2(x, x')$ *is a valid kernel.*

2. (Multiplicative Property)
   $K^*(x, x') = K_1(x, x')K_2(x, x')$ *is a valid kernel.*

2.4.2 Kernels for Structured Data

Kernel methods have been widely adopted in many machine learning techniques [87] thanks to their flexibility in providing an efficient and general mechanism to compute the similarity between objects (Proposition 2.8). As a matter of fact, to date an increasingly series of kernels that perform efficient comparisons between *structured data* have been proposed [167].

An extensive survey of kernel methods for structured data is provided in [75] and in [74].

By structured data, we intend data that are formed by combining simpler components into more complex items, frequently involving a recursive use of simpler objects of the same type [167]. Examples of structured data include complex (discrete) objects such as *trees* and *graphs* [167].

Convolution Kernels represent a general methodology for computing kernels on complex discrete objects [84].
The basic idea is that a complex object may be split into parts and their similarity

calculation may be expressed in terms of their constituent subparts. Thus, assuming to have at disposal a positive semidefinite kernel on the parts, Convolution Kernels describe a way for preserving the positive semidefiniteness of functions by a sum of kernels on the parts (Proposition 2.9).

Definition 2.31. (Convolution Kernel)*:*
Let $\mathcal{X}, \mathcal{X}_1, \ldots, \mathcal{X}_D$ be a $D+1$ non empty separable metric spaces, $x \in \mathcal{X}$ a structure, and $\mathbf{x} = (x^1, x^2, \ldots, x^D)$ the parts of x.
A relation $R : \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_D \times \mathcal{X}$ where $R(\mathbf{x}, x)$ is true iff $(x^1, x^2, \ldots, x^D)$ are the parts of x.
Moreover let $R^*(x)$ be the set of all the subparts of x. Then the **Convolution Kernel** *can be expressed as:*

$$k(x_i, x_j) = \sum_{\mathbf{x_i} \in R^*(x_i)} \sum_{\mathbf{x_j} \in R^*(x_j)} \prod_{d=1}^{D} k_d(x_i^d, x_j^d) \tag{2.34}$$

where the different k_d are kernels defined on the substructures.

Remark 2.17. *Haussler in [84] provide a complete proof which demonstrates that, if the k_d are positive semidefinite, the kernel k in Equation 2.34 is also positive semidefinite, and thus a kernel itself (Proposition 2.8).*

Remark 2.18. R-convolution *The relationship R is called the* R-convolution *relation [84].*
In particular, given $R : \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_D \times \mathcal{X}$ such that $R(\mathbf{x}, x) \mapsto$ true iff. $\mathbf{x}$ is a valid decomposition of x, it is always possible to define the reverse *relationship, namely $R^{-1}(x) = \{\mathbf{x} | R(\mathbf{x}, x) = true\}$*

Convolution Kernels have been successfully applied to a variety of problems involving structured data, thus appearing to be a valid strategy of dealing with structured data. In particular, in the following sections, a brief review of the main contributions for tree and graph structured data is provided.

Tree Kernels

Tree Kernels have been widely used where the information is represented by means of tree-based structures, like Natural Language Processing [41, 144, 146] and Bioin-

formatics [180], where they have been applied to Parse- and Phylogenetic-trees, respectively.

Vishwanathan and Smola describe in [181] a fast kernel that is applicable to strings[†] and trees. In particular, when applied to tree structures, the proposed kernel consider *proper subtrees* (Definition 2.21) of the input tree.

The kernel applies a weighted sum of the number of common subtrees, based on the assumption that the total number of matching subtrees is small if compared to the size of the feature space.
More formally, the proposed kernel is defined as:

$$K(T_i, T_j) = \sum_{t \in T_i} \sum_{u \in T_j} m(t, s) w_t$$

where t, u are subtrees of the input trees T_i and T_j, respectively. Moreover, w_t is the weight associated to the subtree t and $m(\cdot, \cdot)$ is a Boolean (filter) function defined as:

$$m : \mathcal{T} \times \mathcal{T} \to \{0, 1\}$$

Such function returns 1 whether the two input (sub)trees are identical, 0 otherwise.

One limitation of this kernel for tree structures is that it is not able to measure a similarity function based on the common subset trees (Definition 2.22). A kernel for trees, based on counting *matching subset trees*, has been proposed by Collins and Duffy in [41].

Let T be a tree, and $\mathcal{T} = \{T_1, \ldots, T_n\}$ a set of input trees in which m different subset trees are present. Thus each feature, i.e., subset trees, can be indexed by an integer number $1 \leq k \leq m$. Then, the function $h_k(T_j)$ counts the number of times the subset tree indexed by i occurs in the tree T_j. Thus each tree $T_j \in \mathcal{T}$ is represented as: $\Phi(T_j) = [h_1(T_j), h_2(T_j), \ldots, h_m(T_j)]$.

Therefore, the final *Subset Tree Kernel* (SST) is defined as:

$$K(T_i, T_j) = \langle \Phi(T_i), \Phi(T_j) \rangle = \sum_{k=1}^{m} h_k(T_i) h_k(T_j) \tag{2.35}$$

where SST defines a similarity measure between trees which corresponds to the

[†]Strings are considered structured data as well [167]

number of shared subset trees.

Nevertheless, while the definition of the kernel function is quite straightforward, the interesting part concerns the definition of a procedure to efficiently calculate the number of subset trees. To this aim, authors propose a recursive algorithm that is based on a function $I_k(n)$, called the *Indicator function*, which returns 1 if the subset tree indexed by k is rooted at node n, 0 otherwise.
Thus, indicating con N_i the set of nodes of the tree $T_i \in \mathcal{T}$, it follows that $h_k(T_i) = \sum_{n \in N_i} I_k(n)$. Therefore, the SST Kernel in Equation 2.35 can be rewritten as:

$$K(T_i, T_j) = \sum_{k=1}^{m} h_k(T_i) h_k(T_j) \tag{2.36}$$

$$= \sum_{k=1}^{m} \sum_{n \in N_i} I_k(n) \sum_{m \in N_j} I_k(m) \tag{2.37}$$

$$= \sum_{n \in N_i} \sum_{m \in N_j} \sum_{k=1}^{m} I_k(n) I_k(m) \tag{2.38}$$

In the worst case, the overall computational complexity of the SST Kernel is $O(|N_i||N_j|)$. However, authors discussed that this estimation represents an upper-bound complexity, since a more accurate analysis shows that the actual complexity depends on the number of matching subset trees. In particular, considering the case of natural language parsing trees [41], for which SST have been originally defined, the recursive computation may be bound by avoiding the comparison of subset trees rooted in different nodes.

This observation has resulted in the *Fast Tree Kernel algorithm* [144], which has the same worst case complexity but in practical applications may provide a relevant speed up.

In addition, Moschitti in [145] provides a strategy to add expressiveness to the SST Kernels proposed by Collins and Duffy for natural language parse trees. This strategy aims to modify the recursive definition of the kernel function, in order to enlarge the corresponding feature space.

To this aim, the author proposed the *Partial Tree Kernel* (PT) which is able to apply *partial* matchings between subtrees.

Graph Kernels

Similarly to trees, some research efforts are being devoted in the definition of proper kernel functions that are able to compare graph structured data. More formally:

Definition 2.32. (Graph Kernels): *[76] Let $\mathcal{G}$ denote the set of all graphs, and let $\Phi : \mathcal{G} \to \mathcal{H}$ be a map from this set into a dot product space $\mathcal{H}$. Furthermore, let $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ be such that $\langle \Phi(G), \Phi(G^i) \rangle = k(G, G^i)$. If Φ is injective, k is called a* **Complete Graph Kernel**

However, in the general case, the computation of graph similarities is intrinsically *untractable* problem:

Proposition 2.10. *Computing any Complete Graph Kernel is at least as hard as deciding wether two graphs are isomorphic [76].*

In more details:

Proposition 2.11. (Subgraphs Kernel) *[76]:*
Let G and $G^{'}$ be two graphs on a metric space $\mathcal{G}$, and $\sqsubseteq$ the relationship such that $\sqsubseteq (S, G)$ is true iff S is a subgraph of *G (Definition 2.20) (shortly $S \sqsubseteq G$). Thus, the kernel*

$$k_{sg}(G, G^{'}) = \sum_{S \sqsubseteq G} \sum_{S' \sqsubseteq G'} k_{ismf}(S, S') \tag{2.39}$$

where

$$k_{ismf}(S, S') = \begin{cases} 1, & \text{if } S = S' \\ 0, & \text{otherwise} \end{cases} \tag{2.40}$$

Nevertheless, the first condition in Equation 2.40 holds iff the two graphs are *isomorphic*, namely there exists an *isomorphism* between them (Definition 2.18). However, the problem of determining if such isomorphism exists has been proven to be a *NP-Hard* problem [47].

As a consequence, most of the solutions proposed in the literature define Graph Kernels that are specifically suited for a particular *class* of graph structures, making some preliminary assumptions on the input and/or the considered feature

space. For example, the definitions of Tree Kernels consider "specialised" versions of graphs, i.e., trees (Definition 2.19).

Moreover, Gärtner et al. in [76] propose the formulation of the so-called *Product Graph* Kernels, which leverages the definition of the *directed graph product* (Definition 2.11) to accomplish the matching of substructures.

Definition 2.33. Product Graph Kernel *[76]:*
Let G and G^i two graphs on a metric space $\mathcal{G}$, $G_\times = (V_\times, E_\times)$ the direct product of G and G', with adjacency matrix *equals to $E_\times = E(G \times G')$ [47].*
With the sequence of weights $\lambda = \lambda_0, \lambda_1, \ldots$ ($\lambda_i \in \mathbb{R}; \lambda_i \geq 0$ for all $i \in \mathbb{N}$), the **Product Graph Kernel** *is defined as:*

$$k_\times(G, G') = \sum_{i,j=1}^{|V_\times|} \left[\sum_{m=0}^{\infty} \lambda_m E_\times^m \right]_{ij} \tag{2.41}$$

if there exists

$$\lim_{n\to\infty} \sum_{m=0}^{n} \lambda_m E_\times^m \tag{2.42}$$

Therefore, the existence and the computation of the whole kernel function strongly depend on the particular choice of the sequence of weights λ_m, and thus on the corresponding *matrix power series* inducted by the selected sequence (Equation 2.42). In fact, if we select $\lambda_m = \lambda^m$, then the $\sum_m \lambda_m$ corresponds to the geometric series that is known to converge if and only if $|\lambda| < 1$. In this case, the limit is given by $\lim_{n\to\infty} \sum_{m=0}^{n} \lambda^m = \frac{1}{1-\lambda}$.
Similarly, authors in [76] define the *geometric series of a matrix* as:

$$\lim_{n\to\infty} \sum_{m=0}^{n} \lambda^m E^m$$

If $\lambda < 1/a$, where $a \geq \min(\Delta^+(G), \Delta^-(G))$, corresponding to the maximal out-degree/in-degree of the graph G, respectively (Remark 2.13). Feasible computation of the limit of a geometric matrix series is possible by inverting the matrix $\mathbf{I} - \lambda E$, which can be computed in roughly cubic time complexity[76]. The proof is almost intuitive and is reported in [76].

Therefore, the *Geometric Product Graph* Kernel is defined as:

$$k_{\times}(G, G') = \sum_{i,j=1}^{|V_{\times}|} \left[\sum_{m=0}^{\infty} \lambda_m E_{\times}^m \right]_{ij} = \sum_{i,j=1}^{|V_{\times}|} \left[(\mathbf{I} - \lambda E_{\times})^{-1} \right]_{ij} \tag{2.43}$$

Differently, Menchetti et al. in [138] propose a Graph Kernel that belongs to the family of Convolution Kernel (Definition 2.31), called *Weighted Decomposition Kernel* (WDK). In particular, the WDK is computed by dividing objects into substructures indexed by a selector. Two substructures are then matched if their selectors satisfy an equality predicate, while the importance of the match is determined by a probability kernel on local distributions fitted on the substructures [138].

One of the key feature of the WDK is that no prior assumptions are made on the topology of analysed graphs, making the kernel able to handle very large families of graph structures. The only restriction concerns the fact that graphs are supposed to be *directed* (Definition 2.15), *acyclic* (Definition 2.16), and *labelled* (Definition 2.10).

The basic idea of WDK is that, given a pre-determined $R - convolution$ (Remark 2.18) relationship on the feature space, the Graph Kernel is obtained by the composition of two fundamental elements, namely the *selector* and the *context*. More formally, WDK is characterised by:

Definition 2.34. (WDK Structures)
Let $\mathcal{G}, \widehat{\mathcal{G}}_1, \mathcal{G}_2, \ldots, \mathcal{G}_D$ be a $D+1$ non empty separable metric spaces such that $\mathbb{G} = (\widehat{\mathcal{G}}_1, , \mathcal{G}_2, \ldots, \mathcal{G}_D)$ is a D-tuple of non empty subgraphs of $\mathcal{G}$. Moreover, let $\mathbf{k} = (k_1, \ldots, k_D)$ be a D-tuple of positive definite kernels such that
$k_d : \mathcal{G}_d \times \mathcal{G}_d \to \mathbb{R}, 1 \leq d \leq D$, and let $R(\mathbb{G}, \mathcal{G})$ be a convolution relationship.
A Weighted Decomposition Kernel is characterised by the following decomposition structure:

$$\mathcal{R} = \langle \mathbf{X}, R, (\delta, k_1, \ldots, k_D) \rangle \tag{2.44}$$

where $R(\widehat{G_1}, G_2, \ldots, G_D, G)$ is true iff $\widehat{G_1}$ is a subgraph of G called the **selector** *and $\mathbf{G} = (G_2, \ldots, G_D) \in \mathcal{G}_2 \times \ldots \times \mathcal{G}_D$ is a tuple of subgraphs of G called the* **contexts** *of occurrence of $\widehat{G_1}$ in G.*

The main advantage of this formulation is that WDK defines a general Graph Kernel computational framework, which could be easily tailored to the specific problem and domain by a proper definition of a *selector* and *contexts*. However, in order to ensure an efficient kernel computation, some restrictions are placed on the sizes of the above entities [138]. The first assumption concerns the inverse R-convolution relationship, i.e., $R^{-1}(G)$, such that

$$|R^{-1}(G)| = O(|V_G| + |E_G|)$$

In other words, it is assumed that the number of ways a graph can be decomposed grows at most linearly with its size [138].

Furthermore, selectors are assumed to have constant size w.r.t. the graph G, i.e.,

$$R(\widehat{G_1}, G_2, \ldots, G_D, G) \Rightarrow |V_{\widehat{G_1}}| + |E_{\widehat{G_1}}| = O(1)$$

Finally, the general definition of **WDK** is completed by the kernel on parts:

$$K(G, G') = \sum_{\substack{(\widehat{G_1}, \mathbf{G}) \in R^{-1}(G) \\ (\widehat{G_1}', \mathbf{G}') \in R^{-1}(G')}} \delta(\widehat{G_1}, \widehat{G_1}') \sum_{d=2}^{D} k_d(G_d, G_d') \tag{2.45}$$

Part B

Original Contributions

Programming today is all about doing science on the parts you have to work with.

Gerald Jay Sussman

3

Weighting Source Code Lexical Information with a Probabilistic Model for Software Re-modularisation

In the literature some approaches have been proposed to partition software systems into meaningful subsystems exploiting the lexical information provided by programmers into the source code (Section 1.3). However these techniques usually do not consider the programming language sections in which the lexicon appears (e.g.: comments, class names, method names) even if it is a common experience that programmers place different care in choosing terms for different constructs.

However, it is arguable that programmers may place different care in choosing terms for different constructs. For example, developers may choose differently the terms to use for code variables rather than for comments, according to many factors, such as their programming attitudes (e.g., coding conventions), the time-

to-market pressure, the language they used, and the development context.

To investigate the conjecture that the lexical information provided by programmers may convey different levels of relevance, the contribution presented in this Chapter defines a novel approach towards software clustering, considering separately the contribution of six different vocabularies. They are composed of terms extracted by the different code structures, referred as *zones*, where a programmer can add lexical information, namely: (I) *Class Names*, (II) *Attribute Names*, (III) *Function Names*, (IV) *Parameter Names* (V) *Comments* and (VI) *Source Code Statements*.

Thanks to this separation, we applied an automatic weighting mechanism to exploit the contribution of each vocabulary. Since each software system has its own development peculiarities, no general weighting schema can be imposed *a-priori*, but rather it should be tailored for each specific system at the hand.

To this aim, we introduced a *probabilistic model* of the data, whose parameters, including the zone weights, are optimised by means of an iterative algorithm, namely the *Expectation-Maximisation* (EM) [136].

In more details, the software clustering approach described in this Chapter aims at generating software partitions relying only on the lexical information contained within the source code of the analysed software system. Thus in our definition, generated system partitions, (i.e., *clusters*), will contain lexically related software artefacts.

To group related artefacts into different partitions, our approach consists of a pipeline process composed by the following three steps:

1. The first step (Section 3.1) is responsible for the indexing all the source files of the analysed systems in order to collect all the terms appearing in each considered zones. All these terms, together with their corresponding *documents* (i.e., artefacts) are then stored in an Information Retrieval (IR) index [130] further processed in next steps.

2. The second step (Section 3.2) is devoted to automatically weight each zone, thanks to the application of the EM algorithm. Such weights will be applied as multipliers in a *vector space model* representation [130] of the software artefacts, useful to compute similarity among them.

3. In the final step (Section 3.3), all the related artefacts are grouped together by means of a clustering algorithm. In particular, two different well-known clustering algorithms are considered, namely *K-Medoids* [97], and *Group Average Agglomerative Clustering* (GAAC) [148], which have been properly customised to make them more suitable for the software domain.

A summary description of these processing steps is depicted in Figure 3.1.

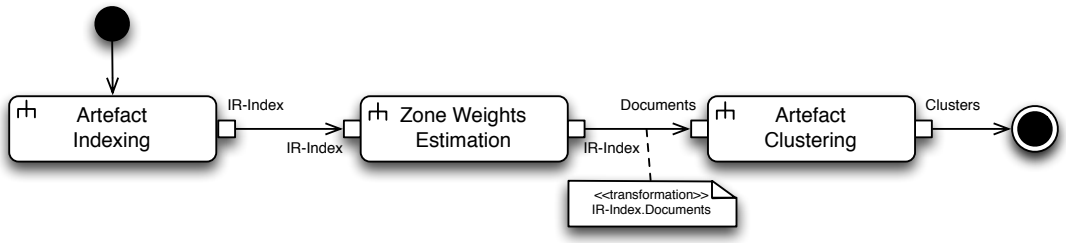

Figure 3.1: The Overall Software Re-modularisation process.

To evaluate whether the introduction of the probabilistic model as well as the use of different clustering algorithms improved the resulting partitions, the approach has been assessed in a case study (Section 3.4).

Since no "gold-standard" partition is available in the software modularisation domain [184], we selected a set of 19 well-known open source software systems implemented in Java and we assessed whether the proposed approach was able to automatically group classes in a fashion resembling an *authoritative partition*, i.e., the original partition defined for these systems. In particular, the authoritative reference partition is gathered from the organisation of the package structures, as already done in other works (e.g.: [28, 163, 165, 187]).

Achieved results indicate that the introduction of the probabilistic model highly enhances the process, leading to clusters significantly more similar to the authoritative partitions.

3.1 Investigating the Use of Source Code Lexical Information

The first task towards the definition of our software clustering approach regards the definition of a technique that is able (I) to extract the lexical information from the source code (Section 3.1.1), and (II) to organise it into some meaningful structures, suitable for further processing (Section 3.1.2).

3.1.1 Source Code Indexing

Since we are interested in the processing of lexical information, we assume that each source file can be treated as a common plain-text document to which text mining and Information Retrieval techniques can be applied.

The first operation required by IR techniques concerns the definition of the the so-called *document unit* [130], namely the granularity at which source files have to be processed. This aspect is particularly important as this correspond to the precise definition of the *artefacts* to consider in the following clustering processing. Since we are interested in dealing with the clustering of object-oriented software systems, we choose the *class* as atomic element, namely the artefact to be indexed. In other words, each class found in the analysed source files is regarded as a different *document*, according to the IR terminology [130].

From now on, we will use the terms *class*, *artefact* and *document* interchangeably.

The extraction and the collection of the terms from the different zones is provided in the so-called (artefact) *indexing* process [130], whose operational steps are depicted in Figure 3.1.

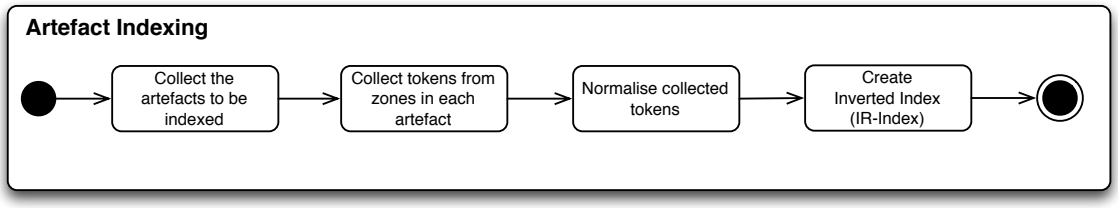

Figure 3.1: Activity Diagram of the Artefact Indexing Processing Step.

For each class of the system, we analyse the relevance of the lexicon extracted by each of the five zones identified by Abebe et al.[5], namely Class Names *(CN)*, Attribute Names *(AN)*, Function Names *(FN)*, Parameter Names *(PN)*, Comments *(Co)*. Moreover, we include also the lexicon extracted by the Source Code Statements *(SCS)*. In particular:

I) The *CN* zone contains all the terms appearing in the signature of a class, i.e., the **name of the class** and possibly the **names of base classes** and/or the **names of implemented interfaces**.

II) The *AN* zone contains the words in the **names of attributes and constants** of the class and their corresponding types (if their are not primitive).

III) The *FN* zone contains the words appearing in **method names** and in their return type (if it is not primitive).

IV) The *PN* zone contains the words that are in **method parameters**, including both the names and the types (if they are not primitive).

V) The *Co* zone contains all the terms extracted from **comments**. It is worth noting that possible copyright disclaimer placed as beginning comment are removed as already done in other works (e.g., [108]). In particular such comments do not provide any specific significance to the source file in which they appear and consequently they are simply discarded.

VI) The *SCS* zone contains all the lexicon occurring in the **body of methods**, such as the names of local variables, invoked methods, and passed parameters.

Therefore, to keep track of the different zones of code from which terms are gathered, every single document in the *collection* is represented as a set of six different buckets (Figure 3.2).

To better explain the indexing process and all the applied operations, a sample Java class is reported in Listing 3.1.

Therefore, given the different artefact to be indexed, the next processing step involves the application of the so-called *tokenization*. Such operation consists in

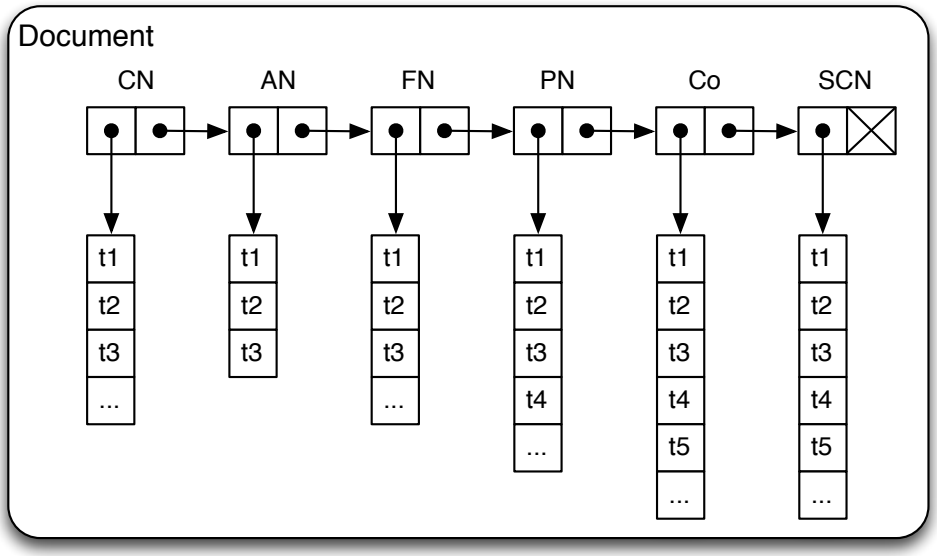

Figure 3.2: Document representation as a set of six different zone buckets.

segmenting the input text into pieces, called *tokens* [130]. In general, such tokenization is performed by separating the different terms on blank and punctuation characters. However, in order to correctly avoid the extraction of useless and noisy information, all the terms in comments are previously filtered in order to remove likely occurring HTML tags as they do not provide any significance for the computation (Lines 1 - 10 in Listing 3.1).

Afterwards, all the terms extracted from documents are processed (i.e., *normalised* [130]) in order to reduce noise and redundancies.
Such *normalisation* applies a set of linguistic pre-processing and transformations to the collected terms, plus an additional step to deal with specific peculiarities of the source code vocabulary [26].

In more details, the applied normalisation consists of the following operations:

1. All the terms written using different coding conventions are split. As a matter of fact, identifiers are usually composed by concatenating multiple words (Section 1.2). To date, the approach is able to handle only the use of

```
1  /**
2   * A handle that doesn't change the owned figure. Its only purpose is
3   * to show feedback that a figure is selected.
4   * <hr>
5   * <b>Design Patterns</b>
6   * <img src="images/red-ball-small.gif" width=6 height=6 alt=" o ">
7   * <b>NullObject</b><br>
8   * NullObject enables to treat handles that don't do
9   * anything in the same way as other handles.
10  */
11 public class NullHandle extends LocatorHandle {
12     /**
13      * The handle's locator.
14      */
15     protected Locator fLocator;
16 
17     public NullHandle(Figure owner, Locator locator) {
18         super(owner, locator);
19     }
20     /**
21      * Draws the NullHandle. NullHandles are drawn as a
22      * red framed rectangle.
23      */
24     public void draw(Graphics g) {
25         Rectangle r = displayBox();
26         g.setColor(Color.black);
27         g.drawRect(r);
28     }
29 }
```

Listing 3.1: Sample Java class (Extracted from JHotDraw 5.1)

the *Camel-case* and *Alternate-case* (capitalised letters used to divide words). For instance, in our example, the variable `NullHandle` is split in two distinct words, i.e., `Null` and `Handle`. The integration of more refined techniques that are also able to deal with abbreviations, such as the one presented in Section 4.1, is a future work we wish to address. Finally, all the remaining terms are lowercased.

2. All the words appearing in a list of common terms, known as *stop words* (e.g.: *the, a, is*, etc...), are removed [130]. This is because such terms do not provide any contribution for the analysis.
   To take into account the peculiarities of the considered domain, we apply different *stop words* lists to the six different zones. In particular, we remove the most common English terms* occurring in the first four zones. As for the fifth and sixth zones, also all the keywords of the programming language are deleted. Considering the sample class in Listing 3.1, the terms `a, as, are, the, in, is, its, only` and `that` are stopped.

3. All the terms are gathered in *equivalence classes* based on their morphological root, or *stem*. Stems allow to discard all the superficial differences in characters (such as plurals, verb conjugations, etc...), and to group all the terms referring to the same "concept".
   To generate the stem of each term, we apply the well-known Porter's algorithm [152]. As for the example class in Listing 3.1, the terms `handles` and `handle` appearing in the comments (Lines 1 - 10, Line 13, and Lines 21 - 22) are all reduced to the common stem `handl` and regarded as a unique term that occurs 6 times.

Once the normalisation process is completed, the final step of the indexing process involves the construction of the so-called *inverted index* [130]. This structure represent one of the central major concept in IR approaches as it provides an efficient strategy to store the different terms and the references to their corresponding documents [130]. The basic idea of an inverted index is to keep a *dictionary* of

*http://www.textfixer.com/resources/common-english-words.txt

terms, whose entries correspond to the list of documents containing the corresponding term, called *postings* [130].

However, in the proposed approach, we want to keep track of the particular zone from which a single term has been extracted, since they are treated as different terms. To this aim, we postpone to each term a suffix corresponding to the name of their corresponding zone, namely _cn, _an, _fn, _pn, _co or _scs. For example, the term `locator` in Listing 3.1 appears at the same time in five out of six zone, namely CN (Line 11), AN (Line 15), PN (Line 17), Co (Line 13, and SCN (Line 18), respectively. Therefore, in this case the dictionary will contain five different entries, corresponding to the five different zones in which the term `locator` appears, namely `locator_cn`, `locator_an`, and so forth.

Finally, it is worth noting that such modification to the terms added to the dictionary also change the meaning of their corresponding postings lists. In fact, in our case, the definition of postings lists becomes: "the list of documents containing the corresponding term *in a particular zone*". Thus, for example, the posting list of term `locator_an` references all the documents where the term `locator` appears in the AN zone.

3.1.2 Representation Model

Following an approach widely adopted by information retrieval systems, we represent each document (namely class) of the collection as a *bag-of-words*, i.e., the multi-set of all the tokens contained in the document, given by all words and their occurrences. In this way, word order gets lost, and any further processing is only based on lexical information.

Within the *bag-of-words* model [130], each document is represented by an array of real numbers whose elements are associated to the corresponding terms in the dictionary. This vectorial representation of the documents allows to represent each document as a point in a multi-dimensional geometrical space: the so called *Vector Space model* [130]. In this space, the similarity among different documents could be expressed in terms of the so-called *cosine similarity* [130] based on the computation of the inner product among normalised document vectors.

In more details, for each term of the dictionary and for each document, a score

following the *idf* schema [130] is computed. Indeed, the *idf* is adopted in a large number of IR applications because of the good trade-off between simplicity and effectiveness in describing the relevance of the terms with respect to the documents. It is defined as follows: given a collection of N documents, namely the number of classes of the system under investigation, the $tf(t,d)$ (*term frequency*) is defined as the number of occurrences of the term t in the document d.

On the other hand, while the $df(t)$ (*document frequency*) indicates the number of documents in which the term t occurs, its inverse gives a evaluation of the rareness of the term. The rationale underlying the *idf* is that, when computing similarity among documents, if a term appears in almost all the documents of the collection, then its discriminative contribution is irrelevant. Conversely, the value of the *idf* is high when a term appears in few documents, attaining its maximum value ($\log N$) when the term occurs in a unique document. Among the different applications of the *idf* schema, due to its numerically good behaviour, we adopted the following one:

$$tf - idf(t,d) = \sqrt{tf(t,d)} \cdot \log \frac{N}{df(t)+1} + 1 \tag{3.1}$$

Concluding, each document is therefore represented by a vector having size equal to the dictionary size, where each element corresponds to the *idf* score for the term in the document. For all the terms not belonging to the document, the corresponding element in the vector is zero.

3.2 Probabilistic Models for Software Re-modularisation

In this approach, we investigate the conjecture that the considered zones of the code could convey information of different relevance starting from the observation that developers may place different care in writing code as well as comments. Therefore, the informative contribution of the different zones should be correctly weighted to best exploit the conveyed information. Moreover since these weights strongly depend on the specificities of each project, their choice can not be made subjectively, but should be automatically estimated from the data.

To this aim we define an automatic technique that is able to estimate such "relevance" on the basis of the lexical characteristics of each considered project.

In particular, we are interested in determining the weights of the zones to be used as multiplicative factors for the *idf* values of the terms during the similarity computation among documents.

A well founded framework to solve such a problem is given by probabilistic approaches, where different sources of information are combined by an *a-priori* probability distribution.

The *Maximum Likelihood Estimation* (MLE) is one of the most widely adopted approaches to estimate parameters of a probabilistic model.

This approach aims at finding the parameters of the considered model which maximise the probability of the set of samples. In simple words, we defined three probabilistic models describing how the words are generated and then we chose for these models the parameters which maximise the probability of our data.

The underlying idea is that each document is produced in two steps: first of all, a discrete random variable chooses the zone to which the token belongs. The distribution probability of this variable corresponds to the *a-priori* probability of each zone, and is multiplied for the probability of the document.

In more details, if we look at the Z zones (six in our case) as a partition of documents, the probability of each document d is given by:

$$\Pr(d) = \prod_{z=1}^{Z} \Pr(d|z) \tag{3.2}$$

Then, the formulation of the different conditional probabilities of documents is characterised by the particular probabilistic model assumed on the dictionary terms. In particular, without loss of generality, such models are assumed to be statistically independent [130].

Therefore, our general probabilistic model is expressed by a *mixture* of *multivariate* probabilistic distributions, that can be differently instantiated (and thus, formalised) according to the assumed distribution model of the data.

The main goal of this model is to evaluate the values of *a-priori* probabilities on the different mixtures, namely z (Equation 3.2) (and possible model parameters),

that maximise the (Log) Likelihood of the project:

$$\log \mathcal{L} = \sum_{i=1}^{N} \log \sum_{z=1}^{Z} z \Pr(d|z) \tag{3.3}$$

It is worth noting that this formulation is correct only if it is assumed that all the documents in the collection are statistically independent, so that the probability of the project is given by the product of the probability of each document.

In this approach, three different probabilistic models are considered, namely (I) the *Gaussian*, the (II) *(multivariate) Bernoulli* and (III) the *multinomial* models. In the first model, the *idf* values of terms in each zone are assumed to follow a Gaussian distribution, and are characterised by the values of mean (μ) and standard deviation (σ), one for each Gaussian model distribution, namely one for each zone. In addition to this, we also consider *Bernoulli* and *multinomial* probabilistic models [130]. In the former, the probability of the document is given by the product of all probabilities of the dictionary terms of being (or not being) in the document. On the other hand, in the latter, only the probabilities of terms appearing in the document are considered, taking into account the number of their occurrences. These two representations are conceptually quite different in their definitions. In fact, the Bernoulli model completely disregard in its formulation the number of occurrences of each term in the document; while in the multinomial model only the terms that appear in the input document contribute to the probability computation, with a factor that depends on the corresponding occurrences.

A summary description of the three different formulations of the considered probabilistic models is reported in Table 3.1.

3.2.1 An EM algorithm for Parameter Estimation

As discussed above, we estimate the model parameters by maximising the probability of the set of examples, in our case, the documents in the project, by applying the MLE criterion. To find the values maximising the likelihood, the EM iterative algorithm[53] is applied.

EM is an iterative algorithm whose name refers to the corresponding two main

	Gaussian	**Bernoulli**	**Multinomial**
		Document probabilities	
$\Pr(d\|z) =$	$\prod_{t=1}^{M} \mathcal{G}(\mu_z, \sigma_z)$	$\prod_{t:t\in d[z]} \Pr(t\|z) \prod_{t:t\notin d[z]} (1 - \Pr(t, z))$	$\prod_{t=1}^{M} \Pr(t\|z)^{tf(t,d[z])}$
$\Pr(d) =$	$\sum_{z=1}^{Z} \alpha_z \Pr(d\|z)$	$\sum_{z=1}^{Z} \alpha_z \Pr(d\|z)$	$\sum_{z=1}^{Z} \alpha_z \Pr(d\|z)$
		Initialization	
$\alpha_z =$	$\begin{cases} mu_z = \frac{1}{N} \sum_{d=1}^{N} x_{t,d[z]} \\ \sigma_z^2 = \frac{1}{N} \sum_{d=1}^{N} \left(x_{t,d[z]} - \mu_z\right)^2 \end{cases}$ $\frac{\sum_{d=1}^{N} \sum_{t=1}^{M} tf(t,d[z])}{\sum_{z'=1}^{M} \sum_{d=1}^{N} \sum_{t=1}^{M} tf(t,d[z'])}$	$\Pr(t, z) = \frac{1}{N} \sum_{d:t\in d[z]} 1$ $\frac{\sum_{t=1}^{M} \sum_{d:t\in d[z]} 1}{\sum_{z'=1}^{Z} \sum_{t=1}^{M} \sum_{d:t\in d[z']} 1}$	$\Pr(t, z) = \frac{\sum_{d=1}^{N} tf(t,d[z])}{\sum_{t'=1}^{M} \sum_{d=1}^{N} tf(t',d[z])}$ $\frac{\sum_{d=1}^{N} \sum_{t=1}^{M} tf(t,d[z])}{\sum_{z'=1}^{Z} \sum_{d=1}^{N} \sum_{t=1}^{M} tf(t,d[z'])}$
		Expectation step	
$r_{d,z} =$	$\frac{\alpha_z \Pr(d\|z)}{\sum_{z'=1}^{Z} \alpha_z \Pr(d\|z)}$	$\frac{\alpha_z \Pr(d\|z)}{\sum_{z'=1}^{Z} \alpha_z \Pr(d\|z)}$	$\frac{\alpha_z \Pr(d\|z)}{\sum_{z'=1}^{Z} \alpha_z \Pr(d\|z)}$
		Maximisation step	
$\alpha_z =$	$\begin{cases} \mu_z = \frac{\sum_{d=1}^{N} r_{i,z} x_{t,d[z]}}{\sum_{d=1}^{N} r_{d,z}} \\ \sigma_z^2 = \frac{\sum_{d=1}^{N} r_{d,z} \left(x_{t,d[z]} - \mu_z\right)^2}{\sum_{i=1}^{N} r_{i,z}} \end{cases}$ $\frac{\sum_{d=1}^{N} r_{d,z}}{\sum_{z'=1}^{Z} \sum_{d=1}^{N} r_{d,z'}}$	$\Pr(t, z) = \frac{\sum_{d:t\in d[z]} r_{d,z}}{\sum_{d=1}^{N} r_{d,z}}$ $\frac{\sum_{d=1}^{N} r_{d,z}}{\sum_{z'=1}^{Z} \sum_{d=1}^{N} r_{d,z'}}$	$\Pr(t, z) = \frac{\sum_{d=1}^{N} r_{d,z} tf(t,d[z])}{\sum_{t'=1}^{M} \sum_{d=1}^{N} r_{d,z} tf(t',d[z])}$ $\frac{\sum_{d=1}^{N} r_{d,z}}{\sum_{z'=1}^{Z} \sum_{d=1}^{N} r_{d,z'}}$

Table 3.1: EM computation for the three different models and with frequentist initialisation. $\mathcal{G}(\mu, \sigma)$ depicts the Gaussian distribution of mean μ and standard deviation σ. $x_{t,d}$ indicates the index for the token t in document d.

steps (see Table 3.1) it alternates during the execution: in the *Expectation* step, the weights corresponding to each pair (*document*, *zone*) are (re)computed on the basis of the parameters values. On the other hand, the *Maximisation* step (re)computes the model parameters in a way that the likelihood does not decrease. The algorithm halts when the increase in likelihood corresponding to a given iteration is smaller than a given threshold, or when a maximum number of iterations has been performed.

Finally, among all the resulting parameters, the algorithm returns the values of the zone priors: a large value of z suggests that the z-th zone contribution is important for the model. Thus, we want to combine the zone scores with weights proportional to these priors. As both weights and priors ought to sum to one, we choose priors exactly equal to weights.

Finally, it is worth noting that one of the problems of the EM algorithm is that it can attain a local maximum rather than a global one. Therefore, the choice of the initial values for the parameters is very critical for the optimisation results [137]. However, while in case of the Bernoulli and multinomial models, such initialisation is straightforward, as it is inducted by the model formalisation, in case of the Gaussian model, different strategies may be applied in order to set up the initial values of the model parameters, namely μ and σ. After a prior experimental investigation, in this case, we apply an initialisation strategy that estimates initial model parameters by considering the rate between the number of tokens in the zones and the total number of tokens. In particular, such strategy starts by assigning "more importance" to zones containing more lexical information.

3.3 Clustering of Software Artefacts

As already mentioned, the software re-modularisation problem is also referred sometimes in the literature with the name of *software clustering*, as it regards the clustering of related software entities.

Even if this definition is quite trivial, it emphasise that this problem has several aspects in common with a typical clustering problem as intended in the machine learning literature (Section 3.3).

In more details, the *software re-modularisation* problem belongs to the category

of *hard clustering* tasks, since all the entities, namely the classes of the system, can be associated to one and only one cluster (Section 2.3). Moreover, as any other *unsupervised* machine learning approach, one of the key issues of the technique is the choice of the similarity measure, which is crucial for the clustering performance since it states criteria to decide whether two software entities are similar enough to be put into the same cluster [131].

In the defined vector space model, the similarity between two classes is typically computed applying the well-known *cosine similarity* (Remark 2.8), expressed as the cosine of the angle determined by the two vectors representing them. Nevertheless, the clustering of software entities introduces some constraints imposed by the specific domain. The most important one is that an automatically produced partition should not be either too huge (i.e., containing hundreds of software entities) nor too tiny (i.e., containing very few software entities) [187].

For this reasons, standard algorithms may not be effective unless they are (slightly) modified to impose such constraints.

In the remainder of this Section, a description of the proposed customisation for two well-known clustering algorithms is reported. In particular, the *K-Medoids* clustering algorithm is described in Section 3.3.1), while the *Group Average Agglomerative Clustering* is discussed in Section 3.3.2).

3.3.1 K-medoids

As described in Section 2.3.2, the K-Medoids algorithm is a well-known variation of the classical K-Means algorithm, which is more robust with respect to noise and outliers. Moreover, since the resulting clustering strongly depends on the initial choice of medoids, initial medoids are randomly selected. However, to avoid unbalanced solutions, we introduced a novel halting criterion to avoid the risk of resulting in extremely small or extremely large clusters, which makes sense in the context of software re-modularisation.

Indeed, the original K-medoids algorithm starts with a random choice of the k medoids and iterates assigning at each step all the entities to the most similar medoids, and then recomputing the set of medoids.
Finally the algorithm returns the desired partitions organised as a set of k different

clusters. An algorithmic description of the K-medoids algorithm is reported in Algorithm 1.

However, the main drawback of the algorithm is that resulting clusters strongly depends on the initial configuration. Thus, unlucky configurations could result in a partition including too small clusters: in the variant of the algorithm proposed, the whole procedure is repeated until a final solution where non-extreme clusters is attained or a maximum number of iterations are performed.

Even when the procedure halts due to the latter condition, the algorithm provides the best solution among all the ones found in each iteration.

3.3.2 Group Average Agglomerative Clustering

In addition to the K-Medoids algorithm, also the Group Average Agglomerative Clustering (GAAC) one has been considered, which belongs to the category of the hierarchical clustering algorithm (Section 2.3.3).

In particular, the GAAC algorithm employes a linkage strategy that aggregates two clusters based on the the average similarity of all pairs of entities belonging to them (see Table 2.1). The main advantage of such strategy is that it is more robust with respect to outliers and tends to produce more balanced dendrograms [130]. An example of a dendrogram resulting after the application of GAAC algorithm is reported in Figure 2.2.

The main feature of hierarchical clustering algorithms is that they are deterministic and does not require several random initialisation (as for partitional clusterings, e.g., K-medoids). Moreover, although the asymptotic time complexity of the HAC approach is worse than K-medoids one (Section 3.3.2), in the experiments we performed the K-medoids was slower because it was applied a large number of times on different initial points.

Conversely, from a software re-modularisation perspective, the main drawback of HAC is that it does not provide a flat partition of the system due to its agglomerative nature. Therefore, to get such partitions, the dendrogram has to be properly cut [132]. To this aim, the proposed customisation of the HAC algorithm consist in a specialised cutting strategy criterion. In particular, this strategy optimises the non extremity distribution of the partitions aiming at generating at

Algorithm 5 GAAC Cutting Strategy

Input: Λ : The maximum number of elements admitted in a single cluster.
Input: `T` : The dendrogram to be cut.
Input: `k` : the number of partitions to generate.
Output: `P` : The set of k different partitions.

```
1: function GAACutStrategy(Λ, T, k)
2:     r ← root(T)
3:     if r = null then
4:         return P
5:     end if
6:     if (isLeaf(r)) ∨ (|P| ≥ k) then
7:         P ← P ∪ {T}
8:         return P
9:     end if
10:    leftT ← subtree(left(T))                    ▷ Get the left subtree rooted in T
11:    rightT ← subtree(right(T))                  ▷ Get the right subtree rooted in T
12:    if (|leftT| ≥ Λ) ∧ (|rightT| ≥ Λ) then
13:        P ← P ∪ GAACutStrategy(Λ,leftT,k)
14:        P ← P ∪ GAACutStrategy(Λ,rightT,k)
15:    else if (|leftT| ≥ Λ) ∧ (|rightT| < Λ) then
16:        P ← P ∪ {rightT}
17:        P ← P ∪ GAACutStrategy(Λ,leftT, k)
18:    else if (|leftT| < Λ) ∧ (|rightT| ≥ Λ) then
19:        P ← P ∪ {leftT}
20:        P ← P ∪ GAACutStrategy(Λ,rightT, k)
21:    else                                        ▷ None of the two partition is extreme
22:        P ← P ∪ {leftT}
23:        P ← P ∪ {rightT}
24:    end if
25:    return P
26: end function
```

most k clusters. It is worth noting that this latter aspect is very important for the assessment of the approach as it makes the two clustering solutions, namely K-medoids and HAC, fairly comparable. The algorithm for the proposed cutting strategy is reported in Algorithm 5

3.4 Experimental Settings

The assessment of a clustering is typically based on an annotated test set, the *gold standard* [130], in which each item of the dataset is labeled with the corresponding cluster. In case of software clustering tasks, this gold standard could be represented by a set of large and publicly available software systems with well-understood decomposition that can be used as benchmark [184]. However, from one hand, there is no publicly annotated dataset available; on the other hand, the manual generation of such partitions by software architects may be too subjective to represent a benchmark.

Therefore, following other similar works [28, 163, 187], we adopted a fair and repeatable procedure for constructing the gold case clustering which is built on the original source folder structure of the system under investigation. The idea behind this protocol is the following: given the bunch of classes of a well-engineered system (such as for instance *JHotDraw*, widely used to teach Software Design issues) without any structure, if the approach is able to automatically arrange them in a partitioning that resembles the packages proposed by the developers of the system, then the approach will likely perform well also on other software systems. From the software engineering point of view, this measure is called *Authoritativeness (Auth)* [187].

The *authoritative partition* is automatically derived in accordance with the following three steps:

1. create the subsystem hierarchy based on the directory (package) structure (each directory represents a single subsystem);

2. merge a subsystem with its parent if it contains less than five source files;

3. create a cluster for each resulting subsystem.

Given such authoritative partition, the next challenge is to determine a measure that is able to compare clustering results to this partition. Several researchers in literature have attempted to tackle this problem [11, 103, 109, 184].

One of the first proposed approach was the measure presented by Lakhotia and Gravely [109]. However this measure could be used only on dendrograms of

hierarchical clusterings which in practice strongly limits its applicability to other not-hierarchical clustering algorithms.

Afterwards Anquetil and Lethbridge proposed the use of the well known measures of *Precision* and *Recall* for the evaluation of clustering results [11]. In particular, let A be the automatically identified source partition and B the authoritative partition, they defined the Precision as the percentage of intra-pairs, i.e. pairs of items in the same cluster, in A that are also intra-pairs in B. On the other hand, the Recall is defined as the percentage of intra-pairs in B that are also intra-pairs in A. The main drawback of this measure is that it is too much "sensitive" to the number and the size of considered clusters. As a consequence, few misplaced entities in a cluster could produce very different results.

Koschke and Eisenbarth presented in [103] a complex measure which extends and removes limitations of the approach proposed by Lakhotia and Gravely and that is loosely based on the Precision and Recall measures employed by Anquetil and Lethbridge. The *KE* measure is built on the definition of *GOOD* and *OK* matches. Assuming that p is a threshold parameter and that A_i and B_j are two clusters in the source and authoritative partition respectively, the following two definitions hold:

$$\text{(GOOD match) } A_i \approx_p B_j iff \frac{|A_i \cap B_j|}{|A_i \cup B_j|} \geq p$$

$$\text{(OK match) } A_i \subseteq_p B_j iff \frac{|A_i \cap B_j|}{|A_i|} \geq p$$

These two matching definitions are then used to split the set of clusters in two distinct classes, one for each relationship. Next, once all the clusters have been classified, the GOOD class is enlarged by joining all the OK matches in which one of the two cluster is already in the GOOD class. All the remaining clusters that are neither in GOOD or in OK matches are referred as *false positives* or *true negatives* in case they belong to the source or to the authoritative partition, respectively.

Finally the overall similarity metric is defined as follows:

$$KE(A,B) = \frac{\sum_{(a,b)\in GOOD} \frac{|a \cap b|}{|a \cup b|} + \sum_{(a,b)\in OK} \frac{|a \cap b|}{|a \cup b|}}{|GOOD| + |OK| + |truenegatives|}$$

The *KE* metric is particularly good when the source partition is close to the authoritative partition. Conversely it is not as good in more extreme cases as its definition takes into account only the union and the intersection between clusters, without applying any penalty for the join operations. Last but not least, it relies on the specification of a threshold parameter which could inevitably bias the results.

More recently, Tzerpos and Holt presented in [178] the *MoJo* distance, which is the measure this work builds on. In particular, let A be the automatically identified source partition and B the authoritative partition, $MoJo(A,B)$ is defined as:

$$\text{MoJo(A, B)} = \min(\text{mno}(A,B), \text{mno}(B,A))$$

corresponding to the minimum number of *Move* and *Join* operations necessary to transform either the first partition A to the second partition B or vice versa [178]. The lower the value of MoJo between two partitions is, the more the clustering algorithm is effective in creating the software partition.

Differently from the *KE* measure, this metric explicitly introduce the calculation of a penalty to the join operations but it has a couple of drawbacks that make its original formulation useless for the assessment of our approach. First of all we are interested in determining how the automatically defined partition resembles the authoritative one and not vice versa. Thus we need to calculate only the $mno(A,B)$. Furthermore, the measure does not make the results comparable among different software systems as its value strongly depend on the size of their authoritative partitions.

Therefore, to overcome those limitations, we used a normalised version of *MoJo*, namely the *MoJoFM*[184] defined as follows:

$$\text{MoJoFM(A, B)} = 1 - \frac{\text{mno}(A,B)}{\max(\text{mno}(\forall A, B))} \tag{3.4}$$

where max(mno($\forall A, B$)) calculates the maximal distance to partition B from every possible partition derived by elements of A. In our case study we used the implementation of MoJo available at http://www.cse.yorku.ca/~bil/downloads.

In conclusion, the Auth measure gives an estimate of the similarity between the clustering proposed by our approach and those in an authoritative partition.

Indeed, we also require that the obtained clustering does not include too small or too large clusters, similarly to other related work [28, 163, 165].

To this aim, a measure called *Non-extremity cluster distribution (NED)* has been introduced by [187]. NED is defined as follows:

$$NED = \frac{1}{N} \sum_{c_i \in C : \lambda \leq |c_i| \leq \Lambda} |c_i|$$

where N is the number of classes of the analysed software system, C is the set of clusters, and λ,Λ parameters indicate the minimum and the maximum size allowed for clusters, respectively. In accordance to other similar researches, we limited cluster size to be included between $\lambda = 5$ and $\Lambda = 100$. In other words, clusters with less than 5 or more than 100 software entities are considered as extreme lower and upper limits, respectively [187]. The larger the NED value is, the more non-extreme the size distribution of the clusters is.

Therefore, it is worth noting that the Authoritativeness provides an indicator of the "quality" of the clusters identified by the approach. Differently, the NED is defined to assess only other characteristics of the clustering based approach.

The assessment phase aims at evaluating the contribution of weighted zones to clustering performance. However, the question is two-fold, because the improvement could come from the zone introduction alone or from the zones when weighted by the probabilistic models we consider. Therefore, the following three systems' configurations are considered for the assessment:

***complete* system** includes all the proposals, namely the lexical features with zones weighted by means of the EM algorithm, and exploited by one of the two clustering strategies considered;

***flat* system** is the baseline where zones are not considered at all: the clustering algorithm is applied to the lexical features without zones and weights;

***unweighted* system** represents an intermediate case which only disregards the probabilistic model: the clustering algorithm exploits the lexical features with zones but without any weighting schema.

Note that the flat and the unweighted configurations consider different idf scores. In fact, when no zone is considered, the tf is given by the number of occurrences of the term in any part of the document, while the document frequency is the number of documents in which the term occurs. On the other hand, when zones are introduced and the corresponding idf values are combined with a constant weight, the score is computed for each zone separately and the results are then summed. While the flat configuration tf can be obtained by summing the tf for each zone, this is not the case for the df, as a term can occurs in a different number of documents depending on the considered zone.

According to the criteria, the measures and the systems defined above, we formulated the following two research questions:

RQ1: Does the *unweighted* system outperform the *flat* approach?

RQ2: Does the *complete* system outperform the *unweighted* one?

3.4.1 The Dataset

To conduct the investigation presented here, we have used the following 19 open source Java software systems:

1. **Apache Ant** - a Java library and command-line tool aimed to define build files for software applications implemented in the Java language.
2. **Apache Lucene** - a Java framework that implements IR algorithms.
3. **Apache Tomcat** - a well-known Servlet/JSP container for Java web applications.
4. **Azureus** - Azureus is a Java-based client for sharing files using the BitTorrent file-sharing protocol.
5. **Hibernate** - an ORM (Object Relational Mapping) library for Java applications.

6. **iText.Net** - iText is an open source library for creating and manipulating PDF, RTF, and HTML files in Java.

7. **jEdit** - a text editor suited to support programming tasks.

8. **jFreeChart** - a tool supporting the visualisation of bar charts, pie charts, line charts, scatter plots, histograms, simple Gantt charts, bubble plots, and more.

9. **jFTP** - a graphical Java network and file transfer client.

10. **jHotDraw**- a GUI framework for technical and structured graphics.

11. **jRefactory** - a GUI application for the refactoring of Java applications.

12. **jUnit** - the Java version of the xUnit testing framework.

13. **Liferay Portal** - open source enterprise web platform for building business web-based solutions.

14. **Pmd** - a Java source code analyser. It finds unused variables, empty catch blocks, unnecessary object creation, and so forth.

15. **Synapse** - Synapse is a lightweight and high-performance Enterprise Service Bus application, which provides support for XML, web services and REST applications.

16. **Tiger Envelopes** - Tiger Envelopes is an open source personal mail proxy that automatically encrypts and decrypts mail.

17. **Velocity** - a Java framework to build web and non-web applications.

18. **Xalan** - a XSLT processor in Java for transforming XML documents into HTML, text, and other XML document types.

19. **Xerces** - a collection of components and utilities to parse, validate, and serialise XML documents.

System	Classes	KLOCs	KCLOCs
Apache Ant	1452	103.5	89.6
Apache Lucene	1015	63.2	36.1
Apache Tomcat	1530	163.8	110.5
Aureus	4785	333.1	97.1
Hibernate	2267	156.0	95.8
iText.Net	1201	77.4	50.3
jEdit	869	88.4	36.1
jFreeChart	89	8.7	7.8
jFTP	469	23.5	4.7
jHotDraw	899	73.0	38.3
jRefactory	1522	110.7	91.3
jUnit	547	15.0	4.1
Liferay Portal	3961	379.1	137.3
Pmd	680	49.6	8.9
Synapse	613	45.7	20.9
Tiger Envelopes	917	73.4	25.9
Velocity	419	35.8	25.0
Xalan	915	123.7	128.3
Xerces	578	71.5	63.6

Table 3.1: Descriptive statistics of considered dataset

Some descriptive statistics of the dataset are reported in Table 3.1. In particular, this table shows names of the software systems and the analysed versions, together with the corresponding total number of classes, thousands of lines of code (KLOCs), and thousands of lines of comments (KCLOCs).

3.5 Results and Discussions

In this section, we present the achieved results and threats that could affect their validity.

3.5.1 Does the *unweighted* system outperform the *flat* approach?

Our first research question aims at assessing if a *flat* set of lexemes performs better than the approach based on the zones. In particular, we want to investigate if the lexical information provided in the six different vocabulary leads to better results.

Figure 3.1 shows obtained results after the application of the K-medoid clustering algorithm.

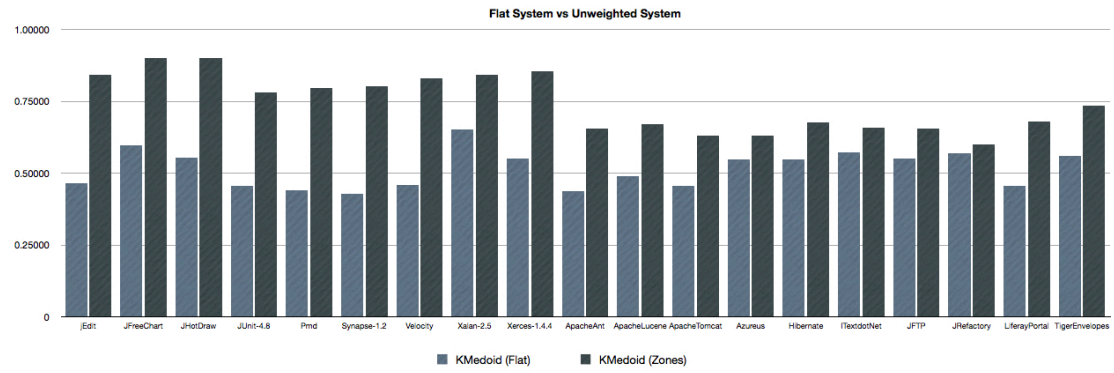

Figure 3.1: Authoritativeness results for RQ1 (Flat system versus the unweighted system) obtained by the application of the K-medoid clustering algorithms

Results report that the introduction of the different vocabularies in the indexing process of the lexical information provides far better results in terms of authoritativeness. Moreover, the same results have been obtained applying the hierarchical clustering algorithm to the two different system configurations.

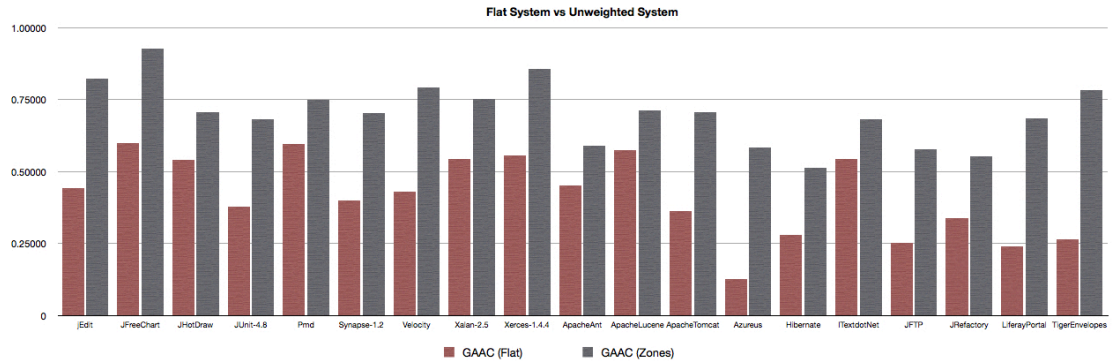

Figure 3.2: Authoritativeness results for RQ1 (Flat system versus the unweighted system) obtained by the application of the HAC clustering algorithms

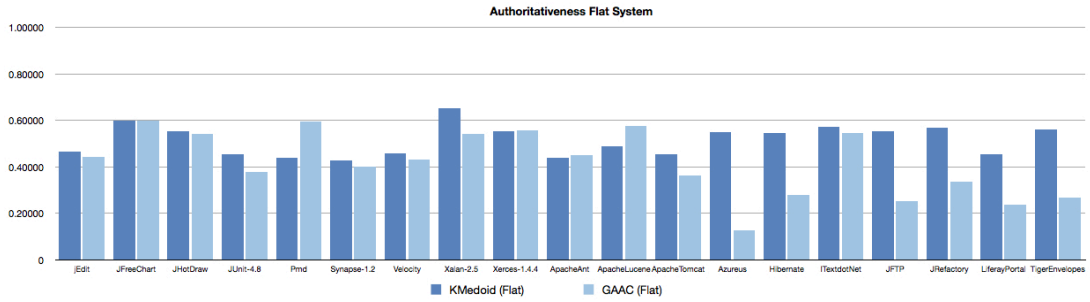

Figure 3.3: Authoritativeness results comparison of the K-medoid and GAAC clustering algorithm on the *flat* system configuration.

Authoritativeness results for the GAAC algorithm are shown in Figure 3.2.

In particular, it is worth noting that in some cases, the results of the GAAC algorithm in the *unweighted* configuration are by far better than results in the *flat* one. This is the case of systems such as Hibernate, jFTP, Liferay Portal, and Azureus, where the improvement in authoritativeness results is up to 5 times better. Conversely, the K-medoid algorithm produces results that tend to be quite acceptable even in the *flat configuration* thanks to the ability of the algorithm in dealing with outliers and in changing different initialisation points.

A point-wise comparison of the two clustering algorithms for the considered system configurations is reported in Figure 3.3 and Figure 3.4.

Results show that, on average, the K-medoid algorithm (the darkest bar in the

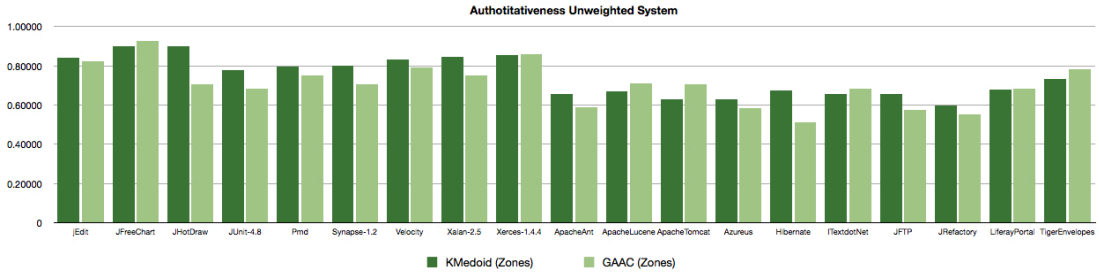

Figure 3.4: Authoritativeness results comparison of the K-medoid and GAAC clustering algorithm on the *unweighted* system configuration.

Figures) produces better results for both configurations.

In conclusion, if we introduce and exploit the six zones, we always got better results than the flat vocabulary, and consequently we can positively answer our first research question. In particular, the results is confirmed despite the selected clustering algorithm.

3.5.2 Does the *complete* system outperform the *unweighted* one?

Once assessed this crucial point, we can start investigating if a smarter combination of the zones can improve results. Such combination is achieved by automatically weighting the relevance of each zone, for each software system, by means of the EM algorithm. Moreover, we are also interested in verifying if the same result is obtained regardless the clustering algorithm applied and the particular probabilistic model on the date assumed.

To this aim, we start the discussion by initially considering the Gaussian distribution as for the model employed in the EM algorithm.

Figure 3.5 shows the results obtained applying the K-medoid clustering in the *complete* system configuration. In particular, it is worth noting that results reported in the bar chart for the *flat* and the *unweighted* configurations are exactly the same discussed for the first research question.

In this case, the boosting in results after applying the automatically generated weights to zones, is less outstanding. However, the quality of the produced clusters is improved or, at least, not worsened.

Conversely, a different scenario is presented, if we consider the application of the

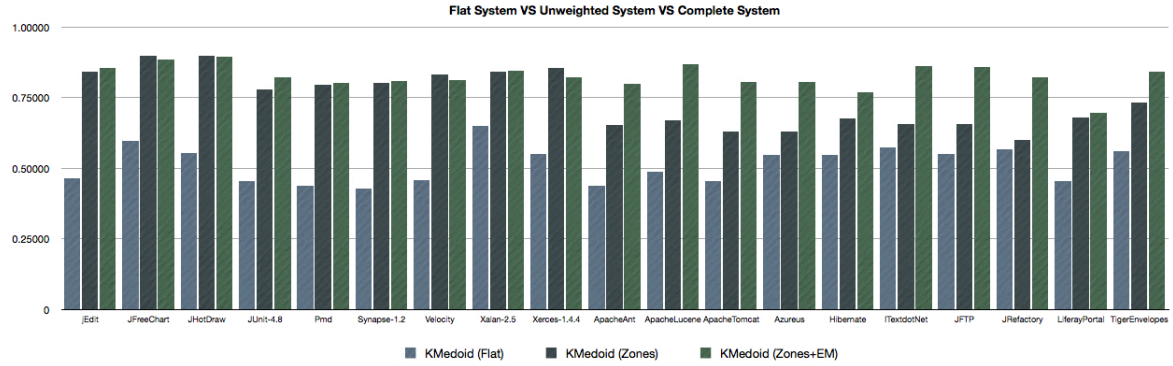

Figure 3.5: Authoritativeness results for RQ2 (Unweighted system versus the Complete system) obtained by the application of the K-medoid clustering algorithms

GAAC algorithm (Figure 3.6). In fact, in some cases, using the weights to zones does not lead to any improvements in results.

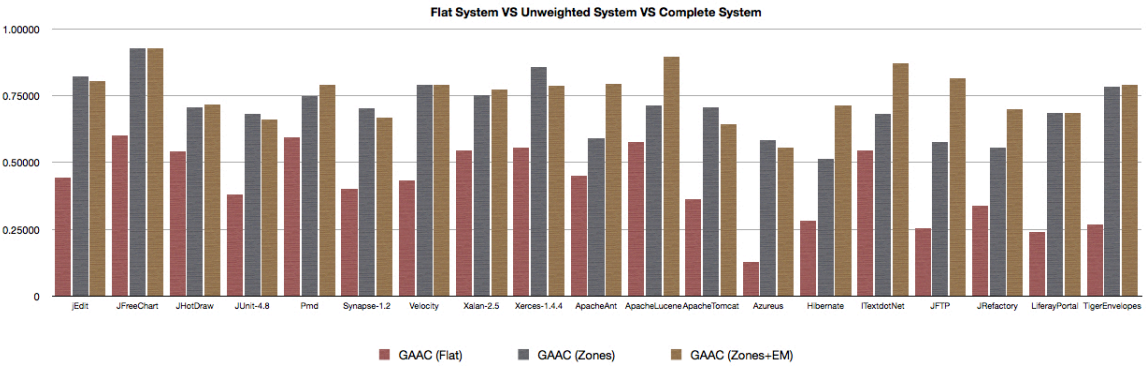

Figure 3.6: Authoritativeness results for RQ2 (Unweighted system versus the Complete system) obtained by the application of the GAAC clustering algorithms

This phenomenon again confirms the limitation of the hierarchical clustering algorithm for software re-modularisation. On the one hand, more advanced cutting strategies must be adopted in order to better partition the agglomerated clusters originally produced by the GAAC.

All the discussed results for the *complete* system configuration, considered a Gaussian distribution as for the probabilistic model assumed on data, and applied during the iterations of the EM algorithm. However, as already mentioned, such model introduces some issues in determining strategies to select the initial

configuration.

As a consequence, to investigate the influence of the different defined probabilistic models on the produced clusters, also the Bernoulli model has been considered in the experimentation.

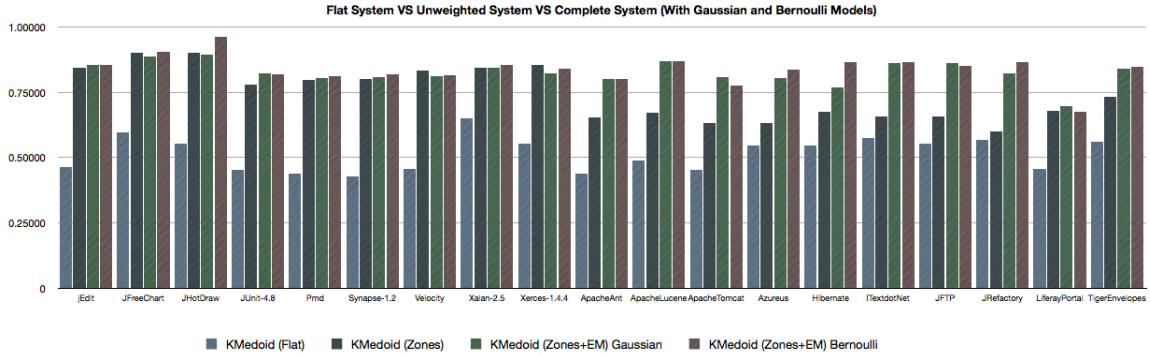

Figure 3.7: Authoritativeness results for RQ2 (Unweighted system versus the Complete system) obtained by the application of the K-medoid clustering algorithms, considering the Gaussian and the Bernoulli models.

As reported in Figure 3.7, the application of the Bernoulli model (the brown bar in the chart) tends to positively affect the quality of the generated clusters. In fact, on the one hand, it confirms the improvements already achieved with the Gaussian model and the K-medoid clustering over the *unweighted* system configuration. On the other hand, this model tends to additionally improve results over those obtained with the Gaussian model (the green bar in the chart). This is the case of systems such as TigerEnvelopes, jRefactory, Xalan and jHotDraw (Figure 3.7).

One possible reason for such improvements is related to some numerical aspects concerning the iterations performed by the EM algorithm during the estimation of model parameters. In fact, as the Bernoulli model does not considers the occurrences of the terms in the documents in its formulation (Section 3.2, its computation does not require to to deal with some numerical issues such as numerical underflows. Conversely, different is the situation for the Gaussian model, where several tricks have been required to guarantee the (numerical) convergence of the EM algorithm.

Thus, in conclusion, we may positively affirm that the the *complete* system

configuration outperform the *unweighted* one, and that best results have been achieved with the Bernoulli model in combination with the K-medoid algorithm.

3.5.3 Threats to Validity

To comprehend the strengths and limitations of the empirical investigation, the threats that could affect the results are presented and discussed. In our case the reliability of the used measures (e.g., Auth) represents a critical issue that may affect the generalizability of the results. In fact, the results could be strongly affected by the used authoritative partition. Also, the lower and upper limits to compute the NED measure may affect the results. Future work will be devoted to adopt different approaches to further investigate this concern.

Another issue is represented by the implicit randomness of the K-medoid clustering algorithm, whose initialisation strategy is performed by selecting randomly elements in the data set as medoids. To reduce biases, we performed 50 runs for each system under investigation. Then we considered the mean NED and Authoritativeness values of these runs.

Also the software systems we have used our empirical study may affect the results. Even if in this work the number of software systems is quite large, and their size is much more variable, it is also true that we exploited only open source software systems. This could threaten the validity of the results. In fact, in contrast with more centralised models of development such as those typically used in commercial software companies, these kind of systems are mainly developed according to a distributed, voluntary collaboration. In these communities, the efforts of a large number of developers are coordinated to create good quality software. This has the positive effect that the used authoritative partitions well approximate the decomposition performed by the original developers. Conversely, the overall quality of these software systems, and in particular of the employed vocabulary, may positively affected the clustering results. To increase our awareness on the presented results, we plan to conduct a further investigation on different commercial software systems.

Finally, we did not perform a comparison of our approach with other clustering based approach on a public dataset. This was due to the lack of such a dataset and

may threaten the generalisation of the results presented in the empirical evaluation presented in the paper. To share with the community the data set used in this work and for replication purposes we have made available on-line an experimental package.

Each problem that I solved became a rule which served afterwards to solve other problems.

Rene Descartes

4

An Efficient Approach to Split Identifiers and Expand Abbreviations

Information Retrieval (IR) techniques are being exploited by an increasing number of tools supporting Software Maintenance activities. Indeed the lexical information embedded in the source code can be valuable for tasks such as concept location, clustering or recovery of traceability links.

The application of such IR-based techniques relies on the consistency of the lexicon available in the different artefacts, and their effectiveness can worsen if programmers introduce abbreviations (e.g: `rect`) and/or do not strictly follow naming conventions such as Camel Case (e.g: `UTFtoASCII`).

Therefore, the processing of the lexical information embedded in the source code requires an additional *normalisation* step in order to automatically split *multi-word* identifiers, and expand possible occurring abbreviations (further details are reported in Section 1.2).

In this Chapter an automatic approach for *source code normalisation*, i.e., *LIN-*

SEN (Linear IdeNtifier Splitting and Expansion), is presented (Section 4.1). The solution is able to deal with both splitting and expansion of identifiers, with the goal of defining a technique intended as a preprocessing step for the wide variety of IR-based software maintenance tools [26].

In particular, LINSEN applies the Baeza-Yates and Perleberg (BYP) algorithm [17] which is an approximate string matching technique, running in linear worst case time if some assumptions are verified. The main advantage provided by such efficiency regards the possibility of exploiting a larger number of dictionaries for the matching. In fact, the approach exploits several dictionaries containing terms gathered from (I) the source code comments, (II) a dictionary of IT and programming terms, (III) an English dictionary, and (IV) a list of well-known abbreviations. These sources are prioritised from the most specific to the most general one, with the idea that in presence of ambiguities, the most specific, domain dependent context should be preferred.

The effectiveness of the proposed approach has been experimentally assessed using 24 software systems mainly implemented in C/C++, and, whenever possible, we compared the achieved results with those reported in three state-of-the-art works [85, 113, 126] (Section 4.2).

Results expressed in terms of accuracy and/or F-Measure [130] show that our approach outperforms state-of-the-art techniques, but the most distinguishing point is that our proposal is by far more efficient (asymptotically), having linear complexity in the size of the dictionary, compared with the cubic one of the solution proposed by Madani et al. [126] (Section 4.3).

4.1 The LINSEN Algorithm

The LINSEN approach aims at finding a mapping between each source code identifier and the corresponding set of *dictionary words*, by exploiting high-level and domain-dependent information gathered from different dictionaries (Section 4.1.1).

The idea underlying the *LINSEN* approach is to adopt a graph-based representation of each input identifier, i.e., the *Matching Graph*, and to apply an approximate string matching algorithm following a two-step process. A description of

Algorithm 6 LINSEN Main Process

Input: `Identifiers` : The list of identifiers to analyse.
Output: `IdMapping` : The structure containing the mapping results for each identifier, i.e., list of dictionary words associated to it.

```
1: function LINSEN(Identifiers)
2:     𝔻 ← {D_1, ..., D_k}                          ▷ The set of adopted Dictionaries
3:     for each: identifier ∈ Identifiers do
4:         tokens ← SplitMatching(identifier, 𝔻)
5:         dictionaryWords ← collectDictionaryWords(tokens, 𝔻)
6:         nonDictionaryWords ← tokens \ dictionaryWords
7:         longForms ← ExpansionMatching(nonDictionaryWords, 𝔻)
8:         IdMapping[identifier] ← dictionaryWords ∪ longForms
9:     end for
10:    return IdMapping;
11: end function
```

such process is depicted in Algorithm 6.

The first step, implemented by the function *SplitMatching* in Algorithm 7, partitions each input identifier into tokens: all the tokens corresponding to dictionary words (i.e., appearing in at least one of the considered dictionaries) are considered correctly mapped and will not require further processing (Line 4-5).

Algorithm 7 Identifier Splitting

Input: `identifier` : An arbitrary identifier;
Input: $\mathbb{D} = \{Dict_1, \ldots, Dict_k\}$: Set of adopted dictionaries.
Output: $\mathtt{L_{match}}$: List of tokens matching the given identifier (`identifier`).

```
1: function SplitMatching(identifier, 𝔻)
2:     φ : φ(word) = 0, ∀ word ∈ 𝔻
3:     for each: D_i ∈ 𝔻 do
4:         L_match ← StringMatching(identifier, D_i, φ, c_SPLIT)
5:         if L_match ≠ NoMatch then
6:             return L_match
7:         end if
8:     end for
9:     return L_match
10: end function
```

Then, all remaining tokens are treated as potential abbreviations and represent the input to the second step (Lines 5-7), the *ExpansionMatching* in Algorithm 8,

further discussed at the end of this section. In particular, Algorithm 8 reports the pseudo code for the single-word short forms expansion. In case of multi-words abbreviations, the process applies a further splitting step, based on the assumption that strings composing a multi word abbreviation in a given file are most likely to occur elsewhere in the same file, or in the same project [61]. In conclusion, for each identifier, the final output is given by the union of the tokens produced by both steps (Line 8).

Algorithm 8 (Single Word) Abbreviations Expansion

Input: `nonDictionaryWords` : List of non-dictionary word tokens;
Input: $\mathbb{D} = \{D_1, \ldots, D_k\}$: Set of adopted dictionaries.
Output: `LongForms` : List of resulting long forms.

```
1: function ExpansionMatching(nonDictionaryWords, 𝔻)
2:     LongForms ← expandKnownAbbr(nonDictionaryWords)
3:     toExpand ← nonDictionaryWords \ LongForms
4:     for each: token ∈ toExpand do
5:         for each: D_i ∈ 𝔻 do
6:             if vowels(token) > consonants(token) then
7:                 φ : φ_PREFIX
8:             else
9:                 φ : φ_EXP
10:            end if
11:            L_match ← StringMatching(token, D_i, φ, c_EXP)
12:            if L_match ≠ NoMatch then
13:                LongForms ← LongForms ∪ L_match
14:                break;                          ▷ Breaks the InnerMost Loop
15:            end if
16:        end for
17:    end for
18:    return LongForms
19: end function
```

The approximate string matching applied in both steps is reported in Algorithm 9. The main part of this algorithm is devoted to the construction of the Matching graph (MG) (Lines 2-8). The graph includes a node for each character in the input identifier, and an edge for each approximate matching with a dictionary word found by the Baeza-Yates & Perlberg (BYP) algorithm [17] (Line 4). Each edge is labeled by the corresponding matched word and its corresponding cost.

Therefore, the MG is defined as a *directed labelled graph* (Definition 2.10).

Algorithm 9 String matching algorithm

Input: $\texttt{token} = \langle \texttt{ch}_1 \cdots \texttt{ch}_N \rangle$: An arbitrary token (identifier);
Input: `Dictionary` : A dictionary of words;
Input: $\varphi(\cdot)$: The tolerance function;
Input: $c(\cdot)$: The cost function.
Output: The list of labels in the minimum cost path.

```
 1: function StringMatching(token, Dictionary, φ(·), c(·))
 2:     G = (V, E) ← initializeMatchingGraph(t)
 3:     for each: word ∈ Dictionary do
 4:         matchingSequence ← BYP(token, word, φ(word));
 5:         for each: (⟨ch_i ... ch_j⟩, word) ∈ matchingSequence do
 6:             G(E) ← G(E) ∪ {⟨(i, j), word, c(word)⟩}
 7:         end for
 8:     end for
 9:     bestPath ← Dijkstra(G)
10:     return getEdgeLabels(bestPath)
11: end function
```

The BYP algorithm exploits a so-called *tolerance function* $\varphi(\cdot)$ to apply an exact multiple pattern matching based on Aho-Corasick automata [7]. In particular, the value of the tolerance function $\varphi(w)$ corresponds to the maximum number of errors allowed in the matching with an input dictionary word w. It is worth noting that too restrictive tolerances would allow only exact matchings, while too slack ones would provide solutions that can be too far from the input to be acceptable.

LINSEN considers two distinct $\varphi(\cdot)$ functions to control acceptable matching errors. In the splitting step, it applies a null tolerance function, i.e., exact matching (Line 2 in Algorithm 7), since it looks for identifying dictionary words composing an identifier. On the other hand, the tolerance functions exploited in the expansion phase bounds the length of possible matching words to be $\varphi(w) = O(|w|/\log|w|)$ (Lines 7 and 9 in Algorithm 8).

These functions guarantees a linear asymptotic complexity [17] with respect not only to the length of the input identifier, but also to the size of the considered dictionary, intended as the sum of the length of its entries.

The cost associated to each edge of the MG is different in the splitting and in the expansion steps and is determined by the length and the total number

of occurrences of each word (in case of the *application-aware* dictionaries - see Section 4.1.1). In particular, this cost is chosen in order to favour longer words and words appearing in *application-aware* dictionaries with respect to broader and generic terms. However, the cost of each word is computed during dictionaries construction and it is accessed in constant time during identifier processing, with no effect on the overall asymptotic complexity.

Finally, the minimum cost path starting in the initial node is built by applying the Dijkstra algorithm (Line 9). The set of labels associated to edges in this path, i.e. matched dictionary words, are returned. In conclusion, the MG can be constructed in linear time with respect to both the length of the input identifier and of the considered dictionary.

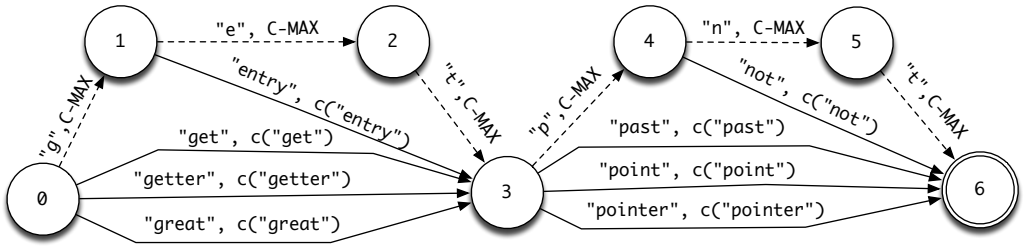

Figure 4.1: Example of *matching graph* for the identifier `getpnt`

Figure 4.1 shows an example of a matching graph for the identifier `getpnt`. Along with the edges corresponding to matched dictionary words, the graph also includes a set of additional edges (Line 2 in Algorithm 9), represented by dashed lines in the figure. Although their function is to ensure that the graph is always connected (Definition 2.17) so that a solution is always produced, their cost is so large that they are included in a solution only when no alternative matching exists.

In conclusion, some more details are required for the description of the *ExpansionMatching* function (Algorithm 8).

This algorithm first checks if the analysed token appears in a list of well known short forms (Line 2) (e.g., acronyms such as *URL* or *XML*) that do not require any further processing (See Section 4.1.1).

Afterwards, each remaining token is processed to identify possible approximate matchings with words in the different dictionaries analysed in an incremental fash-

ion, similarly to the splitting step (Algorithm 7). However, differently from the splitting phase, the algorithm selects the matchings that only requires the set of operations typically applied in shortening words, namely characters deletions.
In particular, according to empirical evidences reported by Madani et al. [126], only intermediate and final deletions are considered (Line 9). These evidences indicate that final deletions are more likely than intermediate ones, and vowels are more likely to be deleted than consonants when in the core of the string. Although excluding initial deletions is not always correct, (e.g., the acronym `XML`, where X stands for eXtensible) the trade off is in favour of such choice.

Another heuristic assumption is based on the hypothesis that short forms with more vowels than consonants are prefixes and in this case only final deletions are allowed (Line 7).

Both these strategies are embedded in the definition of two distinct tolerance functions considered in this phase (Lines 7 and 9 in Algorithm 8).

Similar considerations inspired the definition of the cost function. In fact, both the two types of deletions (i.e. intermediate and final) and the type of deleted characters (i.e. vowels or consonants) are taken into account in the the cost function c_{EXP} (Line 11 in Algorithm 8).

4.1.1 The adopted Dictionaries

Since a given identifier could have multiple and equally correct mapping solutions with different dictionary words, LINSEN considers a set of multiple dictionaries, which convey terms belonging to different *contexts* The idea is to look for the dictionary words matching an identifier name by firstly considering the most specific contexts and then widening the search up to the most general ones.

In particular, the set of considered dictionary contains:

1. $\mathbb{D}_{\texttt{File}}$: a dictionary of terms extracted from the comments of the source file containing the current identifier;

2. $\mathbb{D}_{\texttt{System}}$: a dictionary of terms extracted from the comments of all the source files of the analysed software system;

3. $\mathbb{D}_{\texttt{IT}}$: a dictionary of computer science and programming terms;

4. $\mathbb{D}_{\texttt{ENG}}$: an English dictionary.

The first two correspond to the so called *application-aware* dictionaries [80] that contain terms gathered appearing in comments of the analysed source files. In particular $\mathbb{D}_{\texttt{File}}$ is restricted to terms appearing in the the source file to which identifier belongs, while $\mathbb{D}_{\texttt{System}}$ gathers the terms from comments of the whole system. All these terms are tokenised on non-alphabetical characters, removing any occurrence of *stop words* (e.g.: *the, a, is*). In particular we consider some common English words included in the Apache Lucene system[*], together with a set of proper nouns of length 4 or longer, already exploited by Hill et al. [85]. The number of occurrences of all terms in these two dictionaries are collected during their construction and are then used to compute the edge costs in the MG.

The dictionary $\mathbb{D}_{\texttt{IT}}$ is devoted to contain terms related to computer science and programming, such as *database*, *wireless*, or *applet*. In particular the dictionary contains 22,940 computer science domain terms automatically extracted from 13,647 entries and definitions about Engineering and Information Technology from a glossary available in the Internet[†].

Eventually, for the most general context, $\mathbb{D}_{\texttt{ENG}}$, we consider the *ispell* English dictionary[‡]. This dictionary contains 108,315 words, from which we removed entries corresponding to person and country names and uppercased terms.

Beside these four dictionaries, we also consider a list of widely used English and Information Technology abbreviations, including both the short forms and their corresponding long forms. This list has the purpose to speed up the matching algorithm process as it would be useless and time-consuming to process well-known abbreviations [80].

In fact, this list is exploited in both the splitting and expansion steps to identify well-known abbreviations and associate them to the corresponding long forms, respectively.

These abbreviations have been collected with no prior knowledge about the identifiers to be processed. In particular, English abbreviations correspond to

[*] http://goo.gl/3MZSU
[†] http://www.computer-dictionary-online.org/
[‡] http://www.gnu.org/software/ispell/ispell.html

single and multi-word short forms are taken by a publicly available list[§], discarding the ones containing multiple possible long forms. On the other hand, Information Technology abbreviations have been automatically gathered from the definitions included in the on-line glossary already exploited to build up the $\mathbb{D}_{\text{IT}}$ dictionary.

4.2 Experimental Settings

In this Section, the design of the case study and the dataset used to assess the proposed approach are presented.

In particular, the *goal* of the study is to investigate the ability of the proposed LINSEN approach with the *purpose* of evaluating its ability to map a source code identifier on the (dictionary) words composing it. The *quality focus* is the precision and recall of the approach, as defined by Guerrouj et al. [80], for both splitting and expansion, using freely available oracles. For the sake of comparison with other techniques, the accuracy rate measure will be also considered. The *perspective* is of researchers interested in improving the effectiveness of software analysis/maintenance tools based on IR techniques applied to the source code.

Since LINSEN encompasses multiple processing phases, namely splitting of identifiers and expansion of abbreviations, the empirical study described in this section aims at analysing the effectiveness of the overall approach, and at providing a deeper understanding on the characteristics of each step.

Finally, a preliminary quantitative insight on the computational performances of the LINSEN approach is presented.

The empirical assessment of the proposed approach has been conducted with the following three research questions in mind:

(RQ1) *How does LINSEN compare with state-of-the-art approaches as for the splitting of identifiers?*

(RQ2) *How does LINSEN compare with state-of-the-art approaches as for the mapping of identifiers to dictionary words?*

(RQ3) *What is the ability of the LINSEN approach in dealing with different types of abbreviations?*

[§] http://www.acs.utah.edu/acs/qa_standards/psstd02a.htm

4.2.1 Experiment Design

System	Version	Ids in Oracle	KLOC
JHotDraw	5.1	957	16
Lynx	2.8.5	3,085	174
a2ps	4.14	211	6
which	2.20	487	174
Mozilla-source	1.0	573	4,595
MySQL	5.0.17	194	2,028
Cinelerra	2.0	191	3,533
eMule	0.46	92	262
Quake3	1.32b	80	705
Gcc	2.95	70	1,289
Ghostscript	7.07	66	437
Samba	3.0.0	49	662
Asterisk	1.21	44	459
Minux	2.0	31	334
Mozilla-source	1.3	29	11,458
Mozilla-source	1.2	28	4,681
Mozilla-source	1.4	27	4,710
Mozilla-source	1.1	24	4,676
Httpd	2.0.48	22	558
Azureus	3.0	551	2,682
iText.Net	1.4.1	751	3,380
Liferay Portal	4.3.2	651	3,949
OOPortable	2.2.1	699	3,442
Tiger Envelopes	0.8.9	577	2,647

Table 4.1: Statistics of the analysed systems grouped by the different Dataset to which they belong to (from top to bottom: [126], [113], [114], [85]).

The choice of the works in the related literature to consider for designing the comparative study has been mainly guided by two aspects: (I) the replicability of the experimental settings and (II) the availability of an oracle to qualitative evaluate the results.

Given these criteria, obtained results are compared with those provided by Madani et al. [126] and by Lawrie et al. [113] in investigating RQ1 and RQ2. In particular, in order to provide a more exhaustive empirical evaluation of the

proposed approach, the comparison of LINSEN splitting results has been also conducted against the *LUDISO* oracle [114]. Finally, as for the RQ3, the study conducted by Hill et al. [85] has been selected to investigate the effectiveness of the proposed approach in dealing with the different types of abbreviations appearing in the source code.

The summary statistics of all the considered software systems are reported in Table 4.1.

In order to allow other researchers to replicate our study, in the following as much details as possible on the experimental settings will be provided for RQ1, RQ2 (Section 4.2.2), and RQ3 (Section 4.2.3).

4.2.2 Experimental Settings for RQ1 and RQ2

As for the comparison with Madani et al. [126], results are compared against a manually-built oracle consisting of 957 and 3,085 identifiers extracted from the `JHotDraw` and `Lynx` systems, respectively. The effectiveness of the approach is measured in terms of the *accuracy* rate, defined as the number of the correct results over the total number of inputs. However, since the oracle does not provide any indication on the source file from which each identifier has been extracted, our approach may produce different results according to the specific context where the identifier appears. For instance, in case of the `JHotDraw` identifier `borddec`, the LINSEN approach produces both `b-ord-dec` (error) and in `bord-dec` (correct) as splitting results. Consequently, in the evaluation we measure performance in terms of *worst case accuracy*, where at least a single different splitting result w.r.t. the oracle count as an error (in our example, `borddec` is counted as a misclassification).

Concerning the comparison with Lawrie et al. [113], authors kindly provided the oracle they used in their experimentation. Such oracle contains the splitting and the expansion of identifiers appearing in two software systems, namely `which` and `a2ps`. In particular it contains the list of all the 487 unique identifiers appearing in the former system, and a randomly selected sample of 211 identifiers (among a total of 4,393) for the latter. In this case, the evaluation follows the two-level accuracy rate criteria described by Lawrie et al. [113]: the former, referred as the *identifier-level*, imposes that each expansion must be completely correct. On

the other hand, the latter, the *soft-word level*, gives "partial credit" to each word correctly expanded. It is worth noting that in case of *identifier-level* evaluation, performance are again measured in terms of *worst case accuracy* as no indication on the source files are reported in the oracle.

Finally, in the evaluation of the splitting phase, the *LUDISO* oracle [114] has been also considered, which contains 2,663 identifiers gathered from 750 software systems, together with their splitting results.

However, due to the unavailability of the complete software package, the evaluation has been restricted to the 15 software systems with the highest number of entries in the oracle, for a total of 1,520 identifier (i.e., the 58% of the total identifiers). The performance has been evaluated employing precision and recall, according to the definition reported by Guerrouj et al. [80]. In particular, given an identifier id_i to be processed, indicating by $t_i = \{term_{i,1}, ..., term_{i,n}\}$ the set of terms obtained by the approach, and by $o_i = \{oracle_{i,1}, ..., oracle_{i,m}\}$ the corresponding results in the oracle, precision and recall values are computed as follows:

$$precision_i = \frac{|t_i \cap o_i|}{|t_i|} \qquad recall_i = \frac{|t_i \cap o_i|}{|o_i|}$$

To provide an aggregated, overall measure, the F-Measure (F1) has been further used, which actually computes the harmonic mean of precision and recall:

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

4.2.3 Experimental Settings for RQ3

The assessment of our Expansion step has been evaluated against a gold set containing short forms together with their corresponding long forms. In particular this gold set indicates 250 abbreviations randomly selected from five Java software systems (see Table 4.1), grouped by the different types of abbreviations, namely *prefix* (PR), *dropped-letters* (DL), *acronyms* (AC), *combination words* (CW), *single-letters* (SL) and *others* (OO). Further details about the distribution of the short forms in the different types for the gold set may be found in [85]. Moreover, differently from previous ones, this oracle provides indications to the source file in

which each extracted short form appears. Such extra information allowed us to precisely assess our approach, considering the exact occurrence of the identifier referenced in the oracle.

In more details, the evaluation protocol follows this *methodology*: we fed each identifier into our expansion module assuming that it was produced by a previous splitting phase. Then we compared the resulting output with the gold set following the same evaluation criteria used by Hill et al. [85]: if the produced expansion has the same *stem* than the one appearing in the oracle, then the expansion is considered to be correct. Performance has been evaluated in terms of the *accuracy* rate, calculated for each individual type of short form and for the overall set of identifiers, according to the different distributions of types in oracle.

4.3 Results and Discussions

In this Section, the results for the considered research question are presented (Sections 4.3.1, 4.3.2, and 4.3.3, respectively). A discussion of a quantitative evaluation of the approach, and threats to validity of the empirical investigation conclude the Section.

4.3.1 **RQ1**: How does LINSEN compare with state-of-the-art approaches as for the splitting of identifiers?

As for the first research question, results are compared with those provided by Madani et al. [126] and by Lawrie et al. [113]. It is worth noting that results of the comparison with the Camel Case splitting are not reported, because it is always by far outperformed by any other technique. Even if this can seem a trivial remark, currently the Camel Case splitter is still one of the most used techniques in software maintenance tools for the preprocessing of identifiers. This definitely highlights the importance of new techniques to support IR-based approaches.

From the results of the first comparison reported in Table 4.1 we can note that the accuracy for the `JHotDraw` system are always better than those in `Lynx`, that the spread between the two approaches is smaller on the Java system than on the C one, and that the LINSEN approach performs better than the one proposed by

Madani et al. [126], improving the splitting accuracy of about 5% for `JHotDraw`, and of about 14% for `Lynx`. The conclusions we can draw are that the quality of the naming has a high impact on the splitting step, no matter how good is the used technique, and that our approach turns out to be more robust than the one proposed in [126]. A summary description of these results is shown in Figure 4.1.

As for the comparison with the approach proposed by Lawrie et al. [113], a summary representation of results are shown in Figure 4.2, while their detailed description is reported in Table 4.2.

Here we can see that the LINSEN approach always gathers better results. In particular it improves the splitting accuracy for `a2ps` of about 8% in the Identifier Level and of about 12% in the Soft-word level together with an improvements of about 44% for `which` at both the evaluation levels.

Finally, to provide deeper insights on obtained splitting results, we assess the LINSEN approach considering the previous four systems together with those appearing in the *LUDISO* dataset [114], evaluating results in terms of F-Measure. However, since F-measure values are calculated for each single identifier, instead of only considering the mean value for each analysed system, we decided to represent results by using *box plots*. The box plot reported in Figure 4.3 shows that the me-

System	Unique Ids	DTW	LINSEN
`JHotDraw 5.1`	957	93.1%	94.9%
`Lynx 2.8.5`	3085	70.3%	80.3%

Table 4.1: RQ1: Percentage of correct Splitting compared with Single-iteration results reported in [126].

System	Identifier Level		Soft-word Level	
	GenTest	LINSEN	GenTest	LINSEN
`which 2.20`	58.0%	64.6%	70.0%	78.3%
`a2ps 4.14`	35.0%	50.3%	52.0%	75.1%

Table 4.2: RQ1: Percentage of correct Splitting compared with best results attained by the GenTest Splitting algorithm [113, 115].

dian values of the F-measure are equal to 1.0 for all projects (horizontal segments), while the corresponding mean values are depicted by dots. As a matter of fact,

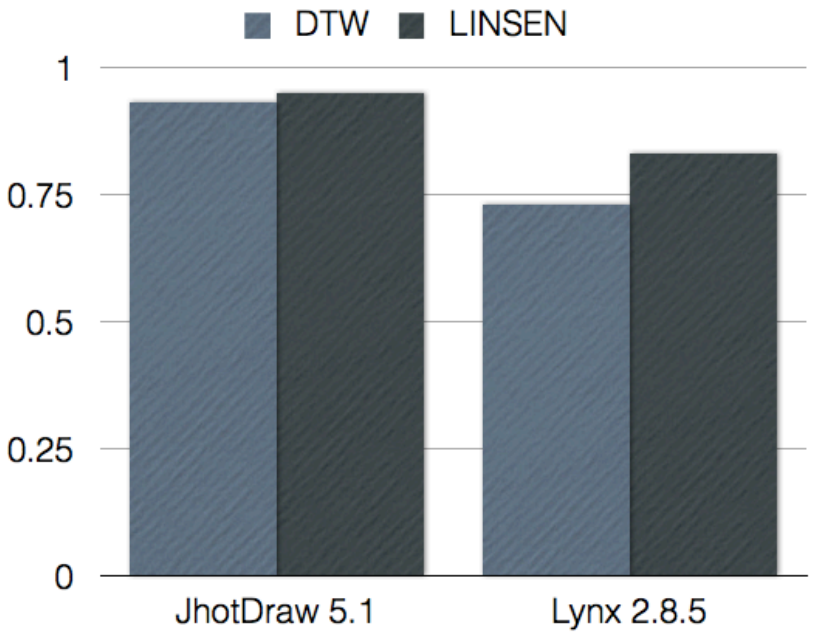

Figure 4.1: Bar Chart of splitting results compared with Single-iteration results reported in [126] (Table 4.1).

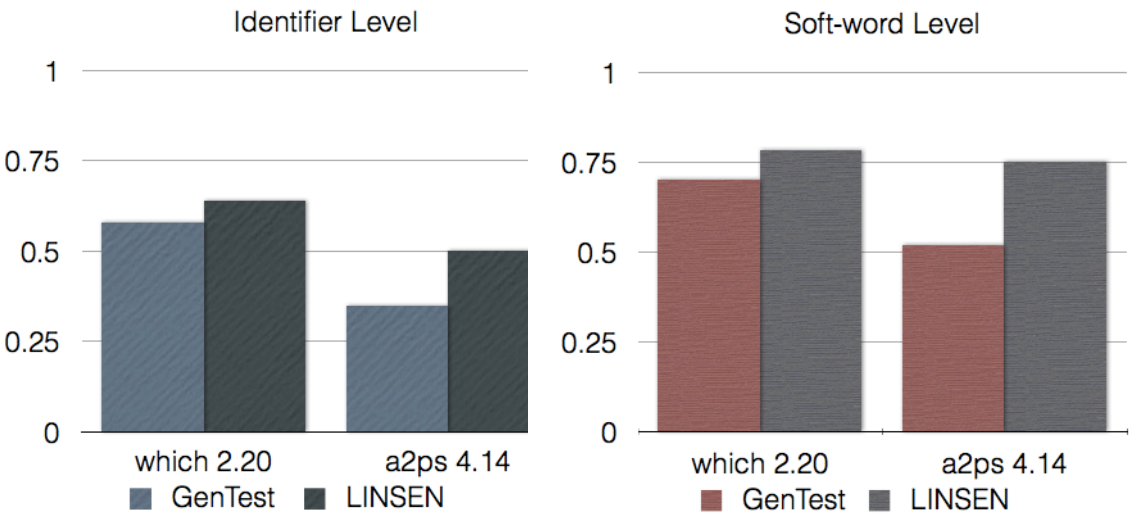

Figure 4.2: Bar Chart of splitting results compared with best results attained by the GenTest splitting algorithm [113, 115] (Table 4.2).

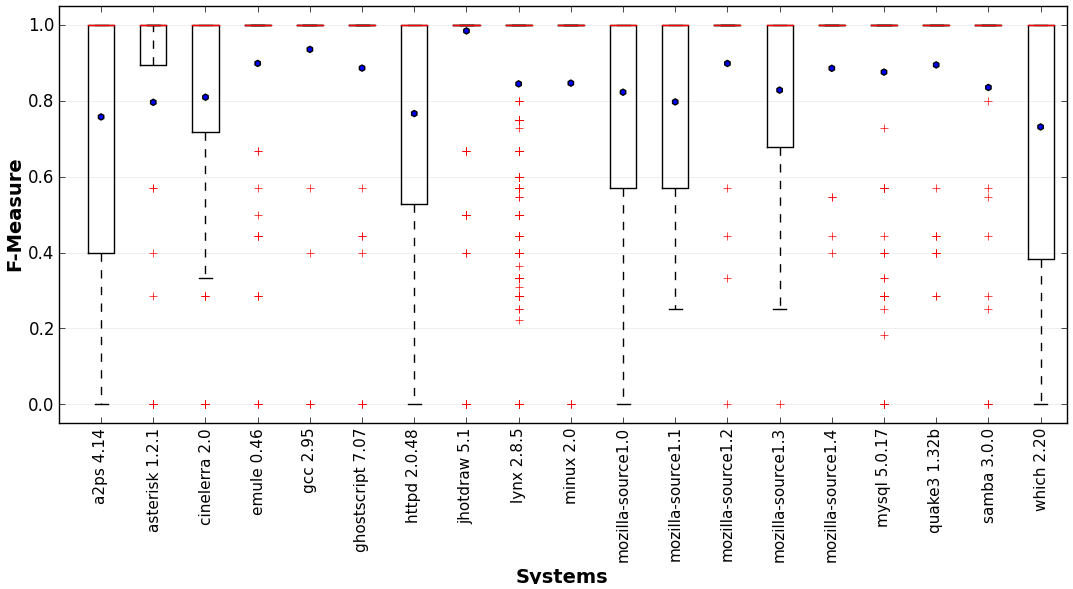

Figure 4.3: Splitting results (F-measure) for systems gathered from [113, 114, 126]

box plots for most systems are very skewed, thus reflecting the good performance of the splitting algorithm included in the LINSEN approach. In particular, such positive trend in results is also remarked by the values of the means (dots) that are all greater than 0.7.

4.3.2 **RQ2**: How does LINSEN compare with state-of-the-art approaches as for the mapping of identifiers to dictionary words?

To address the second research question, we compared the result of the whole LINSEN approach again with those presented by Madani et al. [126] and by Lawrie et al. [113].

Results reported in Table 4.3 show a reduction in the number of errors for both techniques on the two case studies, especially considering the `Lynx` system. This can be also due to the experimental protocol adopted in [126], where the expansion module was fed by only the identifiers not correctly split, and not by the whole list of identifiers. A summary description of mapping results are reported in

In particular we got an improvement of the 3% and of the 2% for the two systems

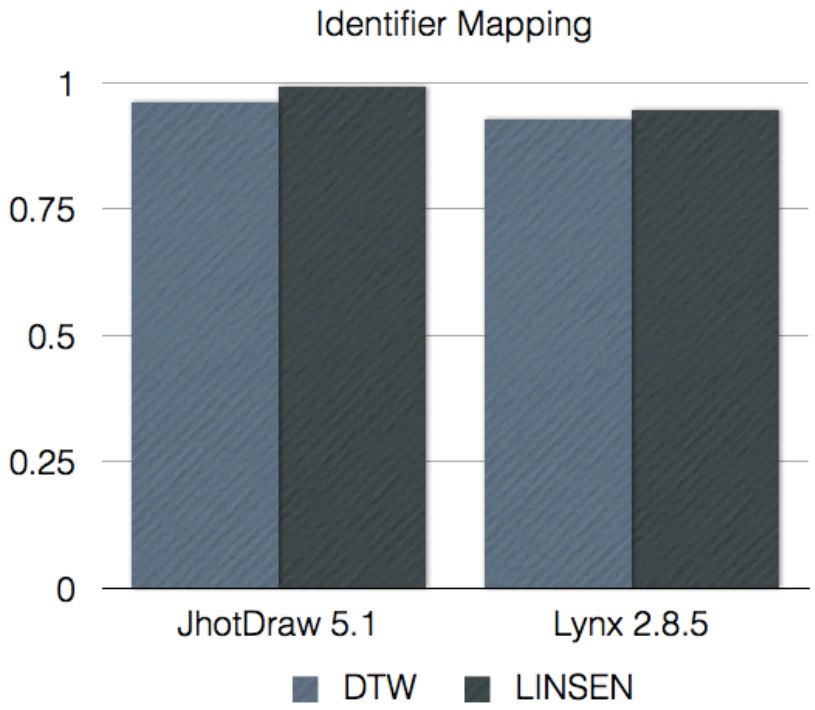

Figure 4.4: Bar Chart of normalisation results (i.e., splitting of identifiers and expansion of abbreviations) compared with results reported in [126] (Table 4.3).

over results proposed in [126], together with an advance of the 1% and of the 17% with respect to our splitting results (RQ1).

System	Unique Ids	DTW	LINSEN
`JHotDraw 5.1`	957	96.1%	99.1%
`Lynx 2.8.5`	3085	92.9%	94.5%

Table 4.3: RQ2: Percentage of Correct Splitting and Expansion compared with results presented in [126].

System	Identifier Level		Soft-word Level	
	Normalize	LINSEN	Normalize	LINSEN
`which 2.20`	53.0%	56.7%	68.0%	71.3%
`a2ps 4.14`	33.0%	56.6%	46.0%	83.5%

Table 4.4: RQ2: Percentage of correct splitting and expansions compared with results presented in [113].

In the first case, results prove the effectiveness of the LINSEN approach. On

the other hand, this empirical evidence enforces our previous conclusion on the influence of the quality of identifiers, pointing out the importance of an abbreviation expansion technique, used in combination with a splitting algorithm to get more accurate results.

As for the comparison with the approach proposed by Lawrie et al. [113] (Figure 4.5), we can see interesting improvements in the results with respect to those presented in the first research question. Here the experimental protocol is different from [126], since the expansion module is always fed by all the identifiers in the source code. Again, the difference in the oracles is the key to understand the pattern in the results, especially when compared with Table 4.3: while expansion results reported in [113] are worse than splitting ones and the same trend is observed in our results for `a2ps`, this is not the case for `which`.

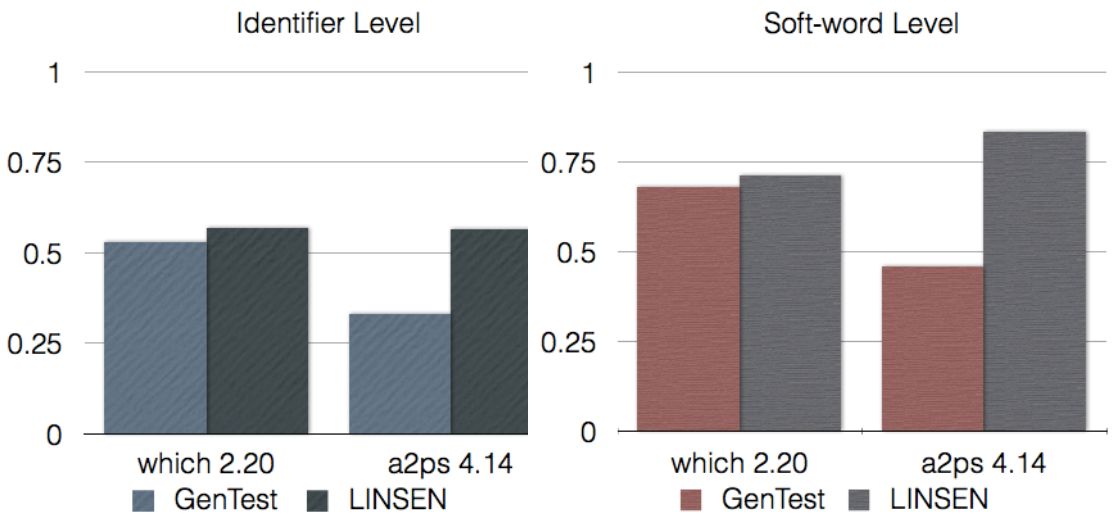

Figure 4.5: Bar Chart of normalisation results (i.e., splitting of identifiers and expansion of abbreviations) compared with results reported in [113] (Table 4.4).

An explanation could be that some identifiers are only partially correctly mapped w.r.t. the oracle. For example, the identifier `S_IRUSR`, standing for *Status Is Readable User*, is expanded by LINSEN in *String Infrared User*, as the substring `IR` appears as the acronym of *Infra Red* in the considered list. In addition, the `which` system contains a lot of identifiers composed by typical C library abbreviations, such as `argv` or `fprintf` that LINSEN maps to *argument value* and *file print file*

instead of *argument vector* and *format print f* respectively. In fact, this trend in results generalises to both levels of evaluations, namely the Identifier Level and the Soft-word Level. However, while LINSEN results for the `which` system are better of the others of more than 7% in the Identifier Level and of about 5% in the Soft-word level, for the `a2ps` the difference is more than 71% in the Identifier Level and more than 81% for the Soft-word Level.

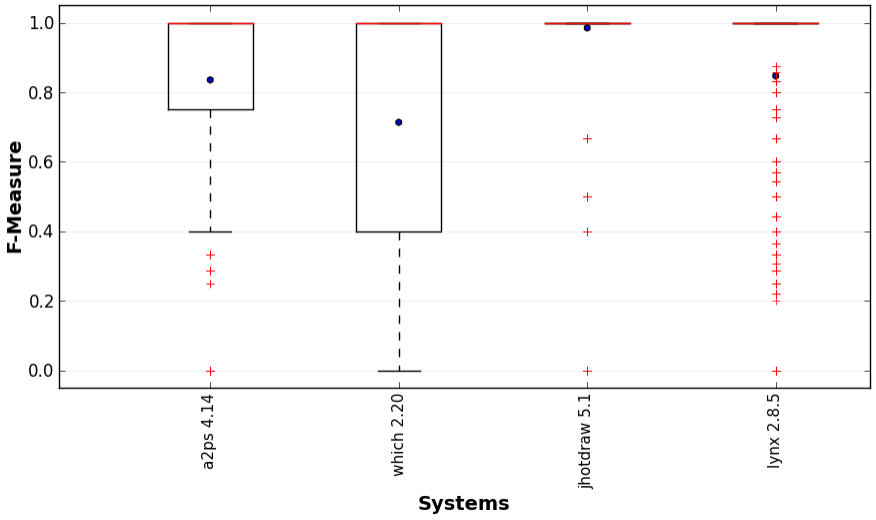

Figure 4.6: Normalisation results (F-measure) for systems considered in [113, 126]

In conclusion, results of the evaluation of the overall approach on all the previous four systems expressed in terms of F-measure are shown in Figure 4.6. Also in this case, box plots show the good performance on the two software systems considered by [126], while graphically remarking issues of the approach on processing with the C-language identifiers appearing in the `which` system.

4.3.3 **RQ3**: What is the ability of the LINSEN approach in dealing with different types of abbreviations?

As for the RQ3, the overall results of the expansion step are reported in Table 4.5, and depicted in Figure 4.7, both organised according to the type of short forms.

Type	% of Types	AMAP	LINSEN
Combination Words (CW)	9.2%	17.4%	**66.7%**
Dropped Letters (DL)	3.6%	**77.8%**	**77.8%**
Others (OO)	18.4%	47.8%	**50.0%**
Acronyms (AC)	19.6%	**46.9%**	36.0%
Prefix (PR)	23.6%	79.7%	**86.0%**
Single Letters (SL)	25.6%	**68.8%**	53.1%
Total (unweighted mean)		56.4%	**61.6%**
Total (weighted mean)		58.8%	**59.1%**

Table 4.5: RQ3: Percentage of correct expansions for each type of short form, compared with results presented in [85].

On average, our approach performs better than AMAP, with an improvement of about 5%. Indeed, although there are two types of abbreviations where AMAP obtains better results, namely Acronyms (AC) and Single Letters (SL), the improvement obtained in the other cases, in particular on Combination Words, compensates such deterioration.

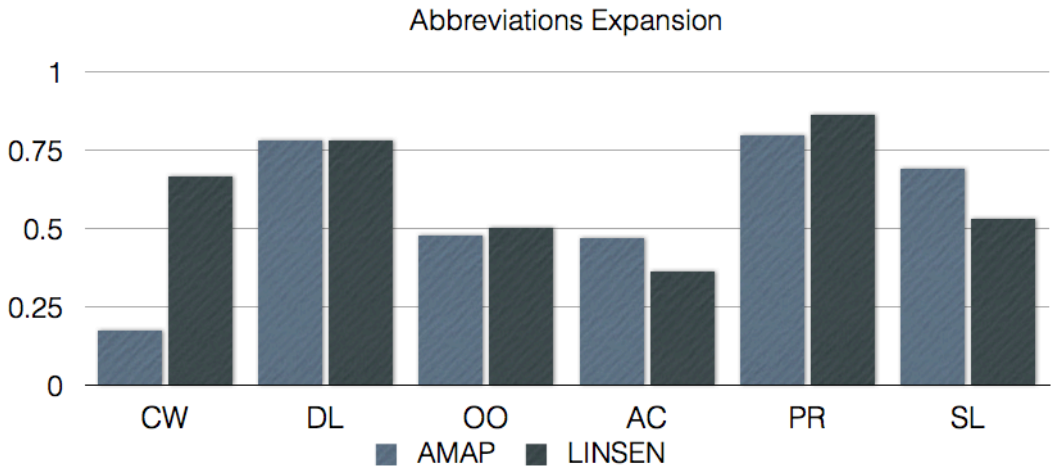

Figure 4.7: Expansion results for systems considered in [85]

In particular, the worse results in the SL case are probably due to the fact that the approach by Hill et al. considers fine grained scopes, totally related to the source code. A similar consideration can be drawn for the AC. Indeed, in our

case if an acronym does not appear in the predefined list, we try to expand it as a generic Multi-word abbreviation. On the other hand, in the AMAP approach, also acronym expansions are totally inferred from the code, and this can be an advantage with very technical abbreviations contained in identifiers. For example, they are able to correctly expand the identifier `DHT`, since in the considered source code it corresponds to a class name (Distributed Hash Table).

4.3.4 Computational Performance

In order to provide a preliminary investigation on how the asymptotic computational efficiency of the BYP algorithm reflects to the actual running time, the logging of the time necessary to LINSEN to split and expand each identifier in each analysed software system. Due to space constraints we are not able to provide detailed data but in our current and quite unoptimised implementation, the overall computation takes on average about 3 seconds for each identifier on a 2.6Ghz AMD Opteron. On the one hand, we judge this result promising if compared with the 8 seconds per identifiers (on average) required by the *Normalize* approach [113]. On the other hand, we think that this running time allows the LINSEN algorithm to be integrated into a Software Maintenance tool, in order to be applied in an off-line computational process.

4.3.5 Threats to Validity

To comprehend the strengths and limitations of our empirical investigation, the threats that could affect the results are presented and discussed. In this kind of works, the bigger threat to *construct validity* (i.e. the relation between the theory and the observation) can arise from errors in the oracles. However we used the same oracles, together with the same versions of software systems exploited in the works we considered for comparison. It is worth noting we found some minor typos in the oracles, that we fixed by adding (rather than replacing) a new possible interpretation of the expansion.

Regarding the *internal validity*, authors in [126] based the evaluation on a two-step experimental protocol: splitting performance are assessed considering the so called *zero-distance* identifiers, i.e., identifiers directly mapped to dictionary

words. Remaining identifiers are further processed in the second (expansion) step and overall performance are finally attained. As a consequence, we used their results of the first step as a benchmark for our splitting phase, and their results of the second step for the evaluation of the splitting and expansion steps. Moreover, as for the comparison of results reported in [113], we considered *GenTest* results in the evaluation of the splitting phase (RQ1) and *Normalize* as for the (RQ2).

As for the threats to *external validity*, potential issue may arise from the software systems we used in our empirical study. Indeed, even if they are written in different languages and their size is quite varying, we exploited only open source software systems. In this context, usually developers are aware that their code will be read by other people, and thus the quality of the employed vocabulary may be higher than those of other development settings. To get a better insight on the presented results, we plan to conduct a further investigation on commercial software systems.

Finally, as for *conclusion validity*, we are aware that our approach can provide splitting and/or expansion that may not reflect the original intent of the programmers. Indeed, our goal was to compare our solution with current state-of-the-art approaches.

If the only tool you have is a hammer, you tend to see every problem as a nail.

Abraham Maslow

5

A Kernel Based Approach for Clone Detection

AUTOMATIC approaches for clone detection intrinsically require the definition of a proper measure to compute the similarity between code fragments (Section 1.4.1) However, as reported in classification of code clones in Section 1.4.2, only Type 1 clones are represented by exactly the same set of instructions, while the other three types involve lexical and syntactic variations between the two fragments.

To cope with this issues, the similarity computation could not rely only on the sole lexical information provided by the code, but additional information should be considered, such as the syntactic structure or instructions dependencies. Furthermore, the applied similarity measure should be able to combine such information in order to produce correct solutions.

To this aim, two Kernel-based solutions for clone detection are presented (Sections 5.1 and 5.2) in this Chapter that exploit different structural representation

of the source code, namely Abstract Syntax Tree (AST) and Program Dependency Graph (PDG) to define effective solutions for clone detection.

Kernel Methods [87, 167] have shown to be effective in approaches considering the similarity between complex input structures such as trees and graphs (see Section 2.4.2). However, these techniques have never been applied for software maintenance tasks, thus representing one of the main contribution of the presented approaches.

In Section 5.1 the description of a Tree Kernel-based clone detection technique is provided.

The proposed approach exploits together syntactical (AST) and lexical information (e.g., name of methods, variables, etc...) for the identification of software clones up to Type 3.

The main drawback of this approach is that it could not be effectively applied in the identification of Type 4 clones, as the definition of similarity it embeds mainly considers the program text of the compared code fragments.

Therefore, to deal with Type 4 clones, the information about the *program behaviour* becomes particularly relevant.

As a matter of fact, most of the clone detection techniques that are able to detect Type 4 clones [71, 101, 106] use Program Dependency Graphs (PDGs) to represent the source code (see Section 1.4.3 for further details).

To this aim, we propose the application of Graph Kernels to PDGs, similarly to what done on AST, to detect meaningful similar subgraphs.

However, as briefly described in Section 2.4.2, the main limitation of such approaches regards the computational effort they require, which is in fact much larger than what is needed by Tree Kernels. Thus, to find a good trade-off between such cost and the information considered in the (sub)graphs comparison, we are focusing on the application of Weighted Decomposition Kernels (WDK) [138] (Section 2.4.2) as they enable to define specific criteria to reduce the total number of comparisons.

An qualitative empirical evaluation has been conducted to asses the validity of the two proposed approaches, considering the *scenario-based evaluation* presented in [159]. Moreover, the Tree Kernel-based solution has been also quantitatively assessed, comparing achieved results with those obtained by another related clone

detection tool, namely *clone digger* [32, 33]

Finally, the Chapter concludes with a proposal of a *mutation-based* algorithms, devoted to automatically generate clones, towards the definition of a *supervised* kernel-based clone detection solution (Section 5.3).

5.1 Tree Kernels for Clone Detection

The key idea underlying our approach is to exploit a Tree Kernel to compute similarities of source code fragments represented by means of ASTs.

In particular, to effectively define a novel Kernel-based solution for clone detection, it is required to identify a proper set of *features* characterising the information of the domain, and to define an appropriate kernel function that is able to exploit the structured nature of data.

All these aspects will be detailed in the next Sections. In particular, in Section 5.1.1 provides a description of the adopted features, while defined kernel functions are detailed in Section 5.1.2).

5.1.1 The Defined Features

The first crucial step to consider in the definition of ML algorithms concerns the definition of an effective representation of the input data, namely the source code fragments, and their corresponding set of features to be exploited in the learning process.

As briefly introduced in the previous section, the basic idea of the proposed approach is to apply the similarity computation on source code fragments represented by means of their corresponding AST.

The AST is a well-known and widely adopted code structure generated by static program analysis algorithms, which is able to provide a tree structured representation of the syntactic information of a fragment of code. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, comments are disregarded, and a syntactic construct like an `if-then` expression may be denoted by means of a

single node with two branches. Furthermore, nested statement blocks correspond to nested (sub)trees (Definition 2.21).

An Example of (partial) AST is depicted in Figure 5.1, together with the corresponding C code function from which it has been generated (Listing 5.1).

```
1  int function (int parameter){
2      int k = 10;
3
4      printf("Hello, this is the function");
5
6      int i = 0;
7      while (i < 7){
8          i += 1;
9          // do something cool
10     }
11 }
```

Listing 5.1: Excerpt of a C code

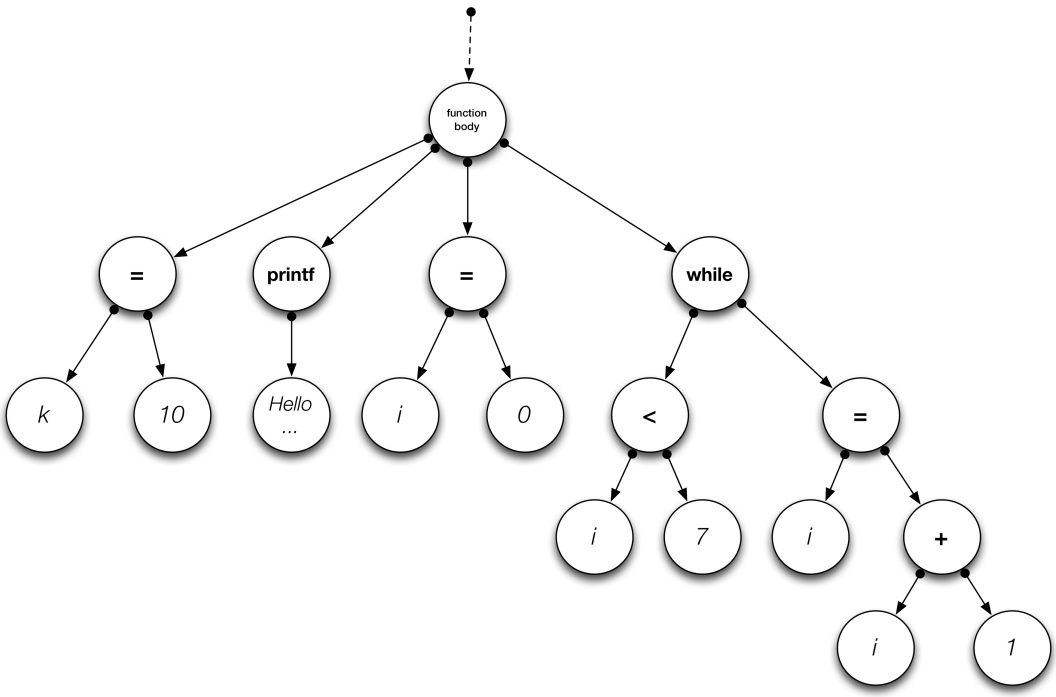

Figure 5.1: Example of a partial AST generated from the above code example.

However, it is worth noting that while *internal nodes* (represented in boldface

in the Figure) convey syntactical information, *leaves* (in light italic in the Figure) provide information about the names of variables and literals, called *lexemes*, used through out the corresponding code

As a consequence, such different kinds of information could be exploited in the similarity computation in order to distinguish the different possible code variants. In fact, code fragments that share the same syntactical and lexical information (Type 1 clones) are different from clones that shares only all or some syntactic information, namely Type 2 and 3 clones respectively.

Therefore, to properly combine such information, the proposed Tree kernel method is based on a set of four different features, which are used to annotate the nodes of the AST, namely (I) IC (Instruction Class); (II) I (Instruction); (III) C (Context); (IV) Ls (Lexemes).

The Instruction Class represents the "class" of the instruction associated to a node, such as `Loop`, `Conditional Control`, `Assignment`, etc. Intuitively, this feature allows discriminating if two instructions, i.e., nodes of the AST, are comparable. For instance, a `while` statement can be compared with `do while` or `for` statements since all belongs to the same class of instructions, i.e., `Loop`. On the contrary, the same set of instructions are not comparable with an `if` statement since the latter belongs to a different class of instructions, namely `Conditional Control`.

The next feature is Instruction. It contains the token provided by the lexer during the parsing process. This is the typical information considered by AST-based approaches [20, 104].

Table 5.1 reports about some example instances of the features Instruction Type and Instruction regarding the Java programming language.

The Context feature indicates the Instruction Class of the statement in which the node is contained. The rationale behind this feature is to increase the similarity value of two nodes if they appear in the same Instruction Class.

For example, considering the code reported in Listing 5.1, nodes corresponding

Language Instruction	Instruction Type	Instruction
`for`	`Loop`	`for-loop`
`while`	`Loop`	`while-loop`
`if`	`Conditional-Control`	`if-statement`
`switch`	`Conditional-Control`	`switch-statement`

Table 5.1: INSTRUCTION TYPE and INSTRUCTION features examples.

to the assignment expression `i += 1` (Line 8) will be assigned `Loop` as for their CONTEXT feature, since the whole expression appears in the body of a (`while`) loop statement. Thus, when comparing two different code fragments, in case of exactly the same instruction (the assignment expression in this case), the similarity computation will take into account the two *context* of the code where the expressions appear.

Nevertheless, the computation of this feature is based on the whole AST and therefore requires a post-processing phase after ASTs construction.

Finally, the LEXEMES feature associates to each node the set of lexemes of the subtree rooted in that node. This allows taking into account also the lexical information in the computation of the similarity among (sub)trees. In more details, the LEXEMES feature is defined recursively:

(**Leaf Nodes**): Ls corresponds to the lexeme associated to the node itself;

(**Internal Nodes**): Ls corresponds to the union of all the lexemes associated to its subtrees with the minimum height.
In particular, only the information of the minimum height subtrees is considered since they should convey the closest lexical information for an internal node. Moreover, this limits the amount of lexemes percolated through the tree.

Figure 5.2 shows an example of features annotation considering some of the nodes in the AST depicted in Figure 5.1. In particular, some values are worth mentioning in order to better clarify the role of each of the defined features.

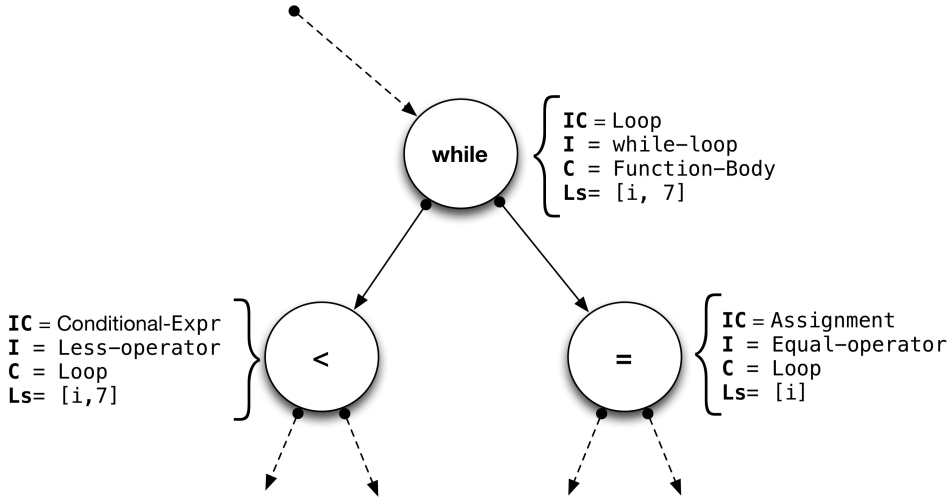

Figure 5.2: Associated features to node of the AST reported in Figure 5.1.

In particular, let us note that the CONTEXT of the `while` node correspond to `Function-Body` is the loop is not nested in any other block or statement. Differently, the context of the nodes labeled with $<$ and $=$ is equal to `Loop` as expected. Furthermore, the LEXEMES associated to the `while` node, contains the terms i and 7 that have been propagated from the leftmost (and nethermost) subtree (See Figure 5.1).

5.1.2 The Defined Tree Kernel

Tree Kernels evaluate similarity between two trees in a recursive fashion: *(I)* computing similarities among nodes; *(II)* aggregating up this information. Thus, they require a similarity measure on the nodes and a function that traverses recursively the tree combining these similarity values. In our approach, we define two primitive kernel functions m and k that operate on single nodes in terms of their features. To combine results on the overall tree structure, we define the Tree Kernel K.

The binary function m (Eq. 5.1) determines whether two nodes (n_1 and n_2) are comparable according to their INSTRUCTION CLASS. The function k (Eq. 5.2) defines a value of similarity between compared nodes according to remaining three features, namely INSTRUCTION, CONTEXT and LEXEMES.

Please note that we use the notation $n.f$ to denote the feature f (e.g., IC or Ls) associated to the node n.

$$m(n_1, n_2) = \begin{cases} 1 & \text{if } n_1.IC = n_2.IC; \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

$$k(n_1, n_2) = \begin{cases} 1 & \text{if } \begin{cases} n_1.I = n_2.I & \text{and} \\ n_1.C = n_2.C & \text{and} \\ n_1.Ls = n_2.Ls \end{cases} \\ 0.8 & \text{if } \begin{cases} n_1.I = n_2.I & \text{and} \\ n_1.C = n_2.C \end{cases} \\ 0.7 & \text{if } \begin{cases} n_1.I = n_2.I & \text{and} \\ n_1.Ls \cap n_2.Ls \neq \emptyset \end{cases} \\ 0.5 & \text{if } \begin{cases} n_1.C = n_2.C & \text{and} \\ n_1.Ls \cap n_2.Ls \neq \emptyset \end{cases} \\ 0.25 & \text{if } \begin{cases} n_1.C = n_2.C & \text{or} \\ n_1.I = n_2.I & \text{or} \\ n_1.Ls \cap n_2.Ls \neq \emptyset \end{cases} \\ 0 & \text{otherwise (no match)} \end{cases} \tag{5.2}$$

Two nodes have maximal similarity (i.e., 1) if all the involved features have the same values, while their similarity is 0 if they are totally different. In case two nodes have some differences in their features, structural similarities are more important than the lexical ones for detecting clones. Accordingly, their similarity is 0.8 if they differ only in their lexemes lists. The similarity is 0.7 if the nodes are in different contexts and share lexical information and are the same instruction. If two nodes are within the same context and share some lexemes, but are two

different instructions their similarity is 0.5. Finally, if they share only one feature value, their similarity is 0.25.

These similarity values has been chosen according to a pilot experimentation conducted on the original system used in the case study. This may represent a possible threat for the empirical assessment of the approach. Therefore, we are going to investigate first the effect of using different similarity values and then define some heuristics to properly choose these values.

The Tree Kernel K_{AST} is defined on the subtrees T_1 and T_2, whose roots are r_1 and r_2, respectively:

$$K_{AST}(T_1, T_2) = m(r_1, r_2) \cdot \left\{ k(r_1, r_2) + \sum_{n_1 \in \texttt{Ch}(r_i)} \max_{n_2 \in \texttt{Ch}(r_j)} K(t(n_1), t(n_2)) \right\} \quad (5.3)$$

where $t(n)$ indicates the subtree rooted in n, while $\texttt{Ch}(r_i)$ denotes the list of the children of the node r_i. The index i, at each step, corresponds to the node between r_1 and r_2 having less children: in this way, the function K is symmetric.

The rationale for defining the function K relies on the fact that we are interested in the identification of the maximum isomorphic subtrees. Since the size of two subtrees is not constant, their similarity value must be normalised in the range $[0, 1]$:

$$K_{norm}(T_1, T_2) = \frac{K_{AST}(T_1, T_2)}{\sqrt{K_{AST}(T_1, T_1) \cdot K_{AST}(T_2, T_2)}} \quad (5.4)$$

Also K_{norm} is symmetric, and it is a Kernel [167]

5.1.3 Experimental Settings

The description of the experimental settings is divided in two different parts, one for each step considered in the evaluation protocol. The first part (Section 5.1.3) is devoted to the description of the so-called *scenario-based evaluation* [159] which is preliminary performed in order to get useful insights and indications on the defined similarity measure. On the other hand, the second part (Section 5.1.3) is devoted to the description of the conducted comparative study.

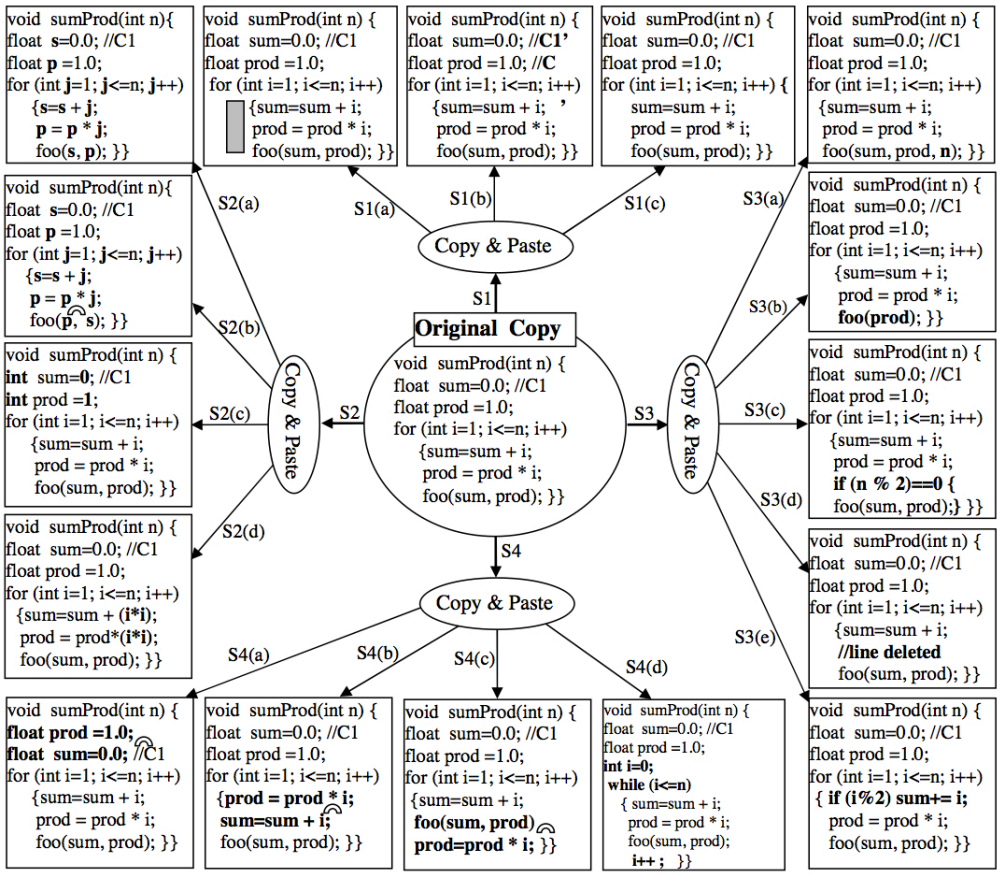

Figure 5.3: Editing Taxonomy Scenarios (extracted from [159])

First of all it is worth noting that the following evaluation protocol has been also considered in the assessment of the Graph-kernel approach described in the Section 5.2, and thus its description will not be further replicated.

In the survey by Roy et al. [159], authors proposed a qualitative approach to compare and evaluate almost well-known existing clone detection techniques, called *scenario-based evaluation*. In particular, such evaluation strategy defines a top-down editing taxonomy based on a set of hypothetical program editing scenarios that are representative of typical changes to copy/pasted code [191].

All the 16 proposed scenarios are represented in Figure 5.3.

The defined scenario are categorised in 4 different groups, corresponding to

	Systems			
	Cleaned	**Type 1**	**Type 2**	**Type 3**
Methods	351	423	422	433
Methods per Class	6.38	7.82	7.84	8.40
KLOC	7.6	8.8	9.4	8.9

Table 5.2: Descriptive Statistics.

the four Types of code clones. Then, each scenario comprises possible editing transformations to source code to generate clones according to those accepted in the definition of the corresponding clone Type. For example, in case of the editing Scenario $S1$, corresponding to Type 1 clones, only changes to layout and to comments are applied. On the other hand, in case of Scenario $S3$, added or changed statements are also considered.

As a result, authors compared the performance of each clone detection techniques in each scenario, and obtained results show that Scenario $S1$ is the easiest one, while, as expected, Scenario $S4$ is hard for most techniques [159]. Moreover, no existing techniques can perform very well in all the given scenarios [191].

Comparative Study

Although our approach is general and language-independent, the implemented prototype detects clones at *method* level and supports systems written in Java.

To assess the approach and the tool prototype, we considered a Java software system we are particularly familiar with. This system is a typical academic application, developed at the University of Naples "Federico II" by a Master student in Computer Science.

The software system has been checked against the presence of up to Type 3 clones by one of the authors. The tool CloneDigger [32] and our prototype have been used to check the presence of clones. All the detected clones have been removed through refactoring operations. This produced the cleaned software system used in the case study.

We manually and randomly injected in the code of the cleaned software system some artificially created clones, thus creating three mutations of the cleaned soft-

ware system. Successively, we verified whether our tool was able to detect all the injected clones. Some descriptive statistics of the cleaned system and the mutated ones are reported in Table 5.2. It is worth mentioning that we did not analyse the original system since it contained a few numbers of clones, thus making it not meaningful for assessing the proposed approach.

Furthermore, we applied CloneDigger to get an indication on whether our approach outperform an AST based approach. We considered here clones at the method level taking into account only methods that have at least one instruction, as in [157].

In this preliminary investigation we considered three different experimental trials, one for each type of clone. Thus, in each trial we injected only one type of clones at a time and then applied the two detection tools. Regarding Type 3 clones, we considered both the definitions of Structure-substituted clone and Modified clone [175].

We used the *precision* measure to assess the *correctness* of the results, while the completeness has been estimated by employing the *recall* measure. Precision is the fraction of real clones identified among all the retrieved ones, while recall is the fraction of real clones among all the actual ones:

$$precision = \frac{\#(\text{actual clones identified})}{\#(\text{total candidates clones identified})} \tag{5.5}$$

$$recall = \frac{\#(\text{actual clones identified})}{\#(\text{total actual clones})} \tag{5.6}$$

Since precision and recall measure two different concerns, we used F-measure (F_1) to get a trade-off between the correctness and completeness:

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{5.7}$$

F_1 gives the same relevance to correctness and completeness and may be intended as an estimation of the authoritativeness of the clones identified by the approach (i.e., how much automatically identified clones resembles the group of clones identified by an expert).

5.1.4 Results and Discussions

In the following the obtained results for the qualitative and quantitative evaluation of the Tree Kernel based technique are presented.

Results obtained for the scenario-based evaluation of the approach are reported in Figure 5.4

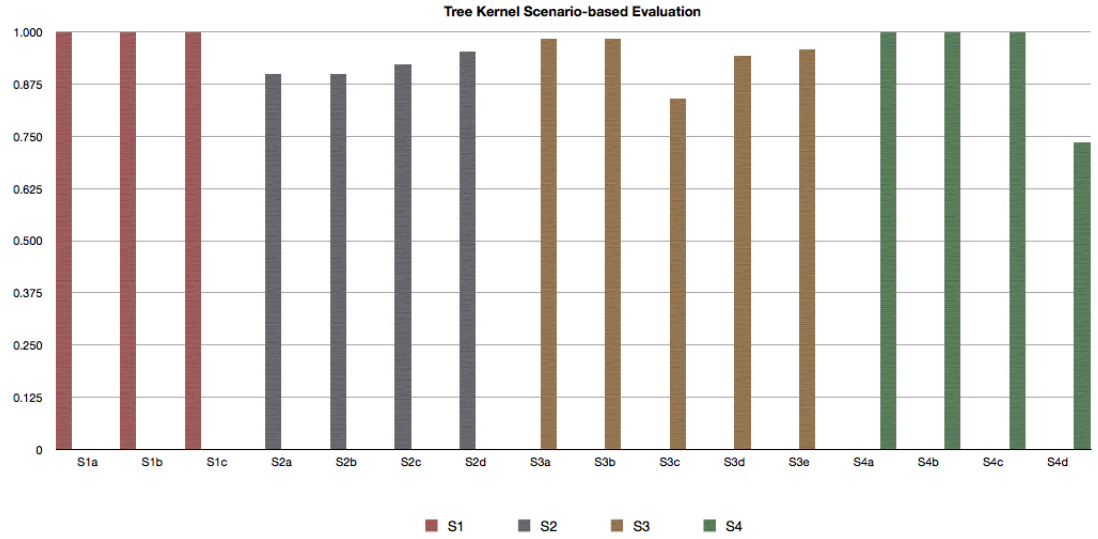

Figure 5.4: Results for the qualitative evaluation of the proposed Tree Kernel K_{AST} clone detection technique.

Obtained results show the effectiveness of the proposed technique in dealing with all the considered scenarios. Duplications in the first two Scenarios, namely $S1$ and $S2$, are all correctly identified as clones with returned similarity values ranging from 0.9 up to 1.0. Moreover, very good results have been also obtained for the other two Types of clones, i.e., Type 3 and Type 4., showing good potentials in correctly identifying clones in almost every considered scenario.

Nevertheless, it is worth noting that the considered scenarios for Type 4 clones do not take into account in their definitions the extended classification proposed by Kim et al. [99]. Moreover, the first three ones, namely $S4a$, $S4b$, and $S4c$, correspond to very trivial modifications and are not very good representatives for "real" Type 4 clones. As a matter of fact, the presented approach assigns a similarity value equals to 1.0 for all three of them. This is because the definition

of the K_{AST} Kernel function intrinsically takes into account the case of statement inversions during the traversal of the compared tree structures.

Conversely, the worst result has been obtained with the *S4d* Scenario, corresponding to the Type 4 clone that involves replacements with semantically equivalent control structures, also indicated as *Type4a* [99].

Therefore, the presented qualitative assessment confirms our expectations on the very good potentials of the proposed Tree Kernel technique in dealing with clones up to Type 3.

Furthermore, these expectations have been also met in results achieved in the quantitative evaluation.

In particular, for the Type 1 clones our tool and CloneDigger were able to detect all the clones without any false positive. So for both the tools we got $F_1 = 1$. The same holds for Type 2 clones. Thus, both the tools expressed the potential of the AST-based approaches. In fact, independently from the used identifiers and literals they were able to detect all the methods with the same syntactic structure.

Some further considerations are needed for Type 3 clones. In particular, it is worth mentioning that it cannot be easy to find a crisp boundary to identify Type 3 clones, since the definition does not specify a limit in the number of differences that could exist between two fragments of code. To deal with this issue, we imposed different threshold values on the minimum similarity necessary to classify two code fragments as clones. This parameter can be easily tuned by a software engineer during the clone detection process.

Figure 5.5 summarises the results achieved to detect Type 3 clones by using different thresholds values. The results indicates that the choice of the threshold values strongly affects the detection performance. Indeed, high threshold values may miss clones with a large number of modifications and causes a drop in recall. On the other hand, for values larger than approximately 0.7 the precision is close or equal to 1, as such conservative approach is not affected by false positives. Conversely, lower threshold values improve the recall, but at the cost of introducing false positives. The best trade-off between precision and recall corresponds to the maximum in F_1 (i.e., 0.74) and is achieved for a threshold value equal to 0.7.

Worse results were observed by applying CloneDigger on our data set. In par-

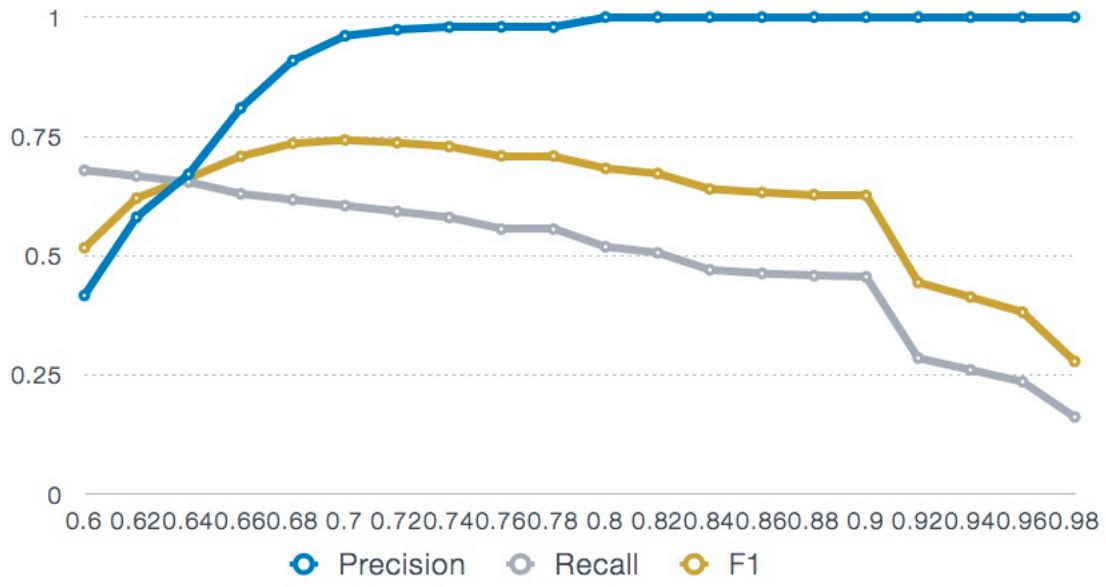

Figure 5.5: Precision, Recall and F-Measure plot of achieved results for Type 3 clones

ticular, on the Type 3 clones we obtained: precision = 0.35, recall= 0.41, and F_1=0.38. These values are far from our results and have been obtained after a heavy tuning of the many parameters of the considered clone detector. A description of obtained results are shown in Figure 5.6.

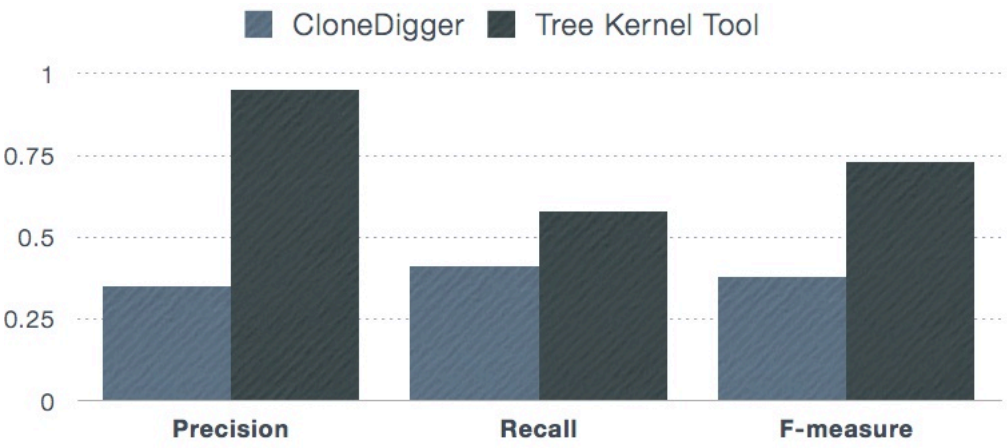

Figure 5.6: Bar chart summarising obtained results on Type 3 clones by CloneDigger [32] and Tree Kernel Clone Detector.

A possible drawback of Tree Kernels is the running time. This suggested us to accomplish a further investigation to compare the performances between our tool and CloneDigger. To this end, we ran both the detectors, on the cleaned system and the different mutations, on a laptop equipped with a 2Ghz Intel Core 2 Duo process with 4 GB of RAM.

The execution time of CloneDigger took about 10 seconds on average, while 30 seconds was the average time of our tool. It is worth noting that the computation time of our tool is nearly the same on each mutated system.

To preliminary assess the scalability of our detector, we also used it on a bigger Java project, namely the Eclipse-jdtcore (about 150K LOC). The needed time to analyse this system was about 10 minutes. This is acceptable in case the detection is performed off-line.

Threats to Validity

Construct validity threats concern the relationship between theory and observation. Precision, recall, and F-measure well reflect the performance of our approach. However, the issue may be concerned to the systems used to assess the approach. We tried to mitigate this threat by injecting clones in a random and controlled way in a software system on which we were particularly familiar on.

Regarding the external validity, an important threat is related to the size of the software system and to the fact that it was developed by a student. The rationale for selecting this system relies on the fact that we could easily have an oracle to evaluate the cloning results. Future work will be devoted to assess the approach on larger software systems, thus also assessing its scalability. The evaluation of the approach on commercial software systems represents a possible future direction for our work. The assessment of the approach on public benchmarks [21, 157] also represents a possible future direction.

5.2 Graph Kernels for Clone Detection

The second proposed technique for clone detection leverages the flexibility provided by the WDK (Graph) Kernel formulation (Definition 2.34) to define a novel Kernel-based function able to compute the similarity of code fragments represented by

means of a PDG. In particular, Sections 5.2.1 introduces the main definitions of the PDG and the formulation of the proposed Graph Kernel. Then, Section 5.2.2 describe the preliminary empirical evaluation conducted to assess the presented technique.

5.2.1 WDK for Dependency Graphs

A *Program Dependency Graph* is a graph representation of the source code of a procedure [67]. Basic statements, such as variable declarations, assignments, and procedure calls, are represented as nodes in the PDG. Moreover, every node has a single associated *type*, corresponding to the particular code structure it refers to. The list of types for the nodes in the PDG is reported in Table 5.1, which illustrates how the source code is decomposed and mapped to program nodes [124]. On the other hand, edges in the graph correspond to *data* and *control* dependencies. In particular:

Definition 5.1. (Control Dependency Edge) *[124]: There is a control dependency edge from a "control" node to a second program node if the truth of the condition controls whether the second node will be executed.*

Definition 5.2. (Data Dependency Edge) *[124]: There is a control dependency edge from program node n_1 to n_2 if there is some variable* `var` *such that:*

- *n_1 may be assigned to* `var`*, either directly or indirectly through pointers.*
- *n_2 may use the value in* `var`*, either directly or indirectly through pointers.*
- *There is an execution path in the program from the code corresponding to n_1 to the code corresponding to n_2 along which there is no assignment to* `var`*.*

Definition 5.3. *[124]: The* **Program Dependency Graph** *G for procedure P is a 4-tuple element $G = (V, E, \mu, \delta)$, where:*

- *V is the set of program nodes in P;*
- *$E \subseteq V \times V$ is the set of dependency edges in P;*

Type	Short description
`assignment`	Assignment expression
`increment`	Increment expression $(++, +, --, \ldots)$
`return`	Function return expression
`expression`	General expression except return, assignment or increment.
`declaration`	Declaration of a variable or a formal parameter
`jump`	Goto, break, or continue
`call-site`	Call to other procedures.
`control`	Control structures (loops, conditionals)
`switch-case`	Case or Default
`label`	Program labels

Table 5.1: Types for nodes in a PDG (Adapted from [124])

- $\mu : V \to S$ *is a function assigning the types to program nodes, i.e.,* $n \mapsto \mu(n) = t_n \ : \ t_n \in \{\texttt{call} - \texttt{site}, \texttt{return}, \ldots\}$;
- $\delta : E \to T$ *is a function assigning dependency types, either data or control, to edges, i.e.,* $e = (n_1, n_2) \mapsto \delta(e) = t_e \ : \ t_e \in \{\texttt{data}, \texttt{control}\}$.

Therefore, a PDG is a directed, labelled graph (Definition 2.9) which represents the data and control dependencies *within* one single function. In other words, the PDG represents how the data flows between the statements, and how statements control or are controlled by other statements [124].

Figure 5.1 shows the PDG corresponding to the C function reported in Listing 5.1. In particular, program nodes are labeled with their corresponding *type*. Moreover, *data* and *control* edges are shown in solid and dashed lines, respectively.

Therefore, every node and edge in a PDG is intrinsically featured by a corresponding type. These types will represent the features associated to elements of the graph and considered by the defined kernel functions.

The formulation of the WDK requires the definition of two kernel functions, namely the *selector* and the *contexts* (see Definition 2.34). In particular, the aim of the selector function is to filter the different comparison between all the possible substructures, while the context has the purpose of traversing the substructures for further comparison.

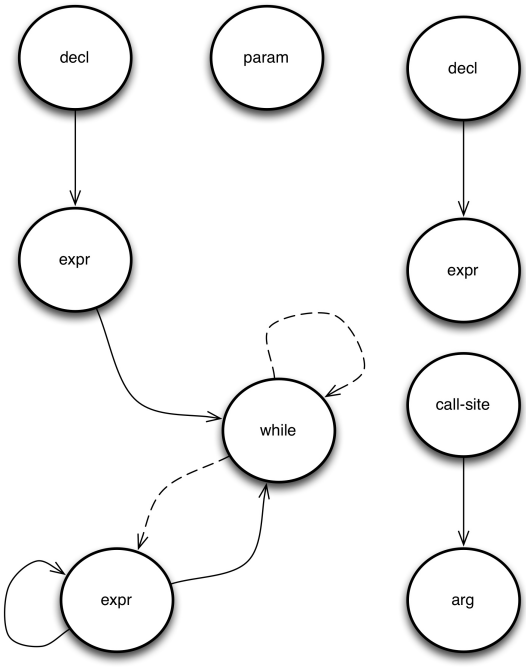

Figure 5.1: Example of the PDG corresponding to the function reported in Listing 5.1. Nodes are labelled with their corresponding *types*, while *data* and *control* edges are depicted in solid and dashed lines

.

Similarly to what defined for the Tree Kernel function, we define the the selector kernel function $\widehat{\delta}^*$ as a filtering function based on the type of the two compared nodes. More formally:

$$\widehat{\delta}(n_1, n_2) = \begin{cases} 1 & \text{if } \mu(n_1) = \mu(n_2); \\ 0 & \text{otherwise.} \end{cases} \tag{5.8}$$

On the other hand, the formulation of the *context* kernel function requires some other preliminary definitions:

Definition 5.4. (Neighbourhood Types Set) *Let n be a program node of a PDG. The* **neighbourhood types set** *of the node n is given by the set of all the*

*We indicate with $\widehat{\delta}$ the selector kernel function considered in the WDK formulation to not lead to confusions when referring to the neighbourhood of a node of the PDG.

types of the nodes in the neighbourhood of n, i.e.,

$$\mathrm{M}(\delta^{+}(n)) = \{\mathrm{M}(u), \forall u \in \delta^{+}(n)\} \tag{5.9}$$

Definition 5.5. (Neighbourhood restricted to a type) *Let n be a program node of a PDG, and T a given program node type. The* **neighbourhood** *of the node n* restricted to *the type T corresponds to:*

$$\delta^{+}(n)_{[T]} = \{u \mid \forall u \in \delta^{+}(n) \wedge \mathrm{M}(u)\} \tag{5.10}$$

Proposition 5.1. (Program Nodes Intersections)*:*
Let V and $V^{'}$ the sets of program nodes of two different PDGs G and $G^{'}$. The **intersection set** *of their two program nodes is defined as:*

$$V_{\cap} = V \cap V^{'} = \{n \in V, n^{'} \in N^{'} \mid \sigma(n, n^{'}) = 1\} \tag{5.11}$$

where σ corresponds to the selector kernel function defined in Equation 5.8.

Remark 5.1. *Proposition 5.1 leads to the following formulation:*

$$\forall n \in V, n^{'} \in V^{'}, \mathrm{M}(\delta^{+}(n)) \cap \mathrm{M}(\delta^{+}(n')) = \mathrm{M}(\delta^{+}(n) \cap \delta^{+}(n')) \tag{5.12}$$

Then, from Equation 5.12, we define

$$\mathrm{M}^{+}(n, n^{'}) = \mathrm{M}(\delta^{+}(n) \cap \delta^{+}(n^{'})), \forall n \in N, n^{'} \in N^{'} \tag{5.13}$$

In conclusion, the final formulation of the WDK_{PDG} is reported:

Definition 5.6. (WDK for Dependency Graphs)*:*
Let $n \in V$ *and* $n^{'} \in V^{'}$ *be the nodes of two distinct PDGs,* G *and* $G^{'}$*. The* **WDK for Dependency Graph** *is defined as:*

$$WDK_{PDG}(n, n^{'}, l) = \lambda \cdot \widehat{\delta}(n, n^{'}) + (1 - \lambda) \sum_{T \in \mathrm{M}^{+}(n,n')} k_{max\_ctx}(n, n^{'}, T, l) \tag{5.14}$$

where $0 \leq \lambda \leq 1$ *is a parameter that controls the penalisation factor of the traversed contexts, and* l *is a value indicating the length of the maximum traversable path in the context.*
Moreover, the kernel function k_{max_ctx} *on the* contexts *is defined as:*

$$k_{max\_ctx}(n, n^{'}, T, l) = \begin{cases} \frac{1}{|\delta^{+}(n)_{[T]}|} \cdot \sum_{u \in \delta^{+}(n)_{[T]}} & max\{WDK_{PDG}(u, u^{'}, l-1), \\ & \forall u' \in \delta^{+}(n^{'})_{[T]}\} \qquad \textit{if } l > 0 \\ 0, & \textit{otherwise} \end{cases} \tag{5.15}$$

assuming, without loss of generality, that $|\delta^{+}(v)_{[T]}| \leq |\delta^{+}(v')_{[T]}|$.

Therefore, the basic idea is to recursively iterate the exploration of the context admitting at most a number l of recursions. This strategy has two main advantages: it allows to bound the total number of performed comparisons, while, on the other hand, this makes the whole kernel able to handle the cases of possible loops in the graph. Furthermore, the similarity computation recursively selects at each step the substructures that provide the highest similarity values.

5.2.2 Preliminary Evaluation

A preliminary evaluation of the proposed Graph Kernel clone detector has been performed, generating the analysed PDGs using the Code Surfer tool.[†] The main purpose of this preliminary investigation is to analyse the quality of clones, namely the Types, that the WDK_{PDG} Kernel is able to detect. For the sake of compara-

[†] http://www.grammatech.com

bility, the empirical study considered the scenario-based evaluation, already used for the qualitative assessment of the proposed Tree Kernel. The description of the experimental settings is reported in Section 5.1.3, while obtained results are reported in Figure 5.2

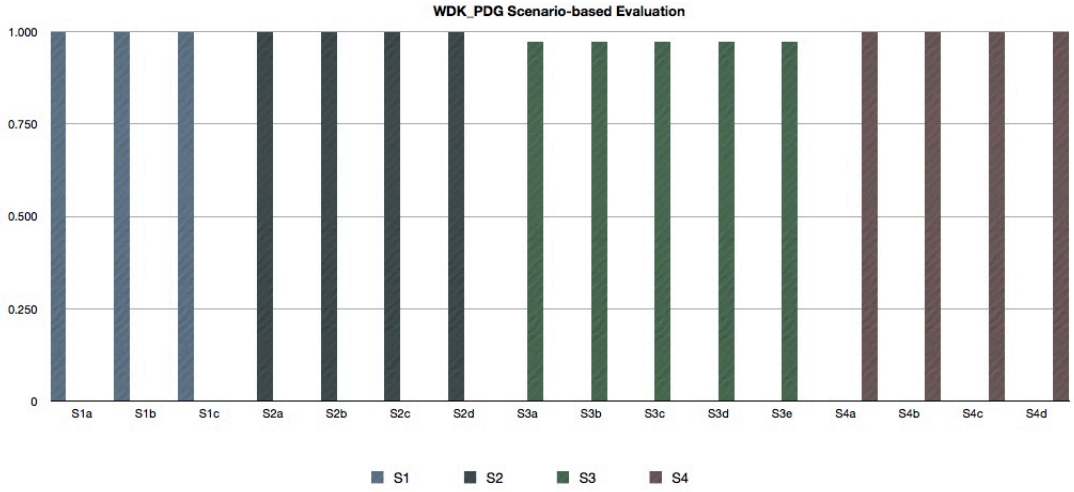

Figure 5.2: Results for the qualitative evaluation of the proposed Graph Kernel WDK_{PDG} clone detection technique.

Results show the very good potentials of the proposed Graph Kernel function for clones. In fact, the WDK_{PDG} kernel outperforms the K_{AST} Tree kernel in dealing with the Type 3 and Type 4 considered clones. This is because the PDG is inherently unaffected by statement inversions since the structure only contains information about the relationships among the different instructions. As a consequence, the WDK_{PDG} seems to be a very promising technique for the identification of Type 4 clones, which are usually the harder to detect [159], and represent the real "frontier" for researchers. However, the improvements in results is traded-off by a penalisation in the overall (asymptotically) computational complexity, w.r.t. the Tree Kernel based solution.

Nevertheless, to deal with this issue, the implemented prototype tool leverages the benefits of the *Message Passing Interface* (MPI) parallel framework. In particular, since the similarity computation could be performed in isolation, the Kernel applies in parallel the comparison of multiple PDGs at the same time, thus lower-

ing the total computation time necessary for the whole clone detection process.

5.3 Towards a Supervised Kernel Learning Approach for Clone Detection

To quantitatively evaluate the effectiveness of a clone detection approach, it is necessary to exactly know which clones occur in the source code of the analysed software system. However, it is widely recognised that this task is usually not feasible when considering an actual software system [104].

In other fields of empirical software engineering, a possible solution could consist in using a public benchmark. Unfortunately, this is not the case with clone detection, since to date there is no benchmark for deeply quantitatively assessing clone detection techniques. In fact, the public data sets used in [21] can be only employed to get an indication of the correctness of the cloning results since only a small fraction of the actual clones is tracked.

In more details, Bellon et al. in [21] defined a *pooling process* [130] where a limited set of results, gathered from different clone detection tools, has been manually cross-checked. However, the effect of such a procedure is that there is no guarantee of completeness and so only an underestimation of the actual performance can be provided.

From a ML perspective, in this scenario only unsupervised techniques, such as clustering (Section 3.3), can be applied. However, clustering approaches guide the learning process only by the similarity computation. Conversely, supervised techniques could be potentially more effective as they could be trained in order to automatically *learn* and infer properties and characteristics of code clones.

Nevertheless, a typical problem in developing supervised ML approaches regards the necessity to arrange two different sets of annotated data, namely the training and the assessment sets.
This problem is particularly important in the case of clone detection where these labeled data sets are even more difficult than usual to produce, since a manual annotation of large systems is infeasible. Furthermore, examples gathered from the dataset provided by Bellon et al. [21] are not so effective for training new algorithms, since they present a bias towards the clones detected by the solutions

used to produce the data set.

As an alternative, in this section an algorithm to construct the training data by simulation is presented. In particular, these data are automatically generated by modifying parts of the code of the analysed system, in order to be used afterwards to train a classifier for clone detection.

To assess the effectiveness of the proposal, results gathered by the application of the Tree Kernel-based clone detection (Section 5.1) approach, considering different clone types, and using different similarity thresholds. In particular, the empirical evaluation aims to investigate the validity of generated data by comparing achieved results with those reported in Section 5.1.4.

5.3.1 Automatic Injected Code Clones

The simulation process artificially produces clones by modifying parts of the input project and injects them by following predefined probability distributions. In this way, the quality of the training set can be controlled without any need of imposing restrictions on its size. A Kernel-based classifier is then trained on this data set.

To this aim, we designed and implemented an algorithm able to inject clones in the source code. In particular this algorithm would allow us to automatically generate a training set and to apply a more reliable strategy in the definition of the supervised Kernel learning process. The main core of our clone injection algorithm is represented by the function `InjectClone`, whose Pseudocode is reported in Algorithm 10. This algorithm is able to generate function clones and to track their location in the source code, thus obtaining a labeled dataset of clones of the given input Type.

In more details, the algorithm starts its computation by parsing the stream of source code of the analysed software system in order to extract all the target functions (Line 2). Afterwards each function is processed one at a time, deciding whether or not it has to be cloned (Line 6) and how many clones should be generated (Line 9). In particular, we consider that each function has a probability `probCloning` of being cloned. Moreover, if a function has to be cloned, the number of clones to generate is randomly chosen according to a geometric probability distribution with parameter 0.5, namely $\Pr(\texttt{nCopies}) = 0.5^n$ (Lines 9 - 11).

Finally, the algorithm invokes the procedures `Clone` and `Inject` to perform the generation and the injection of clones in the source code respectively, and returns the tracking info of generated data.

The Pseudocode of the `Clone` procedure is reported in Algorithm 11.

Algorithm 10 Clone Injection Algorithm

Input: `sourceCode` : Source code of the system under analysis;
Input: `type` : Type of the clones to generate;
Input: `probCloning`: (Constant) The probability of functions/methods to be cloned.
Output: The source code of the system with randomly injected clones;
Output: The tracking info of injected clones in the source code.

```
 1: function InjectClones(sourceCode, type)
 2:     functionList ← parseAndExtractFunctionsFrom(sourceCode)
 3:     clonesTrackInfo ← ∅
 4:     for each: function ∈ functionList do
 5:         probGenerateClone ← random(0,1)
 6:         if probGenerateClone ≤ probCloning then
 7:             nCopies ← 0
 8:             dice ← random(0, 1)
 9:             while not (2^−(nCopies+1) ≤ dice ≤ 2^−nCopies) do
10:                 nCopies ← nCopies + 1
11:             end while
12:             for i = 1 to nCopies do
13:                 newClone ← Clone(function, type)
14:                 trackInfo ← Inject(sourceCode, newClone)
15:                 add(clonesTrackInfo, trackInfo)
16:             end for
17:         end if
18:     end for
19:     return clonesTrackInfo
20: end function
```

The `Clone` procedure is able to perform the generation of clones up to Type 4 by employing a set of different procedures to apply specific modifications to the program text (mutation) of the target function. The invocation of such procedures is performed in accordance with the Type of the clone to generate. We are not reporting the Pseudocode of such functions in the current document due to space limitations.

The first mutation operation is performed by the `CopyAndChangeLayout` function (Line 2) that is always applied to the target function, regardless the se-

Algorithm 11 Clone Generation Algorithm

Input: `function` : Target function/method of the analysed system to clone;
Input: `type` : Type of the clone to generate;
Output: The artificially generated clone of the input function.

```
 1: function Clone(function, type)
                                   ▷ This mutation operation holds for every Type of clones
 2:     clone ← CopyAndChangeLayout(function)
 3:     if type ≥ 2 then
                        ▷ This mutation operation is also applicable to Type 3 and Type 4 clones
 4:         SubstituteIdsAndLiterals(clone)
 5:     end if
 6:     if type = 3 then
 7:         AddOrDeleteStatements(clone)
 8:     else if type = 4 then
 9:         ReorderStatements(clone)
10:         SubstituteEquivalentControlStructures(clone)
11:     end if
12:     return clone
13: end function
```

lected clone Type. This is because all the four definitions of clones allow some modification in the layout of the program text. The substitution of identifiers and literals is performed for Type 2 clones up to Type 4 ones, by invoking the `SubstituteIdsAndLiterals` procedure (Line 4). In particular, such procedure processes every literal and identifiers of the input function, each of which has a probability `probSubstituteId` of being substituted with a randomly generated identifier.

When dealing with Type 3 clones, in addition to mutations applied for Type 2, other additional operations should be considered. Indeed, in a Type 3 clone, two fragments of code may differ also in the statements, that could be added or removed (Line 7). Therefore, we assigned the same probability (i.e., 1/2 for each operation) to the insertion of a new statement randomly extracted from the considered software system and the deletion of a statement. Furthermore, we impose an upper bound to the total number of operations which is a randomly generated fraction of the total number of statements in the analysed function. In this way, we may avoid the generation of a totally different function which will not be an actual clone of the target one.

Finally, in case of Type 4 clones, the mutation operations include the reorder-

ing of statements (Line 9) and the replacement of equivalent control structures (Line 10). In particular the former is applied only to declaration and independent statements, while the latter substitutes possibly occurring control structures with other semantically equivalent. For instance, `for loops` may be replaced with `while loops`, as well as `if` − `elseif` conditions substituted by `switch` − `case` structures.

5.3.2 Experimental Settings

In the preceding Section we discussed the limits of the existing data sets for clone detection and described how an artificial data set can be produced. Although such data set aims at training, we used it also to assess the Tree Kernel technique described in Section 5.1. Even if we can not assume that the performance obtained on artificial data will generalise to the real case, these experiments allow us to obtain a better understanding of the force and weakness of a Kernel based clone detection approach.

Dataset

The considered target system is an academic application implemented in Java and developed by a Master student in Computer Science at the University of Naples "Federico II". As first step, the system has been manually analysed to remove the clones introduced during implementation. Afterwards, we applied to this cleaned code the Tree Kernel-based clone detector (see Section 5.1). This is to verify whether it detected either any false clone or actual clones which had escaped the manual search. We then applied the clone generator on this system, limiting the generation of clones to Type 3, to make results comparable with ones reported in Section 5.1.4.

Research Questions and Variables

We assessed in this investigations the three following research questions:

RQ1 : Are the clones identified by the approach correct?

RQ2 : Is the group of clones identified by the approach complete?

Clone Type	Threshold	Precision	Recall	F_1
Type 1	N.A.	1.0	1.0	1.0
Type 2	0.7	0.6	0.9	0.7
Type 2	0.8	0.7	0.6	0.6
Type 3	0.7	0.6	0.8	0.7
Type 3	0.8	0.6	0.8	0.7

Table 5.1: Summary statistics of the results

RQ3 : Does the group of clones identified by the approach comply a good compromise between correctness and completeness?

Note that the definition of the third research question is motivated by the fact that the completeness requirement is opposite to the correctness one, as the former suggests outputting a large number of candidate clones, while the latter implies a more conservative approach, where only quite likely clones are detected.

As already done in previous experimental evaluation, the Precision (Equation 5.5 has been used to measure the correctness of the results, while the completeness has been assessed by employing the Recall measure (Equation 5.6). Finally, to assess whether the approach is effective (RQ3), we computed the F_1 measure (Equation 5.7).

5.3.3 Results and Discussion

Since the Tree Kernel based approach does not include any formatting detail in its internal source code representation, Type 1 clones include no variability, and thus no Similarity threshold is necessary. With this kind of clones, it is easy to obtain 1.0 as F-Measure.

Regarding the other two types of clones, some modifications in the identifiers (Type 2 and 3) and in statements (Type 3 only) have been performed. In these cases, larger values of the threshold (e.g. 0.9) produce a small number of candidates. As a consequence, the recall is low, since only code fragments which are very similar are considered as clones. This effect is particularly evident for Type 3 clones, where no clones at all are detected. On the other hand, threshold values

like 0.7 and 0.8 lead to better performance. In particular, the value 0.7 seems to improve completeness without affecting correctness, and is therefore preferable. Such attained results are strongly comparable with those reported in Section 5.1.4 in terms of all the three indicators we are considering, namely correctness, completeness and effectiveness, thus confirming the validity of artificially generated data.

Threats to Validity

We focused our attention in this section on the *construct validity* and the *external validity*. Construct validity threats concern the relationship between theory and observation. Precision, Recall, and F_1 well reflect the performance of the proposed approach. However, the used data set has been obtained by manually removing source clones and then introducing new clones of Types 1, 2 and 3 in a controlled way. The performed mutations may bias the results since they could affect the values of these measures. However, the defined mutation approach has been conceived to reduce this effect on the results as much as possible.

To increase our awareness on the achieved results we also plan to assess the validity of the results using different measures to determine various aspects of detection quality [21].

External validity threats regard the generalisation of the results. An important threat is related to the studied software system. In particular, the size and the fact that the system was developed by a student may threaten the validity of the results. Also, the fact that this system was implemented in Java may affect the generalisation of the results. To this aim, we plan to conduct case study replications on commercial software systems implemented in different programming language. This will increase our awareness on the validity of applying Kernels Methods in the detection of software clones. Regarding the scalability, software systems with different size and clone density will be studied in the future.

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

Bertrand Russell

6 Conclusions

Software maintenance is a key phase of the software development lifecycle, and consequently many research efforts are devoted to provide new solutions to improve its effectiveness. Nevertheless it represents one of the most expensive and challenging phase of the whole development process. In particular, most of the effort and the time necessary for maintenance activities is spent on understanding the system and its source code.

Indeed, the documentation of large software systems is usually not present or not up-to-date, and so the source code remains the only valuable and reliable source of information for software maintainers. To this aim, most of the automated maintenance tools gather information directly from the source code in order to aid the comprehension of the system, and thus supporting the practitioners in their duties.

Among the different maintenance tasks, this thesis presented the proposed research contributions to three specific problems, namely *software re-modularisation*, *source code vocabulary normalisation* and *clone detection*. In particular, all the

presented approaches are based on the definition and the application of unsupervised machine learning techniques, which provide powerful and flexible solutions to cope with the different considered problems.

A summary description of the proposed approach, together with an outline of possible future research directions are described in the following Sections. In particular, Section 6.1 concerns the problem of software modularisation, while Section 6.2 deals with the problem of code normalisation. Finally, Section 6.3 summarises contributions for clone detection, and concludes the Chapter.

6.1 Software re-modularisation

Summary: A common scenario that has to be faced during the maintenance of a software system is the lack of reliable architectural documentation, that often is missing or not properly up-to-date. In this situation, reverse engineering tools have to be employed to align it with the actual implemented software architecture [132, 164]. These tools usually rely on clustering-based approaches to group sets of related classes, exploiting some structural-based measures of similarity among software artefacts.

In this thesis the use of a similarity measure based on lexical information for the clustering related software artefacts has been proposed. In particular, the solution investigates the effects of mining the lexical information gathered from six independent vocabularies: *Class Names*, *Attributes Names*, *Function Names*, *Parameter Names*, *Comments*, and *Source Code Statements*.

The results of the clustering on these features have been then evaluated using two criteria: Authoritativeness and Non-Extremity Distribution (NED). These have been applied on 19 open source Java systems.

The first interesting finding we gathered is that considering the lexical information gathered from the six considered zones always improves results over the "flat" configuration, namely all the terms from the zones merged together.

The other key finding is that each project has its own peculiarities as for the distribution and the relevance of the terms within these zones. Thus, to exploit at its best the potentials of the lexical information embedded in a software system, a mechanism to automatically weight the importance of each vocabulary is

absolutely required.

In particular, the presented contribution leverages the Expectation-Maximisation algorithm based on different initialisation strategies and adopted probabilistic models, namely Gaussian, Bernoulli e Multinomial.

We got the empirical evidence that the introduction of a weighting technique highly improves results, with a mean enhancement of 40% in terms of authoritativeness, on the considered dataset.

The last contribution concerns the adopted clustering that is based on the application of two well-known clustering strategies, namely the K-Medoid and the Group Average Hierarchical Clustering, which have been properly customised to make them more suited for the considered domain.

The final results show that lexical information, if properly weighted, can be successfully employed for software clustering, since it provides better results than unweighted ones.

Future research directions: Several are the possible future direction. The first and the most easy to attain is to extend the preprocessing phase using the LINSEN algorithm, and to automatically associate labels to the clusters.

At the same time, another direction could be the investigation of software systems implemented in different object oriented programming languages. This could be easily attained thanks to the flexibility inducted by srcML. Moreover, we will investigate the use of commercial software systems, rather than open source ones.

At the same time, it will be very interesting to investigate the possibility to infer potential relationships between the relevance of each zone, and some process-specific elements, such as the adopted development methodology.

Finally, the possibility of merging lexical information with the structural one, coming from the original structure of the classes within the packages will be investigated. To this aim, we plan to investigate the applicability of Kernel Methods to software re-modularisation.

6.2 Source code vocabulary normalisation

Summary: Lexical information provided by programmers in the source code identifiers is crucial for many software analysis/maintenance tools, but in order to be properly exploited, identifiers must be processed, to split concatenated words and to expand abbreviations.

This thesis describes a novel technique, called LINSEN, that is able to map a given identifier to the set of corresponding dictionary words. The technique is based on a pattern matching algorithm, the BYP, and is suitable to deal with both the splitting of an identifier into its constituent tokens and the expansion of possible occurring abbreviations.

The proposal has been implemented in a prototype we have assessed using 24 open source software systems, mainly written in C/C++. Results highlight that our proposal outperforms three state-of-the-art solutions in terms of accuracy or F-Measure, showing also interesting computation time even in its currently unoptimised implementation.

One of the main features of the proposed approach regards the usage of multiple dictionaries of words, providing information from different domains, in order to tackle the problem of possible ambiguities.

Future research directions: As future work, there are many directions that could be investigated to gain a better insight on the covered phenomena.

The first interesting one concerns a more detailed investigation on the impact of the choice of the different exploited dictionaries on the effectiveness of the LINSEN algorithm. Similarly, another interesting extension of the presented work concerns the introduction of more programming-oriented dictionaries, such as the ones exploited by Enslen [61] and Hill [85], in order to compensate the actual limitation of the approach in dealing with abbreviations related to specific code structures, such as custom data types (e.g., `DHT` as for *Dynamic Hash Table*).

A replicated study is also required, involving commercial software systems, in order to understand whether the open source development scenarios could bias the results.

Moreover it would be very interesting to perform a psycho-linguistic study, in-

volving a number of programmers, in order to understand the relation between abbreviated and expanded forms of identifiers, in ambiguous cases. The outcome of this study could provide interesting indications which could be embedded in new versions of the proposed approach.

6.3 Clone detection

Summary: The presence of software clones may lead to difficulties to perform maintenance tasks (e.g., corrective maintenance operations). Accordingly, there is a need for tools able to detect clones within software repositories [21].

Although many research efforts have been devoted to this end, there is still the needed for approaches that combine more techniques and/or information to effectively detect code clones [159].

The research contributions presented in this thesis for clone detection are focused on Kernel methods, exploiting them as a powerful and flexible tool for measuring "similarity" between code fragments.

Indeed, Kernel Methods are a natural candidate for learning problems involving richly structured objects, and the two proposed solutions exploit structural representation of the source code, namely the Abstract Syntax Tree (AST) and the Program Dependency Graph (PDG), in order to compute the similarity between code fragments. To this aim, Tree and Graph Kernel based techniques have been proposed, respectively.

In more details, the first solution uses a proper set of features to annotate the nodes of the AST and a set of kernel functions to measure the similarity among subtrees. On the other hand, the latter defines a proper WDK Graph Kernel for clone detection exploiting the peculiarities of nodes and edges of the PDG.

Both the presented techniques have been empirically assessed in an qualitative and quantitative evaluation. The qualitative evaluation considers the different editing *scenarios* proposed by Roy et al. [159], to get insights and indications on the potential capabilities of the Kernel-based code similarities for clones. On the other hand, the quantitative evaluation has been performed by comparing achieved results with other state-of-the-art techniques

The promising results in clone detection using kernels on AST and PDG are

encouraging, showing the potential of structured kernels in uncovering similarities between fragments.

In more details, the Tree Kernel approach has been applied on a Java software system manually cleaned of all the cloned methods. Successively, this cleaned system has been mutated to randomly introduce clones of Types 1, 2, and 3. Clones at method granularity level have been considered in the experimentation, and obtained results have been compared to those obtained by another pure AST-based technique on the same dataset. The results indicated that Tree Kernels are particularly suitable to detect up to Type 3 clones, outperforming the considered state-of-the-art technique.

Conversely, the Graph Kernel algorithm has been applied on two open source projects, namely Apache and Python, and results compared with another graph-based state-of-the-art technique. Results show an improvement over the existing technique on a publicly available oracle.

In addition, a promising new method for the automatic generation of labeled clones benchmark is presented as well.

Future research directions: A first improvement that should be made is to apply a more extensive evaluation of the Graph Kernel approach for clones in order to corroborate the promising results obtained in the performed qualitative evaluation. In particular, these results should also compared with the Tree Kernel approach in order to investigates the benefits provided by both the two proposed solutions.

Another important research direction to investigate is the one that motivated the development of the *mutation algorithm* presented at the end of Chapter 5 (Section 5.3). In particular, the whole main objective of this work is to try to set up a Kernel Learning framework based on a supervised learning strategy.

The wider variability of code found within functional modules requires an adaptation of kernels in order to effectively detect them. The problem can be addressed by combining kernel redesign with kernel learning approaches [78], where the similarity measure is not fully specified a-priori, but is learned from examples as a combination of similarity patterns (e.g. involving different types of lexical and structural information). Logic kernels [70, 111] are particularly promising in this

context, as they allow to encode arbitrary domain knowledge concerning relationships between code fragments from which similarity measures are to be learned. As a result, in this scenario, no more feature weighting schemas would be necessary as the corresponding "importance" of each feature is automatically inducted by the training algorithm.

References

[1] Google's Python Style Guide. URL http://google-styleguide.googlecode.com/svn/trunk/pyguide.html?showone=Naming#Naming.

[2] Java Coding Style Guide. URL http://developers.sun.com/sunstudio/products/archive/whitepapers/java-style.pdf.

[3] *IEEE Trans. Softw. Eng.*, 14(12), 1988. ISSN 0098-5589.

[4] *LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations*, Riva del Garda (Trento),Italy, 2012. IEEE Computer Society. doi: 10.1109/ICSM.2012.6405277.

[5] Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the evolution of the source code vocabulary. In *13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, pages 189–198, 2009.

[6] Alain Abran. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Computer Society, Piscataway, NJ, USA, 2001. ISBN 0769510000.

[7] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communication ACM*, 18(6):333–340, 1975.

[8] A. Aizerman, E. M. Braverman, and L. I. Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[9] J.R. Anderson, R.S. Michalski, J.G.M. Carbonell, and T.M. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Number v. 2 in Machine Learning: An Artificial Intelligence Approach. Morgan Kaufmann, 1986. ISBN 9780934613002.

[10] Periklis Andritsos and Vassilios Tzerpos. Information-theoretic software clustering. *IEEE Trans. Software Eng.*, 31(2):150–165, 2005.

[11] N. Anquetil, C. Fourrier, and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *In Proceedings of the 6th Working Conference on Reverse Engineering*, pages 235–255, Washington, DC, USA, 1999. IEEE Computer Society.

[12] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[13] R. S. Arnold. *Software Reengineering*. IEEE Computer Society, 1993.

[14] R.S. Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989. ISSN 0018-9219. doi: 10.1109/5.24146.

[15] Lowell Jay Arthur. *Software evolution: the software maintenance challenge*. Wiley-Interscience, New York, NY, USA, 1988. ISBN 0-471-62871-9.

[16] Lerina Aversano, Luigi Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In Rene L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca, editors, *CSMR*, pages 81–90. IEEE Computer Society, 2007.

[17] Ricardo A. Baeza-yates and Chris H. Perleberg. Fast and practical approximate string matching. In *In the Third Annual Symposium on Combinatorial Pattern Matching*, pages 185–192. Springer-Verlag, 1992.

[18] B. Baker. On finding duplication and near-duplication in large software systems. In *IEEE Proceedings of the Working Conference on Reverse Engineering*, 1995.

[19] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring Clone Based Reengineering Opportunities. In *Proceedings of the IEEE Symposium on Software Metrics*, pages 292–303. IEEE Computer Society, November 1999.

[20] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE Press, 1998.

[21] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore M. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, pages 577–591, 2007.

[22] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1220164. URL http://doi.acm.org/10.1145/1134760.1220164.

[23] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22:36–49, July 1989. ISSN 0018-9162.

[24] R.V. Binder and J.J.-P. Tsai. Kb/rms: an intelligent assistant for requirement definition. In *Tools for Artificial Intelligence, 1990.,Proceedings of the 2nd International IEEE Conference on*, pages 610–616, 1990. doi: 10.1109/TAI.1990.130407.

[25] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To camelcase or under_score. In *IEEE 17th International Conference on Program Comprehension, 2009. (ICPC).*, pages 158 –167, 2009.

[26] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: http://dx.doi.org/10.1109/FOSE.2007.27.

[27] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006. ISBN 9780387310732.

[28] R. A. Bittencourt and D. D. S. Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 251–254, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3589-0.

[29] Max Bramer. *Principles of Data Mining*. Springer, 1 edition, March 2007. ISBN 1846287650.

[30] Jr. Brooks, F.P. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. ISSN 0018-9162. doi: 10.1109/MC.1987.1663532.

[31] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, June 1983. ISSN 00207373. doi: 10.1016/s0020-7373(83)80031-5.

[32] Peter Bulychev. Clonedigger, 2008. http://clonedigger.sourceforge.net/.

[33] Peter Bulychev and Marius Minea. Duplicate code detection using anti-unification. In *Spring/Summer Young Researcher's Colloquium*, 2008.

[34] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: A semiautomated approach. *Journal on Web Engineering*, 3(1):3–21, 2004.

[35] F. Camastra. *Kernel Methods for Unsupervised Learning*. PhD thesis, Universitgli studi di Genova, 2004.

[36] G. Canfora, A. Cimitile, and M. Munro. Re2: Reverse-engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, 6 (2):53–72, 1994. ISSN 1096-908X. doi: 10.1002/smr.4360060202.

[37] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 97–. IEEE Computer Society, 2000. ISBN 0-7695-0753-0.

[38] Jaime G. Carbonell, Ryszard S. Michalski, and Tom M. Mitchell. Machine learning: A historical and methodological analysis. *AI Magazine*, 4(3):69–79, 1983.

[39] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, January 2001. ISSN 1040-550X.

[40] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, 1990. ISSN 0740-7459. doi: 10.1109/52.43044.

[41] Michael Collins and Nigel Duffy. Convolution kernels for natural language. In *NIPS*, pages 625–632, 2001.

[42] A. Corazza, S. Di Martino, and G. Scanniello. A probabilistic based approach towards software system clustering. In *14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 89–98, 2010.

[43] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. Investigating the use of lexical information for software system clustering. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 35–44, Washington, DC, USA, 2011. IEEE Computer Society.

[44] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. A tree kernel based approach for clone detection. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-8630-4. doi: 10.1109/ICSM.2010.5609715.

[45] Anna Corazza, Sergio Martino, Valerio Maggio, and Giuseppe Scanniello. Combining machine learning and information retrieval techniques for software clustering. In *Eternal Systems*, volume 255 of *Communications in Computer and Information Science*, pages 42–60. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28033-7.

[46] Anna Corazza, Sergio Martino, Valerio Maggio, Andrea Moschitti, Andrea Passerini, Giuseppe Scanniello, and Fabrizio Silvestri. Using machine learning and information retrieval techniques to improve software maintainability. In *Eternal Systems*, Communications in Computer and Information Science, page In Press. Springer Berlin Heidelberg, 2013.

[47] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001. ISBN 9780262032933.

[48] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, sep 1995. ISSN 0885-6125. doi: 10.1023/A:1022627411411.

[49] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Adams re-trace: A traceability recovery tool. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 32 – 41, march 2005. doi: 10.1109/CSMR.2005.7.

[50] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Identifying cloned navigational patterns in web applications. *Journal of Web Engineering*, 5 (2):150–174, 2006.

[51] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[52] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Control*, 14:261–282, 2006.

[53] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Roy. Statist. Soc. Ser. B*, 39(1): 1–38, 1977. ISSN 0035-9246.

[54] B. Dit, Latifa Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 11–20, 2011. doi: 10.1109/ICPC.2011.47.

[55] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the Software Technology and Engineering Practice*, pages 73–82, Washington, DC, USA, 1999. IEEE Computer Society.

[56] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.90.

[57] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573 –591, july-aug. 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.19.

[58] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, pages 109–118, 1999.

[59] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*. Pattern Classification and Scene Analysis: Pattern Classification. Wiley, 2001. ISBN 9780471056690.

[60] A. Eastwood. Firm fires shots at legacy systems. *Computing Canada*, 19(2): 17, 1993.

[61] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, pages 71–80. IEEE Computer Society, 2009. ISBN 978-1-4244-3493-0.

[62] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2:17–23, 2000.

[63] Letha H. Etzkorn and Carl G. Davis. Automated object-oriented reusable component identification. *Knowl.-Based Syst.*, 9(8):517–524, 1996.

[64] W.S. Evans, C.W. Fraser, and Fei Ma. Clone detection via structural abstraction. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 150–159, 2007. doi: 10.1109/WCRE.2007.15.

[65] Raimar Falke, Pierre Frenzel, and Rainer Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, 2008. ISSN 1382-3256. doi: http://dx.doi.org/10.1007/s10664-008-9073-9.

[66] Henry Feild, David Binkley, and Dawn Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *In Proceedings of IASTED International Conference on Software Engineering and Applications (SEA)*, 2006.

[67] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987. ISSN 0164-0925. doi: 10.1145/24039.24041.

[68] P. Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012. ISBN 9781107422223.

[69] Institute for Electrical and Electronic Engineera (IEEE). Ieee standard for software maintenance. *IEEE Std 1219-1993*, 2, 1999.

[70] P. Frasconi and A. Passerini. Learning with kernels and logical representations. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming: Theory and Application*, volume LNAI 4911, pages 56–91. Springer, 2008.

[71] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368132.

[72] G. Gan and J. Wu. *Data Clustering: Theory, Algorithms, and Applications*. ASA-SIAM series on statistics and applied probability. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2007. ISBN 9780898718348.

[73] David Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0.

[74] T. Gärtner. *Kernels for Structured Data*. Series in machine perception and artificial intelligence. World Scientific Publishing Company, Incorporated, 2008. ISBN 9789812814562.

[75] Thomas Gärtner. A survey of kernels for structured data. *SIGKDD Explor. Newsl.*, 5(1):49–58, jul 2003. ISSN 1931-0145. doi: 10.1145/959242.959248.

[76] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In Bernhard Schlkopf and Manfred K. Warmuth, editors, *Computational Learning Theory and Kernel Machines — Proceedings of the 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop (COLT/Kernel 2003) August 24-27, 2003, Washington, DC, USA*, volume 2777 of *Lecture Notes in Computer Science*, pages 129–143. Springer, Berlin–Heidelberg, Germany, August 2003.

[77] Reto Geiger, Beat Fluri, Harald Gall, and Martin Pinzger. Relation of code clones and change couplings. In Luciano Baresi and Reiko Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2006.

[78] Mehmet Gönen and Ethem Alpaydin. Multiple kernel learning algorithms. *J. Mach. Learn. Res.*, pages 2211–2268, jul 2011. ISSN 1532-4435.

[79] Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific, 2nd edition, 2003. ISBN 981-238-426-X.

[80] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gauneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[81] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 223–226. ACM, 2010. ISBN 978-1-60558-719-6.

[82] M. Harman. The role of artificial intelligence in software engineering. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*, pages 1–6, 2012. doi: 10.1109/RAISE.2012.6227961.

[83] P. Harrington. *Machine Learning in Action*. Manning Publications Company, 2012. ISBN 9781617290183.

[84] David Haussler. Convolution kernels on discrete structures. Technical report, University of California, Santa Cruz, 1999.

[85] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international Working Conference on Mining software repositories (MSR)*, pages 79–88. ACM, 2008. ISBN 978-1-60558-024-1.

[86] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 232–242. IEEE Computer Society, 2009. ISBN 978-1-4244-3453-4.

[87] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 36(3):1171–1220, 2008.

[88] S. Huff. Firm fires shots at legacy systems. *The Business Quarterly*, (55): 30–32, 1990.

[89] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[90] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, sep 1999. ISSN 0360-0300. doi: 10.1145/331499.331504.

[91] A.K. Jain and R.C. Dubes. *Algorithms for clustering data*. Prentice Hall advanced reference series. Prentice Hall, 1988. ISBN 9780130222787.

[92] Stanislaw Jarzabek. *Effective Software Maintenance and Evolution - A Reuse-Based Approach*. Auerbach Publ., 2007. ISBN 978-0-8493-3592-1.

[93] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.30.

[94] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proc. Conf. Centre for Advanced Studies on Collaborative research (CASCON)*, pages 171–183. IBM Press, 1993.

[95] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[96] Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 19–28, Washington, DC, USA, October 2006. David R. Cheriton School of Computer Science, University of Waterloo, IEEE Computer Society.

[97] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data An Introduction to Cluster Analysis*. Wiley Interscience, 1990.

[98] Rick Kazman, Steven G. Woods, and S. Jeromy Carrière. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, WCRE '98, pages 154–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8967-6.

[99] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: memory comparison-based clone detector. *Software Engineering, International Conference on*, pages 301–310, 2011. doi: http://doi.ieeecomputersociety.org/10.1145/1985793.1985835.

[100] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46:604–632, September 1999. ISSN 0004-5411.

[101] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the International Symposium on Static Analysis*, pages 40–56, July 2001.

[102] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[103] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, 2000.

[104] Rainer Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[105] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1. doi: http://dx.doi.org/10.1109/WCRE.2006.18.

[106] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 301–309. IEEE Computer Society Press, 2001.

[107] Jens Krinke, Nicolas Gold, Yue Jia, and David Binkley. Distinguishing copies from originals in software clones. In *IWSC*, pages 41–48, 2010. doi: http://doi.acm.org/10.1145/1808901.1808907.

[108] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3): 230–243, 2007.

[109] Arun Lakhotia and John M. Gravley. Toward experimental evaluation of subsystem classification recovery techniques. In *Working Conference on Reverse Engineering*, pages 262–269, 1995.

[110] G. N. Lance and W. T. Williams. A general theory of classificatory sorting strategies 1. hierarchical systems. *The Computer Journal*, 9(4):373–380, 1967.

[111] Niels Landwehr, Andrea Passerini, Luc Raedt, and Paolo Frasconi. Fast learning of relational kernels. *Mach. Learn.*, 78(3):305–342, March 2010. ISSN 0885-6125. doi: 10.1007/s10994-009-5163-1.

[112] Pat Langley and Herbert A. Simon. Applications of machine learning and rule induction. *Commun. ACM*, 38(11):54–64, nov 1995. ISSN 0001-0782. doi: 10.1145/219717.219768.

[113] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *27th IEEE International Conference on Software Maintenance (ICSM), 2011*, pages 113 –122, sept 2011. doi: 10.1109/ICSM.2011.6080778.

[114] D. Lawrie and D. Binkley. Loyola university of delaware identifier splitting oracle, 2012. URL http://www.cs.loyola.edu/~binkley/ludiso/.

[115] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *17th Working Conference on Reverse Engineering (WCRE), 2010*, pages 3 –12, oct. 2010. doi: 10.1109/WCRE.2010.10.

[116] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Proceedings of the 14th Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE Computer Society, 2006.

[117] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 213–222. IEEE Computer Society, 2007. ISBN 0-7695-2880-5.

[118] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3:303–318, 2007.

[119] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, September 1984. ISSN 0164-1212. doi: 10.1016/0164-1212(79)90022-0. URL http://dx.doi.org/10.1016/0164-1212(79)90022-0.

[120] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.

[121] António Menezes Leitão. Detection of redundant code using r^2d^2. *Software Quality Journal*, 12(4):361–382, 2004.

[122] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. ISBN 0201042053.

[123] A. Liu and J.J.-P. Tsai. A knowledge-based approach to requirements analysis. In *Tools with Artificial Intelligence, 1995. Proceedings., Seventh International Conference on*, pages 26–33, 1995. doi: 10.1109/TAI.1995.479375.

[124] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06*, pages 872–881. ACM Press, 2006.

[125] Michael R. Lowry. Software engineering in the twenty-first century. *AI Mag.*, 13(3):71–87, Sept. 1992. ISSN 0738-4602.

[126] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *2010 14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 68 –77, march 2010.

[127] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 315, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1905-9.

[128] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[129] Zoltán Ádám Mann. Three public enemies: Cut, copy, and paste. *IEEE Computer*, 39(7):31–35, 2006.

[130] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

[131] O. Maqbool. *Architecture Recovery of Legacy Software Systems Using Unsupervised Machine Learning Techniques*. PhD thesis, LUMS, Lahore University of Management Sciences, 2006.

[132] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[133] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107–114, 2001.

[134] S. Marsland. *Machine Learning: An Algorithmic Perspective*. Taylor & Francis, 2011. ISBN 9781420067194.

[135] A. Von Mayrhauser. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28:44–55, 1995. doi: 10.1109/2.402076.

[136] Geoffrey J. McLachlan and Thriyambakam Krishnan. *The EM Algorithm and Extensions*. Wiley-Interscience (Wiley Series in Probability and Statistics), 2 edition, March 2008. ISBN 0471201707.

[137] J. Mclachlan and T. Krishnan. *The EM algorithm and Extensions*. Wiley inter-science, 1996.

[138] Sauro Menchetti, Fabrizio Costa, and Paolo Frasconi. Weighted decomposition kernels. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 585–592, New York, NY, USA, 2005. ACM. ISBN 1-59593-180-5. doi: 10.1145/1102351.1102425.

[139] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540764399, 9783540764397.

[140] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963. ISSN 0001-0782. doi: 10.1145/366246.366248. URL http://doi.acm.org/10.1145/366246.366248.

[141] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32:193–208, March 2006. ISSN 0098-5589.

[142] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[143] J. Moad. Maintaining the competitive edge. *Datamation*, 4(36):61–62, 1990.

[144] Alessandro Moschitti. Making tree kernels practical for natural language learning. In *11st Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, April 3-7, 2006, Trento, Italy*, 2006.

[145] Alessandro Moschitti. Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pages 318–329, 2006.

[146] Alessandro Moschitti, Roberto Basili, and Daniele Pighin. Tree Kernels for Semantic Role Labeling. In *Computational Linguistics*, pages 193–224, Cambridge, MA, USACambridge, MA, USA, 2008. MIT Press. doi: 10.1162/coli.2008.34.2.193.

[147] K. Muller, S. Mika, G. Ratsch, K. Tsuda, and Bernhard Scholkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181–201, 2001. ISSN 1045-9227. doi: 10.1109/72.914517.

[148] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[149] Hoan Anh Nguyen, Tung Thanh Nguyen, N.H. Pham, J. Al-Kofahi, and T.N. Nguyen. Clone management for evolving software. *Software Engineering, IEEE Transactions on*, 38(5):1008–1026, 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.90.

[150] Paul W. Oman. *Milestones in Software Evolution*. IEEE Computer Society, Los Alamitos, CA, USA, 1990. ISBN 081869033X.

[151] O. Port. The software trap – automate or else. *Business Week*, 9(3051): 142–154, 1998.

[152] M. F. Porter. *An algorithm for suffix stripping*, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-454-5.

[153] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th IEEE International Conference on Program Comprehension (ICPC)*, pages 37–48, 2007.

[154] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681.

[155] Lih ren Jen and Yuh jye Lee. Working group. ieee recommended practice for architectural description of software-intensive systems. *IEEE Architecture*, pages 1471–2000, 2000.

[156] Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Using fold-in and fold-out in the architecture recovery of software systems. *Formal Aspect Computations*, 24(3):307–330, 2012.

[157] Chanchal Kumar Roy and James R. Cordy. An empirical study of function clones in open source software. In *WCRE*, pages 81–90. IEEE, 2008.

[158] Chanchal Kumar Roy and James R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[159] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[160] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society. ISBN 0-89791-216-0.

[161] Kamran Sartipi and Kostas Kontogiannis. A user-assisted approach to component clustering. *Journal of Software Maintenance*, 15(4):265–295, 2003. ISSN 1040-550X. doi: http://dx.doi.org/10.1002/smr.277.

[162] Trevor Savage, Meghan Revelle, and Denys Poshyvanyk. Flat3: feature location and textual tracing tool. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 255–258, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1810295.1810345.

[163] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico. Using the kleinberg algorithm and vector space model for software system clustering. In *Proceedings of the IEEE 18th International Conference on Program Comprehension*, ICPC'10, pages 180–189, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4113-6. doi: http://dx.doi.org/10.1109/ICPC.2010.17.

[164] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico. Architectural layer recovery for software system understanding and evolution. *Software Practice and Experience*, 40:897–916, September 2010. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.v40:10.

[165] Giuseppe Scanniello, Michele Risi, and Genoveffa Tortora. Architecture recovery using latent semantic indexing and k-means: an empirical evaluation. In *SEFM '10: Proceedings of the 2010 IEEE International Conference on Software Engineering And Formal Methods*, pages 103–112. IEEE Computer Society, 2010.

[166] B. Scholkopf and A.J. Smola. *Learning With Kernels: Support Vector Machines, Regularization, Optimization and Beyond*. Adaptative computation and machine learning series. "The" MIT Press, 2002. ISBN 9780262194754.

[167] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004. ISBN 9780521813976.

[168] H.M. Sneed and E. Nyary. Extracting object-oriented specification from procedurally oriented programs. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 217–226, 1995. doi: 10.1109/WCRE.1995.514710.

[169] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 2007. ISBN 9780321313799. URL http://books.google.it/books?id=B7idKfL0H64C.

[170] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments

for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE)*, pages 43–52. ACM, 2010. ISBN 978-1-4503-0116-9.

[171] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. Bradford Book, 1998. ISBN 9780262193986.

[172] Robert Tairas, Jeff Gray, and Ira D. Baxter. Visualizing clone detection results. In *ASE*, pages 549–550, 2007.

[173] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[174] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 67–76, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3793-1. doi: \url{http://dx.doi.org/10.1109/SCAM.2009.16}.

[175] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal*, 19(2):295–331, 2011. doi: http://dx.doi.org/10.1007/s11219-010-9115-6.

[176] Paolo Tonella. Concept analysis for module restructuring. *IEEE Trans. Softw. Eng.*, 27(4):351–363, 2001. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.917524.

[177] V. Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 211–218, 2000.

[178] Vassilios Tzerpos and Richard C. Holt. Mojo: A distance metric for software clusterings. In *Proceedings of the Working Conference of Reverse Engineering*, pages 187–193, 1999.

[179] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *WICSA*, pages 122–134, 2004.

[180] Jean-Philippe Vert. A Tree Kernel to analyse phylogenetic profiles. *Bioinformatics*, 18(suppl_1):S276–284, 2002. doi: 10.1093/bioinformatics/18.suppl_1.S276.

[181] S V N Vishwanathan and Alex Smola. Fast kernels for string and tree. In *Proceedings of the Neurual Information Processing Systems*, 2002.

[182] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 128–135, Washington, DC, USA, 2004. IEEE Computer Society.

[183] Mark Weiser. Program slicing. *IEEE Transaction of Software Engineering*, 10(4):352–357, 1984.

[184] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *IWPC*, pages 194–203. IEEE Computer Society, 2004. ISBN 0-7695-2149-5.

[185] B.A. Wichmann, A. A. Canning, D.L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W R Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69–75, 1995. ISSN 0268-6961.

[186] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 33–43, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8162-4.

[187] A. E. Wu, J. Hassan and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 525–535. IEEE Computer Society, 2005.

[188] R. Xu and D. Wunsch. *Clustering*. IEEE Press Series on Computational Intelligence. Wiley, 2008. ISBN 9780470382783.

[189] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, July 1991.

[190] S.S. Yau, J.S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, pages 60–65. IEEE Computer Society, 1978.

[191] Yang Yuan and Yao Guo. Cmcd: Count matrix based code clone detection. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 250–257, 2011. doi: 10.1109/APSEC.2011.13.

[192] D. Zhang and J.J.P. Tsai. *Advances in MacHine Learning Applications in Software Engineering*. Idea Group Pub., 2007. ISBN 9781591409410.

[193] Du Zhang and J.J.-P. Tsai. Machine learning and software engineering. In *Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings. 14th IEEE International Conference on*, pages 22–29, 2002. doi: 10.1109/TAI.2002.1180784.