# Configuring and Assembling Information Retrieval based Solutions for Software Engineering Tasks

Bogdan Dit

Cluj-Napoca, Cluj, Romania

Master of Science, Wayne State University, 2009
Bachelor of Science, Babeş-Bolyai University (Romania), 2006

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
August, 2015

# APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Bogdan Dit

Approved by the Committee, June, 2015

Committee Chair
Associate Professor Denys Poshyvanyk, Computer Science
The College of William and Mary

Associate Professor Peter Kemper, Computer Science
The College of William and Mary

Assistant Professor Xu Liu, Computer Science
The College of William and Mary

Associate Professor Andrian Marcus, Computer Science
The University of Texas at Dallas

Associate Professor Massimiliano Di Penta, Engineering
University of Sannio (Benevento, Italy)

# ABSTRACT

Textual or unstructured data generated during the software development process contains a significant amount of useful information that captures design decisions and the rationale of developers. One of the ways to exploit this information in order to support various software engineering (SE) tasks (e.g., concept location, traceability link recovery, change impact analysis, etc.) is to use Information Retrieval (IR) techniques (e.g., Vector Space Model, Latent Semantic Indexing, Latent Dirichlet Allocation, etc.).

Two of the most important steps in a typical process of applying IR techniques to support SE tasks are: (i) preprocessing the corpus (i.e., a set of documents associated with a software system) by removing special characters, splitting identifiers, removing stop words, stemming identifiers, etc. and (ii) configuring the IR technique (i.e., setting up its parameters) and applying it on the preprocessed corpus.

In our previous work, we observed that the various options available for the preprocessing steps of the corpus (e.g., splitting identifiers), as well as the different parameter values for configuring IR techniques (e.g., configuring the parameters for LDA) can significantly influence the results produced by IR techniques on different datasets for various SE tasks.

This dissertation proposes the use of Genetic Algorithms (GAs) to automatically configure and assemble an IR process to support software engineering tasks. The approach named IR-GA determines the (near) optimal solution to be used for each step of the IR process. For example, for the corpus preprocessing steps our IR-GA approach will determine which special characters to remove, will choose the method to split the identifiers, will decide whether or not to remove stop words and how to stem identifiers. In addition, for the chosen IR technique it will automatically determine its (near) optimal parameter values. In an extensive empirical study, we applied IR-GA on three different software engineering tasks: (i) traceability link recovery, (ii) feature location, and (iii) identification of duplicate bug reports. The results of the study indicate that IR-GA outperforms approaches previously used in the literature, and that it does not significantly differ from an ideal upper bound that could be achieved by a supervised approach (i.e., one that knows the results a priori) and a combinatorial approach (i.e., one that considers a large number of parameter combinations and knows the results beforehand).

# TABLE OF CONTENTS

# ACKNOWLEDGEMENTS

I would like to express my deepest and sincere gratitude to everyone who was involved in my dissertation or has helped me in some form or another along the way.

First and foremost, I would like to thank my advisor Denys Poshyvanyk. I will be forever grateful for his exemplary guidance, patience and constant support. Being a great mentor, he constantly pushed me improve myself and I have learned a tremendous amount from him. It was a privilege to be his student.

I would also like to thank my dissertation committee members, Massimiliano Di Penta, Peter Kemper, Xu Liu and my former advisor Andrian Marcus for their insightful comments and feedback that helped shape and improve my dissertation.

During my Ph.D. studies I was fortunate to work on various research projects (some of which are part of this dissertation and some are not) involving extraordinary researchers. I am grateful to my research collaborators Annibale Panichella, Massimiliano Di Penta, Rocco Oliveto, Gabriele Bavota, Andrea De Lucia, Latifa Guerrouj, Giuliano Antoniol, Huzefa Kagdi, Fabian Beck, Jane Cleland-Huang and Mark Grechanik. I truly enjoyed working on those projects. Moreover, I would like to express my gratitude to all SEMERU members that I collaborated with on research projects or I had great interactions. Specifically, I would like to thank former SEMERU members Malcom Gethers, Collin McMillan, Meghan Revelle, Evan Moritz, Andrew Holtzhauer, Jaleo Velasco-Madden, Michael Wagner, Trevor Savage and Samuel Klock, as well the current members Mario Linares-Vásquez, Carlos Eduardo Bernal Cardenas, Boyang Li, Qi Luo, Kevin Moran, Michele Tufano, Christopher Vendome and Martin White.

I would also like to express my gratitude to the department's administration staff (in particular to Vanessa Godwin and Jacqulyn Johnson) and to the Reves Center staff (in particular to Stephen Sechrist, Eva Wong and Emily Bailey) who have helped me countless times in successfully navigating the intricate bureaucracy web that is intrinsic with international students.

I would like to thank my family and friends for their encouragement and support. They were always with me when I needed them. Above all, I thank my wife Mia, whose help and support cannot be expressed in words.

The work described in this dissertation was partially funded by the grants CCF-1016868, CNS-0959924 and CCF-0916260 from the U.S. National Science Foundation. Any opinions, findings, and conclusions expressed herein are those of the author and do not necessarily reflect those of the sponsor.

To my family

# LIST OF TABLES

# LIST OF FIGURES

# 1 Introduction

Unstructured or semi-structured textual information generated during the software development process (e.g., source code, bug reports, documentation, diagrams, etc.) contains a significant amount of information that captures design decisions and the rationale of developers. One of the ways to exploit this information in order to support various software engineering (SE) tasks (e.g., concept location, traceability link recovery, change impact analysis, etc.) is to use Information Retrieval (IR) techniques (e.g., Vector Space Model (VSM) [160], Latent Semantic Indexing (LSI) [42], Latent Dirichlet Allocation (LDA) [22], etc.).

Two of the most important steps in a typical process of applying IR techniques to support SE tasks are: (i) preprocessing the corpus (i.e., a set of documents associated with a software system) by removing special characters, splitting identifiers, removing stop words, stemming identifiers, etc. and (ii) configuring the IR technique (i.e., setting up its parameters) and applying it on the preprocessed corpus.

In this dissertation, we investigate the effects of various factors, such as different preprocessing options, various IR techniques and their configurations, on their effectiveness to support SE tasks, and we illustrate that when improperly configured, these factors have a negative impact on supporting SE tasks. In order to address this problem, we introduce a technique that automatically determines the factors that can be used to configure an IR technique to produce near-optimal results when applied in the context of a SE task.

In our previous work we observed that the various options available for the preprocessing steps of the corpus, as well as the different parameter values for configuring IR techniques can significantly influence the results produced by IR techniques on different datasets for various SE tasks. For example, for the preprocessing step, we investigated the

impact of three identifier splitting techniques for the concept (feature) location task. The results of our empirical study revealed that concept location using IR could benefit from advanced splitting algorithms in some cases. However, the results also show that basic splitting algorithms are sufficient when the textual information is combined with dynamic information (see Chapter 2).

For configuring an IR technique, we illustrated the negative impact of configuring an advanced IR technique, namely Latent Dirichlet Allocation, with "ad-hoc" values that do not take into account the characteristics of the preprocessed corpus. To overcome the common issue of having to choose default values for using LDA on SE tasks, we proposed a technique that uses a Genetic Algorithm to identify the (near) optimal parameter values for LDA, by taking into account quality of the IR model generated from the preprocessed corpus (see Chapter 3).

In addition to LDA, other IR approaches are widely used to support various SE tasks. However, previous studies (see Chapter 2 and Chapter 3) showed that inadequate instantiation of the IR technique and process could significantly affect the performance of such approaches in terms of accuracy and completeness. To overcome these shortcomings, we proposed the use of Genetic Algorithms (GAs) to automatically configure and instantiate IR process for software engineering tasks. The proposed approach, named IR-GA (see Chapter 4), determines the (near) optimal solution to be used for each stage of the IR process. For example, for the corpus preprocessing steps our IR-GA approach will determine which special characters to remove (if any), will determine which algorithm to use for splitting identifiers, will determine whether or not to remove stop words and which algorithms to use for stemming identifiers (if any). In addition, for the chosen IR technique it will automatically determine its (near) optimal parameter values.

To achieve its goal, IR-GA takes into account (i) the task specific components and data sources (i.e., software artifacts related to solving a particular SE task) as well as (ii) the internal properties of the IR model built from the underlying dataset using a large number of possible components and configurations. The search space of possible combinations of instances of IR process components (e.g., preprocessors, IR parameters) to select the candidates with the best expected performance for a given dataset used for a SE task are explored using Genetic Algorithms. More specifically, during the GA evolution, the quality of a solution (represented as a GA individual) is evaluated based on the quality of the clustering of the indexed software artifacts (i.e., the internal IR model). *For this reason, the IR-GA approach is* ***unsupervised*** *and* ***task-independent****, whereas the resulting instantiated process is* ***dataset-specific****. Thus, IR-GA can be used to select and generate on demand an adequate IR-based solution given a dataset provided as input, which could potentially support any IR-based software engineering task (e.g., traceability link recovery, feature location, impact analysis, detection of duplicate bug reports, developer recommendations, source code search, bug triaging, clone detection, etc.).*

The results of an extensive empirical study indicate that by using IR-GA it was possible to automatically assemble a near-optimal configuration of an IR-based solution for datasets related to three software engineering tasks, namely (i) traceability link recovery, (ii) feature location, and (iii) duplicate bug reports identification. In addition, the results of the study indicate that IR-GA outperforms approaches previously used in the literature, and that it does not significantly differ from an ideal upper bound that could be achieved by a supervised approach (i.e., one that knows the results a-priori) and combinatorial approach (i.e., one that considers a large number of parameter combinations).

A common problem in software engineering research is that studies are notoriously hard to reproduce due to lack of datasets, tools, implementation details and other factors. The progress in the field is hindered by the challenge of comparing new techniques against existing

ones, as researchers have to devote a large portion of their resources to the tedious and error-prone process of reproducing previously introduced approaches. We address the problem of experiment reproducibility in software maintenance (SM) in Chapter 5. Moreover, with the proposed solution we facilitate the reproducibility and extensibility of LDA-GA and IR-GA techniques presented in Chapter 3 and Chapter 4 respectively.

## 1.1 Research goals and contributions

In this dissertation, we focus on improving the way IR-based solutions are configured and assembled in order to overcome the performance hits on SE tasks, which are imposed by an improper (or "ad-hoc") configuration of these IR techniques. Our findings show that the performance of IR-based solution is impacted by various factors (e.g., preprocessing options, choice and configuration of IR techniques). Existing solutions are aimed at improving some of these factors in isolation, while ignoring the overarching context in which assembling an IR-based solution requires a set of factors that have an influence on one another.

Our proposed solution is aimed at considering all the influencing factors in the entire context of assembling an IR-based solution (i.e., from choosing the preprocessing steps to choosing and configuring the IR technique) and at suggesting the proper factors that can be used to configure and assemble an IR technique in order to produce improved results for SE tasks.

The dissertation makes the following contributions:

- **Investigation of the impact of different preprocessing steps on feature location.** We present a study of two feature location techniques utilizing three different strategies for splitting identifiers, namely: CamelCase, Samurai and manual splitting of identifiers (see Chapter 2). The main research question asked in this study is "*if we had a perfect technique for splitting identifiers,*

*would it still help improve accuracy of feature location techniques applied in different scenarios and settings"*? In order to answer this research question we investigate two feature location techniques, one based on Information Retrieval and the other one based on the combination of Information Retrieval and dynamic analysis, for locating bugs and features using various configurations of preprocessing strategies on two open-source systems, Rhino [127] and jEdit [85]. The results of an extensive empirical evaluation reveal that feature location techniques using Information Retrieval can benefit from better preprocessing algorithms in some cases, and that their improvement in effectiveness while using manual splitting over state-of-the-art approaches is statistically significant in those cases. However, the results for feature location technique using the combination of Information Retrieval and dynamic analysis do not show any improvement while using manual splitting, indicating that any preprocessing technique will suffice if execution data is available. Overall, our findings outline potential benefits of putting additional research efforts into defining more sophisticated source code preprocessing techniques as they can still be useful in situations where execution information cannot be easily collected.

- **Configuring Latent Dirichlet Allocation to support SE tasks: LDA-GA.** We present a novel approach that automatically configures a topic models IR technique, namely LDA, to support SE tasks. IR techniques, and in particular topic models, have recently been used to support essential software engineering tasks, by enabling software textual retrieval and analysis. In all these approaches, topic models have been used on software artifacts in a similar

manner as they were used on natural language documents (e.g., using the same settings and parameters) because the underlying assumption was that source code and natural language documents are similar. However, applying topic models on software data using the same settings as for natural language text did not always produce the expected results. Recent research investigated this assumption and showed that source code is much more repetitive and predictable as compared to the natural language text. Chapter 3 builds on this new fundamental finding and proposes a novel solution to adapt, configure and effectively use a topic modeling technique, namely Latent Dirichlet Allocation, to achieve better (acceptable) performance across various SE tasks. The novel solution introduced, called LDA-GA, uses Genetic Algorithms to determine a near-optimal configuration of LDA's parameters by taking into account the unique characteristics of the preprocessed corpus. We evaluated LDA-GA in the context of three different SE tasks: (i) traceability link recovery, (3) feature location, and (iii) software artifact labeling. The results of our empirical studies demonstrate that LDA-GA is able to identify robust LDA configurations, which lead to a higher accuracy and better results on all the datasets for these SE tasks as compared to (i) previously published results (which used "ad-hoc" or "default" values for configuring the LDA parameters), (ii) heuristics, and (iii) the results of a combinatorial search (i.e., trying a large number of combinations for LDA's parameters).

- **Configuring and assembling IR-based solutions to support SE tasks: IR-GA.** We have developed a novel approach that automatically determines and assembles the (near) optimal solution for each stage of assembling and

instantiating an IR process that will be used to support a SE task. The proposed approach, called IR-GA, determines the (near) optimal solution to be used for each stage of the IR process, by taking into account the task specific components and data sources as well as the internal properties of the IR model built from the underlying dataset using a large number of possible components and configurations. For example, for the corpus preprocessing steps our IR-GA approach will determine which special characters to remove, will determine how to split identifiers, will determine whether or not to remove stop words and how to stem identifiers. In addition, for the chosen IR technique it will automatically determine its (near) optimal parameter values. In an extensive empirical study, we applied IR-GA on three different software engineering tasks: (i) traceability link recovery, (ii) feature location, and (iii) identification of duplicate bug reports. The results of the study indicate that IR-GA outperforms approaches previously used in the literature, and that it does not significantly differ from an ideal upper bound that could be achieved by a supervised approach (i.e., one that knows the results a-priori) and combinatorial approach (i.e., one that considers a large number of parameter combinations).

- **Supporting Reproducible Empirical Research using TraceLab Component Library.** We present the details of a framework that supports the LDA-GA and IR-GA approaches that we present in Chapter 3 and Chapter 4. Moreover, the framework is designed to support other SE techniques and to facilitate the reproducibility of experiments in empirical research. The motivation for this work was that research studies are notoriously hard to

reproduce due to lack of datasets, tools, implementation details (e.g., parameter values, environmental settings) and other factors, and this is a major issue for the research community. For example, when applying an IR technique to address a SE tasks, lack of details about the exact preprocessing steps used, and configuration details about the IR technique are often overlooked or not specified properly. The progress in the field is hindered by the challenge of comparing new techniques against existing ones, as researchers have to devote a large portion of their resources to the tedious and error-prone process of reproducing previously introduced approaches. We address the problem of experiment reproducibility in software maintenance and provide a long-term solution towards ensuring that future experiments will be reproducible and extensible. We conducted a preliminary mapping study of a number of representative maintenance techniques and approaches and implemented them as a set of experiments and a library of components that we make publicly available with TraceLab, called the Component Library. The goal of these experiments and components is to create a body of actionable knowledge that would (i) facilitate future research and (ii) allow the research community to contribute to it as well. Moreover, we have provided all the components required to reproduce the LDA-GA and IR-GA techniques presented in Chapter 3 and Chapter 4.

## 1.2 Bibliographical Notes

This dissertation contains previously published material. This section details collaborations with other researchers.

The material from Chapter 2 is based on a collaboration with Dr. Latifa Guerrouj at École Polytechnique de Montréal (now at McGill University) and Dr. Giuliano Antoniol at École Polytechnique de Montréal. The results of the research project were originally published in the proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'11) [45].

Chapter 3 and Chapter 4 contains the results of a collaboration with Dr. Annibale Panichella and Dr. Andrea De Lucia at University of Salerno, Italy, Dr. Rocco Oliveto at University of Molise, Italy and Dr. Massimiliano Di Penta at University of Sannio, Italy. The findings from Chapter 3 were previously published in the proceedings of the 35th IEEE/ACM International Conference on Software Engineering (ICSE'13) [134] and in the proceedings of the in 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13) [50]. The latter work published at TEFSE'13 [50] also included a collaboration with Evan Moritz from the College of William and Mary.

The idea and findings presented in Chapter 5 appeared in the proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'13) [47] where it won the Best Paper Award. The publication was invited to a special issue of the Empirical Software Engineering, and an extended version was published at EMSE [48]. This project included a collaboration with Evan Moritz and Mario Linares-Vásquez from the College of William and Mary and Dr. Jane Cleland-Huang from DePaul University.

The datasets and benchmarks presented in Appendix A were generated in collaboration with Dr. Huzefa Kagdi from Wichita State University and Andrew Holtzhauer from the College of William and Mary and were published in the proceedings of the 10th IEEE Working Conference on Mining Software Repositories [46].

# 2 Preprocessing Techniques – Splitting Identifiers

Early work on program comprehension and mental models [169, 179] highlighted the significance of textual information to capture and encode programmers' intent and knowledge in software. Recent research efforts have studied how software developers capture and express their intent in natural language embodied in source code. Identifiers used by programmers as names for classes, methods, or attributes in source code or other artifacts contain vital problem domain information [6, 26, 43, 77, 98, 110, 154, 172] and account for approximately more than half the source code in software [43]. These names often serve as a starting point in many program comprehension tasks [26]; thus, it is imperative that these names clearly reflect the concepts that they are supposed to represent, since self-documenting identifiers reduce the time and effort to acquire a basic comprehension level for any maintenance task [6].

The magnitude of a program's lexicon can hardly be underestimated. Identifiers and comments represent an important source of domain information that is used by (semi-) automated techniques to recover traceability links among software artifacts [5, 112] and locate features in source code [57, 105, 115, 143, 156, 157]. Prior work [79, 167] employed a natural language-based representation of source code, based on the conjecture that there is an intrinsic pattern in unstructured textual information, to support a range of program comprehension activities. Due to the large abstraction gap between the domain of a software system and the implementation mechanisms offered by programming languages, the mapping between domain concepts and their implementation in source code is frequently ambiguous, as these concepts are distorted and scattered in the code [154].

The problem of extracting and analyzing the textual information in software artifacts was recognized by the software engineering research community only recently. Information

Retrieval methods were proposed and used effectively to support program comprehension tasks, such as feature (or concept) location and traceability link recovery. These IR-based approaches vary not only in their scope, but also in their underlying indexing mechanisms, corpus generation, or results analysis methods. Identifier splitting is one of the essential ingredients in any feature location or traceability recovery technique [5, 57, 105, 112, 143, 156] , since it helps disambiguate conceptual information encoded in compound (or abbreviated) identifiers. The widely adopted approach is based on the CamelCase splitting algorithm, with more sophisticated strategies, such as Samurai [60] and TIDIER [75], recently proposed in the literature.

In this chapter we investigate the impact of three identifier splitting techniques (CamelCase, Samurai and manually built splitting (i.e., Oracle)) on the accuracy of feature location in presence and absence of execution information. The main research question that we ask in this study is *if we had a perfect technique for splitting identifiers, such as a manually built oracle, would it still help improve accuracy of feature location techniques applied in different scenarios and settings*? To answer this research question we investigate two feature location techniques (FLTs), one based on IR and the other one based on the combination of IR and dynamic analysis (IRDyn), for locating bugs and features using different configurations of preprocessing strategies on two open-source systems, Rhino [127] and jEdit [85]. Our findings reveal that feature location techniques using IR can benefit from better preprocessing algorithms, and that their improvement in effectiveness while using manual splitting over state-of-the-art approaches is statistically significant. However, the results of the IRDyn FLT do not show any improvement while using manual splitting, indicating that any preprocessing technique will suffice if execution data is available.

## 2.1 Background on Preprocessing Unstructured Information in Software

In this section we overview some of the existing work in the field of feature location and identifier splitting. In particular, we overview two feature location techniques and three approaches for splitting identifiers that are used in our empirical study.

### 2.1.1 Feature Location in Software

Unstructured textual information in software, found in identifiers and comments encodes important problem domain and design decisions about a software system. This unstructured data lends itself for further analysis using IR techniques that can be leveraged to support feature location in source code. *Feature location* is the activity of finding the source code elements (i.e., methods or classes) that implement a specific feature (e.g., "print page in a text editor" or "add bookmark in a web-browser") [115, 143]. In this dissertation, we rely on two feature location approaches that use IR and a combination of IR and dynamic analysis. While there are several IR techniques that have been successfully applied in the context of feature location, such as the Vector Space Model [57], Latent Semantic Indexing [105, 143, 156, 157], and Latent Dirichlet Allocation [108], this empirical study focuses on evaluating LSI for feature location, and the notation IR is used to denote that LSI is the default information retrieval method used in the study. We also provide the details of these feature location approaches and explain the role of identifier splitting techniques in this process. Feature location via LSI follows five main steps: generating a corpus, preprocessing the corpus, indexing the corpus using LSI, formulating a search query and generating similarities and finally, examining the results.

**Step one – generating the corpus**. The source code of a software system is parsed, and all the information associated with a method (i.e., comments, method declaration, signature and body) will become a document in the system corpus. In other words, we are using a

method-level granularity for the corpus, so each method from the source code has a corresponding document in the corpus.

**Step two – preprocessing the corpus**. The generated corpus is then preprocessed in order to normalize the text contained in the documents. This step includes removing operators, programming language keywords, or special characters. Additionally, compound identifiers are split using the algorithms that are explained in details in subsection 2.1.2, as these algorithms are at the core of this dissertation. The split identifiers are then stemmed (i.e., reduced to their root form) using the Porter stemmer [138], and finally the words that appear commonly in English (i.e., "a", "the", etc.) are eliminated.

**Step three - indexing the corpus using LSI**. The preprocessed corpus is transformed into a term-by-document matrix, where each document (i.e., method) from the corpus is represented as a vector of terms (i.e., identifiers). The values of the matrix cells represent the weights of the terms from the documents, which are computed using the term frequency – inverse document frequency (tf-idf) weight. The matrix is then decomposed using Singular Value Decomposition [42] which decrease the dimensionality of the matrix by exploiting statistical co-occurrences of related words across the documents.

**Step four – formulating a search query and generating similarities**. The software developer chooses a set of words (i.e., a *query*) that describe the feature or bug being sought (e.g., "print page"). The query is converted into a vector-based representation, and the cosine similarity between the query and every document in the reduced space is computed. In other words, the textual similarity between the bug description and every method from the software system is computed.

**Step five – examining the results**. The list of methods is ranked based on their cosine similarities with the user query. The developer starts investigating the methods in order, from the top of the list (i.e., most relevant methods first). After examining each method the developer

decides if that method belongs to the feature of interest or not. If it does, the feature location process terminates. Otherwise, the developer can continue examining other methods, or refine the query based on new information gathered from examining the methods and starting from Step 4 again.

Feature location via LSI and dynamic information has one additional step, which can take place before the Step 4 described earlier.

**Step for collecting execution information**. The software developer triggers the bug, or exercises the feature by running the software system and executing the steps to reproduce from the description of the feature or bug. This process invokes the methods that are responsible for the bug or feature and these methods are collected in an execution trace. The developer can take advantage of this information by formulating a query (Step 4) and examining the results (Step 5) produced by ranking only the methods found in the execution trace (as opposed to ranking all the methods of the software system). The advantage of using execution information is that it reduces the search space, thus increasing the performance of feature location.

In this chapter, we consider the IR and IRDyn FLTs. While previous studies have shown that the IRDyn FLT outperforms its basic version (i.e., IR FLT) [105, 143, 156, 157], the goal of this work is to study the impact of the preprocessing techniques from Step 2 on the accuracy of feature location.

### 2.1.2 Background on Identifier Splitting Techniques

State-of-the-art approaches to split identifiers into separate words are the CamelCase splitter, the Samurai approach proposed by Enslen et al. [60], and the recent TIDIER approach [75].

### *A. CamelCase Splitting Technique*

The de facto splitting algorithm is CamelCase. This simple, fast, and widely used preprocessing algorithm has been previously applied in multiple approaches to feature location and traceability link recovery [5, 105, 112, 115, 143, 156, 157]. This approach splits compound identifiers according to the following rules:

**RuleA**: Underscore, structure and pointer access, as well as special symbols are replaced with the space character.

**RuleB**: Identifiers are split where terms are separated using the CamelCase convention. For example, *userId* is split into *user* and *Id* while *setGID* is split into *set* and *GID*.

**RuleC**: When two or more upper case characters are followed by one or more lower case characters, the identifier is split at the last-but-one upper-case character. For example, *SSLCertificate* is split into *SSL* and *Certificate*.

Sometimes, a space is inserted before and after each sequence of digits. For example, *print_file2device* is split into *print*, *file*, *2*, and *device*, while *cipher128_code* is split into *cipher*, *128*, and *code*. Overall, a CamelCase splitting algorithm cannot split effectively same-case composite words, such as *USERID*, *currentsize*, into separate terms.

### *B. Samurai Splitting Algorithm*

Samurai [60] is an automatic approach to split identifiers into sequences of terms by mining term frequencies in large source code bases. It relies on two assumptions. First, it assumes that a substring composing an identifier is also likely to be used in other parts of the program (or in other programs) alone or as a part of other identifiers. Second, given two possible splits, the split that most likely represents the developer's intent partitions the identifier into terms occurring more often in the program. In other words, central to Samurai is the idea of using two tables of frequencies: one program specific and one mined out of a large corpus of programs, to find the most likely identifier split. Furthermore, the frequency tables

are used in conjunction with CamelCase rules. In fact, Samurai algorithm first tries to apply CamelCase split and then ranks possible splits according to its identifiers frequency tables. In this way Samurai overcomes the main limitation of CamelCase, by being able to correctly split same-case identifiers, such as *USERID*, *currentsize*, or mixed-case (e.g., *DEFMASKBit*). Refer to [60] for more details on Samurai and its evaluation.

### *C. TIDIER: Term IDentifier RecognizER*

TIDIER [75] is a novel approach to split program identifiers using high-level and domain concepts captured into multiple dictionaries. The approach is based on a thesaurus of words and abbreviations and uses a modified string-edit distance [100] between terms and words as a proxy for the distance between the terms and the concepts they represent. The main assumption made by TIDIER is the fact that it is possible to mimic developers when creating an identifier relying on a set of transformation rules on terms/words.

For example, to create an identifier for a variable that counts the number of software defects, the two words, *number* and *defects*, can be concatenated with or without an underscore, or following the CamelCase convention e.g., *defects_number*, *defectsnumber* or *defectsNumber*. Developers may drop vowels and (or) characters to shorten one or both words of the identifier, thus creating *defectsNbr* or *nbrOfdefects*. TIDIER uses contextual information in the form of specialized dictionaries (e.g., acronyms, contractions and domain specific terms) and mimics the process of transforming words via contraction rules; more details can be found in [75]. It is important to emphasize that TIDIER does not perform significantly better than Samurai on Java code and even though TIDIER and Samurai outperform CamelCase, Samurai is much faster than TIDIER. For this reason, TIDIER was only used as a reference in supporting the construction of the Oracle but not in the empirical study or to generate new terms as in [75].

Table 2-1 The configurations of the two FLTs (i.e., IR and IRDyn) based on the splitting algorithm

| Splitting Algorithm | IR FLT | IRDyn FLT |
|---|---|---|
| CamelCase (Baseline) | $IR_{CamelCase}$ | $IR_{CamelCase}Dyn$ |
| Samurai | $IR_{Samurai}$ | $IR_{Samurai}Dyn$ |
| Oracle (Manual Split) | $IR_{Oracle}$ | $IR_{Oracle}Dyn$ |

## 2.2 Empirical Study Design

The goal of this study is to compare accuracy of two FLTs (i.e., IR and IRDyn), when utilizing three identifier splitting algorithms: CamelCase, Samurai and Oracle (i.e., manual splitting of identifiers). This study is done from the perspective of researchers who want to understand if existing approaches for splitting identifiers can improve accuracy of FLTs under different scenarios and settings, including best possible scenario where splitting is done by experts. In addition, we are interested to know if an advanced splitting algorithm would be still useful for enhancing the accuracy of feature location when execution information is used.

The context consists of two Java applications: Rhino and jEdit where the main characteristics are described in Subsection 2.2.3.

### 2.2.1 Variable Selection and Study Design

The main independent variable of our study is the type of splitting algorithm used: CamelCase, Samurai and Oracle (i.e., manually split identifiers).

The second independent variable is the use of dynamic information. Thus, we have two FLTs, and each has three configurations, which depend on the identifier splitting technique (see Table 2-1). For example, $IR_{CamelCase}$, $IR_{Samurai}$, and $IR_{Oracle}$ are the IR based feature location techniques that use LSI to compute similarities between queries and methods, after applying the CamelCase, the Samurai and the Oracle splitting algorithms on the identifiers from the methods and queries. Similarly $IR_{CamelCase}Dyn$, $IR_{Samurai}Dyn$ and $IR_{Oracle}Dyn$ were defined.

In order to compare which configuration of the FLTs is more accurate than another (i.e., $IR_{CamelCase}$ vs. $IR_{Samurai}$), we considered their *effectiveness measure* [105]. The effectiveness measure is the best rank (i.e., lowest rank) among all the methods from the gold set for a specific feature. Intuitively, the effectiveness measure quantifies the number of methods a developer has to examine from a list of ranked methods returned by the feature location technique, before she is able to locate a relevant method pertaining to the feature. Obviously, a technique that consistently places relevant methods towards the top of the ranked list (i.e., lower ranks) is more effective than a technique that contains relevant methods towards the middle or the bottom of the ranked list (i.e., higher ranks). In this analysis we focus on the scenario of finding just one relevant method, as opposed to finding all relevant methods from the gold set, for two reasons. First, we are focusing on concept location, rather than impact analysis. Second, once a relevant method has been identified, it is much easier to find other related methods by following program dependencies from the relevant method, or by using other heuristics.

In literature, the identifiers that are split using CamelCase are referred as hard-words, whereas the identifiers split using Samurai or TIDIER are called soft-words. During our analysis, we treat the hard and soft words in the same way and we refer to them as split identifiers.

The dependent variable considered in our study is the effectiveness measure of the FLTs.

We aim at answering the following overarching question: *if we had a perfect technique for splitting identifiers, would it still help improve accuracy of FLTs*? We plan to answer this question by examining these more specific **R**esearch **Q**uestions (RQ):

**RQ1**: Does $IR_{Samurai}$ outperform $IR_{CamelCase}$ in terms of effectiveness?

**RQ2**: Does $IR_{Samurai}Dyn$ outperform $IR_{CamelCase}Dyn$ in terms of effectiveness?

**RQ3**: Does $IR_{Oracle}$ outperform $IR_{CamelCase}$ in terms of effectiveness?

**RQ4**: Does $IR_{Oracle}Dyn$ outperform $IR_{CamelCase}Dyn$ in terms of effectiveness?

Previous work [60, 75] compared the CamelCase, Samurai and TIDIER splitting algorithms in terms of their accuracy for correctly splitting identifiers. However, in our study we are addressing the impact that splitting algorithms have on feature location.

### 2.2.2 Building an Oracle – "Perfect Splitter"

The aim of the Oracle is to provide an exact identifier splitting into terms, and possibly mapping acronyms and contractions into terms or English words, thus building a reference dictionary to be used in subsequent feature location phases. Application dictionaries, collected identifiers and terms from comments, may contain thousands of words. Hence, manual verification and splitting is a tedious and error prone task. To simplify Oracle building we applied a multi-step strategy aiming at minimizing the manual effort. In the following subsections we report details of each step.

**Step one – building software application dictionary.** We parsed and extracted identifiers and comments from both Rhino and jEdit and created a dictionary for each system. During this step we also built an application specific identifier (or term) frequency table for Samurai. Following this preliminary step, we filtered some dictionary entries to reduce manual validation effort.

**Step two – filtering concordant identifier split.** For each dictionary entry we ran the CamelCase, Samurai and TIDIER splitters to locate the identifiers for which these three splitting algorithms were in agreement. TIDIER was configured with WordNet[1] dictionary, as well as with acronyms and abbreviations known to the authors. We used the Samurai global

[1] http://wordnet.princeton.edu/

Table 2-2 Summary of the four datasets used in the evaluation: name (number of features/issues), source of the queries and gold sets, and the type of execution information

| Dataset (Size) | Queries | Gold Sets | Execution Information |
|---|---|---|---|
| $Rhino_{Features}$ (241) | Sections of ECMAScript | Eaddy et al. [58] | Full Execution Traces |
| $Rhino_{Bugs}$ (143) | Bug title and description | Eaddy et al. [58] (CVS) | N/A |
| $jEdit_{Features}$ (64) | Feature (or Patch) title and description | SVN | Marked Execution Traces |
| $jEdit_{Bugs}$ (86) | Bug title and description | SVN | Marked Execution Traces |

frequency table made available by Samurai authors [60], as well as a local frequency table estimated from the software application under analysis (see Step 1). Whenever the three splitting algorithms agreed on the identifier term subdivision, we considered this as a strong indication that the resulting split was actually correct. This assumption divided the dictionary into two sub-dictionaries: one on which the algorithms disagree and one where there is agreement among them. The sub-dictionary where the tools agreed was then manually inspected to make sure that no errors were present. For example, out of about 6,000 dictionary entries (or words) for Rhino, about 2,500 words were split in this phase with a minimum manual effort.

**Step three – filtering discordant identifier split.** We manually inspected the identifiers for which the three splitting algorithms did not agree, in order to provide the best splitting. Examples of identifiers from the Rhino dictionary are words such as *DToA*, *DCMPG* or *impdep2*. Most of identifiers were manually split in this step (including careful inspection of the source code to understand the exact context of those identifiers), but there was a reduced set where it was unfeasible to assign any evident meaning even after inspecting the source code. For example, about 120 Rhino dictionary entries fell into this category. Examples of such identifiers include short strings (e.g., *DT*, *i3* or *m5*) and cryptic identifiers (e.g., *P754*, *u00A0* or *zzz*).

During the Oracle building process, the authors validated the split identifiers following a consensus approach (i.e., Latifa Guerrouj proposed an identifier split, which was then verified and validated by Bogdan Dit – see Section 1.2). In a few cases, disagreements were discussed among all the authors. We adapted this approach in order to minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers' domain and solution knowledge, experience, personal preference, etc., thus, it is difficult to decode the true meaning of identifiers in some cases.

### 2.2.3 Systems

We conducted our evaluation on two open source Java systems, Rhino and jEdit, and we constructed four datasets from these two systems. The first system considered is Rhino [127], an open-source implementation of JavaScript written in Java. Rhino version 1.6R5 has 138 classes, 1,870 methods and 32K lines of code. Rhino implements the specifications of the European Computer Manufacturers Association (ECMA) Script[2]. We constructed two datasets from Rhino.

The first dataset is $\text{Rhino}_{\text{Features}}$ and contains 241 features extracted from the specifications. Each feature has a textual description that was used as a query in the evaluation. These descriptions correspond to sections of the ECMAScript specifications. Each feature also has a set of methods, which are associated with the features (i.e., gold set). The gold sets were constructed using the mappings between the source code and the features, which were made available by Eaddy et al. [58]. These mappings[3] were produced by considering the sections of the ECMAScript specification as features, and associating them with software artifacts using the following prune dependency rule, created by Eaddy et al. [58]: "A program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned."

[2] http://www.ecmascript.org/
[3] http://www.cs.columbia.edu/~eaddy/concerntagger/

Table 2-3 Descriptive statistics from datasets: number of methods in the gold set, number of methods in traces, and number of identifiers from corpora

| # of … | Measure | $Rhino_{Features}$ | $Rhino_{Bugs}$ | $jEdit_{Features}$ | $jEdit_{Bugs}$ |
|---|---|---|---|---|---|
| methods in gold set | min | 1 | 1 | 1 | 1 |
| | median | 4 | 1 | 5 | 2 |
| | average | 12.82 | 2.24 | 6.3 | 4.01 |
| | max | 280 | 15 | 19 | 41 |
| | st. deviation | 28.8 | 2.39 | 5.33 | 5.63 |
| | Total | 3,089 | 320 | 403 | 345 |
| unique methods from traces | Min | 777 | N/A | 227 | 227 |
| | median | 917 | N/A | 1.1K | 1.1K |
| | average | 912 | N/A | 1.1K | 1.1K |
| | Max | 1.1K | N/A | 1.9K | 1.9K |
| | st. deviation | 54 | N/A | 310 | 310 |
| identifiers in the corpus (with queries) | split by CamelCase | 3,318 (4,154) | 3,318 (4,223) | 4,227 (4,361) | 4,227 (4,596) |
| | split by Samurai | 2,642 (3,416) | 2,642 (3,411) | 3,439 (3,552) | 3,439 (3,751) |
| | Split by Oracle | 2,030 (2,921) | 2,030 (2,718) | 2,758 (2,852) | 2,758 (3,051) |

These mappings were used in other research papers, such as [57, 58, 156]. Rhino is distributed with a suite of test cases, and each test case has a correspondence in the ECMAScript specification. We used these test cases to collect full traces for each of the features.

The second dataset collected is $Rhino_{Bugs}$ and contains 143 issue reports (i.e., bugs) that were collected from Bugzilla, the issue tracking system of Rhino[4]. Each bug from Bugzilla has a title and a description, and we used this information as queries in the evaluation. As in the $Rhino_{Features}$ dataset, we used the information made available by Eaddy et al. [58] to associate each bug with a set of methods from Rhino which are responsible for the bug (i.e., the gold set). Eaddy et al. [58] extracted the mappings between bugs and source code by analyzing CVS commits. However, there was no association between the 143 issue reports and the test cases, hence, we did not collect any execution traces for this dataset.

[4] https://bugzilla.mozilla.org/

The second system considered is jEdit [85], a popular open-source text editor written in Java. jEdit version 4.3 has 483 classes, 6.4K methods and 109K lines of code. We constructed two datasets from this system. For more details about how we generated the jEdit dataset refer to our Appendix A.

The first dataset is $jEdit_{Features}$ and consists of 64 issues (34 features and 30 patches) extracted from jEdit's issue tracking system[5]. The second dataset is $jEdit_{Bugs}$ and consist of 86 bug reports. We now describe some steps used for collecting additional information for these two datasets. We used the changes associated with the SVN commits between releases 4.2 and 4.3 to construct the gold sets. In addition, the SVN logs were parsed for issue identifiers which were matched against the issues from the tracking system. Similarly to the $Rhino_{Bugs}$ dataset, the title and description of these issues were used in the evaluation as queries. We used a tracer to generate marked traces, by executing jEdit and following the steps to reproduce from the issue description. For more details about the process of generating this dataset, and for the complete dataset, which includes queries and execution traces, please refer to our online appendix [1].

The four datasets, extracted from Rhino and jEdit, which were used in the evaluation, are summarized in Table 2-2. We also present additional information about the datasets used in the evaluation in Table 2-3. First, we present details about the number of methods from the gold sets of each dataset. Each data point (i.e., a feature or a bug) from the $Rhino_{Features}$ dataset has on average 12 methods, whereas the $Rhino_{Bugs}$ dataset has only two methods on average. For jEdit there are on average four to six methods associated with each issue. The features from the $Rhino_{Features}$ dataset have many gold set methods in common, hence the total number of methods is much higher than for the other datasets.

[5] http://sourceforge.net/tracker/?group_id=588

Second, we present information about the number of methods extracted from the traces. For both systems, the average number of unique methods extracted from each trace was about one thousand. Third, we present information about the size of the corpora in terms of the number of identifiers, after applying the CamelCase, Samurai and Oracle splitting techniques. As expected, the more accurately we split the identifiers, the more we reduce the number of unique identifiers. For example, the corpus for $\text{Rhino}_{\text{Features}}$ has 3,318 identifiers after applying the CamelCase splitting technique, and has only 2,030 identifiers after using the Oracle splitting technique. This is explained by the fact that identifiers that could not be split by CamelCase formed an unique identifier, whereas the Oracle split the identifier into two or more (common) terms that already appear in the corpus, hence reducing the number of unique identifiers.

### 2.2.4 Analysis

For each dataset, every FLT will produce a list of ranks (i.e., effectiveness measures) that has the size of the number of features in the dataset. For example, the dataset $\text{Rhino}_{\text{Features}}$ produces 241 ranks for $IR_{CamelCase}$, 241 ranks for $IR_{Samurai}$ and 241 ranks for $IR_{Oracle}$, and each of those ranks represents the best position (i.e., lowest rank) of a method from the gold set associated with that feature. These lists of ranks are used as an input for the following comparison techniques: descriptive statistics, side by side comparisons, and statistical tests.

First, we compare the ranks using descriptive statistics, such as minimum, first quartile, median, third quartile, maximum, and average. We present all these descriptive statistics graphically, using box plots (i.e., whisker charts). Although this technique provides a quick and intuitive view of the data, it only presents a high level perspective.

The second comparison technique examines the data in more details and works as follows. Given two lists of ranks produced by two different FLTs, we compare the ranks side

by side and we count the number of cases the first technique produces lower ranks than the other, as well as the number of cases the second technique produces lower ranks (i.e., better results) than the other. We report these values as percentages.

The third comparison of the ranks is a statistical analysis. We use the Wilcoxon signed-rank test [31] to test whether the difference in terms of effectiveness for two measures is statistically significant or not. This test is non-parametric and it takes as an input two lists of ranks produced by two different feature techniques. In the test we used a significance level $\alpha = 0.05$, and the output of the test is a p-value, which can be interpreted as follows. If the p-value is less than $\alpha$, then the difference in ranks produced by one feature location technique is statistically significantly lower than the ranks produced by the other technique. Otherwise, if the p-value is larger than $\alpha$, then we conclude that the two techniques produce almost equivalent results.

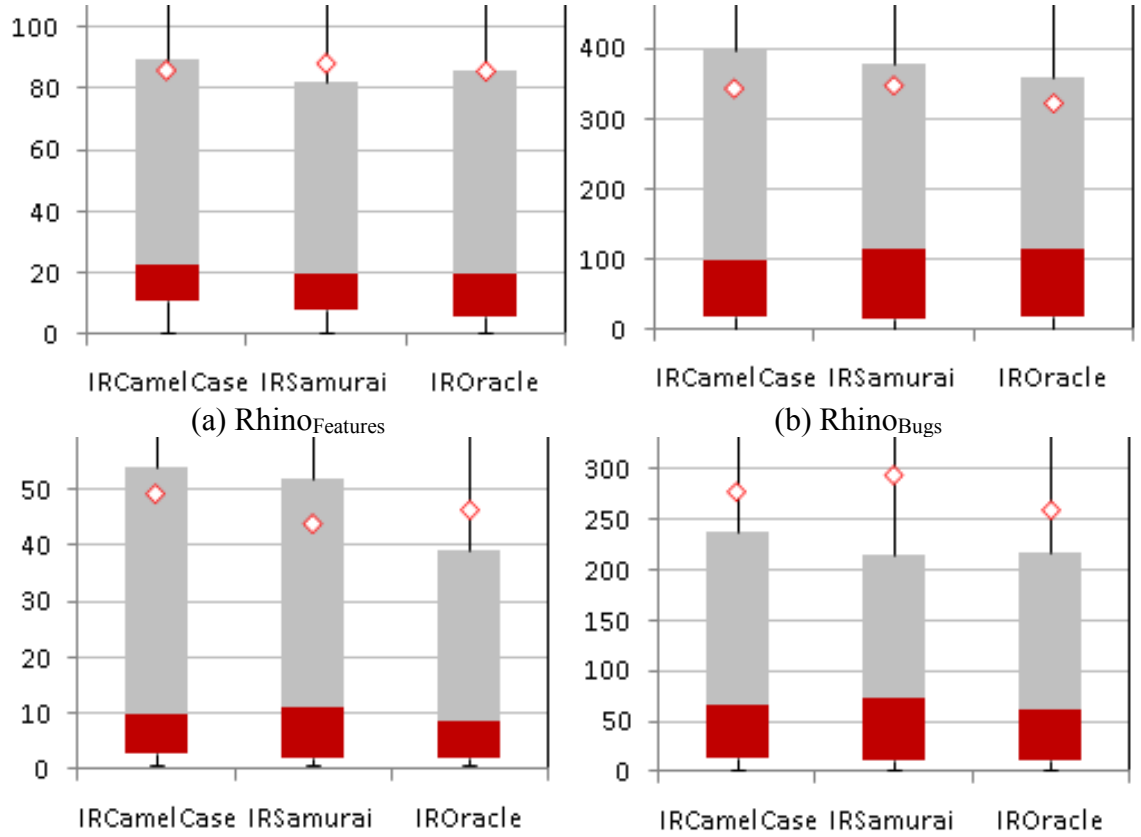
### 2.2.5 Hypotheses

We formulate several null hypotheses in order to test whether an improved splitting algorithm has a higher effectiveness measure than a simple splitting algorithm. For example:

**$H_{0,IRSamurai}$** There is no statistical significant difference in terms of effectiveness between $IR_{Samurai}$ and $IR_{CamelCase}$.

**$H_{0,IRSamuraiDyn}$** There is no statistical significant difference in terms of effectiveness between $IR_{Samurai}Dyn$ and $IR_{CamelCase}Dyn$.

We also define several alternative hypotheses for the case when a null hypothesis is rejected with high confidence. These alternative hypotheses state that an improved identifier splitting technique (e.g., Samurai, Oracle) would produce higher effectiveness than the baseline splitting technique (i.e., CamelCase). The following alternative hypotheses correspond to the null hypotheses defined above.




(a) $\text{Rhino}_{\text{Features}}$ (b) $\text{Rhino}_{\text{Bugs}}$

(c) $\text{jEdit}_{\text{Features}}$ (d) $\text{jEdit}_{\text{Bugs}}$

Figure 2-1 Box plots of the effectiveness measure of the three IR-based FLTs ($IR_{CamelCase}$, $IR_{Samurai}$ and $IR_{Oracle}$) for the four datasets: a) $\text{Rhino}_{\text{Features}}$, b) $\text{Rhino}_{\text{Bugs}}$, c) $\text{jEdit}_{\text{Features}}$ and d) $\text{jEdit}_{\text{Bugs}}$

**$H_{A,IRSamurai}$** $IR_{Samurai}$ has statistically significantly higher effectiveness than $IR_{CamelCase}$.

**$H_{A,IRSamuraiDyn}$** $IR_{Samurai}Dyn$ has statistically significantly higher effectiveness than $IR_{CamelCase}Dyn$.

The corresponding null and alternative hypotheses for the Oracle splitting technique are defined analogously.

## 2.3 Results and Discussion

This section presents the effectiveness measures of the FLTs presented in Table 2-1, which were applied on the four datasets (see Table 2-2) extracted from Rhino and jEdit. Please refer to our online appendix for complete data.

Figure 2-1 presents the box plots of the effectiveness measures of the three IR based FLTs applied on the four datasets. For each dataset, all the instances of the IR feature location technique produce very similar results in terms of lower quartile, median, mean, upper quartile, etc. For example, Figure 2-1 (a) shows that for the $\text{Rhino}_{\text{Features}}$ dataset, using the CamelCase splitting ($IR_{CamelCase}$) we obtain a median of 23 and an average of 86, and if we use the Oracle splitting ($IR_{Oracle}$), we obtain a median of 20 and an average of 86. The same small differences between the descriptive statistics measures are observed among all the IR instances, and in all the four datasets.

Similarly to Figure 2-1, Figure 2-2 presents the box plots of the effectiveness measure of the three IRDyn FLTs which were applied on the following three datasets: $\text{Rhino}_{\text{Features}}$ (Figure 2-2 (a)), $\text{jEdit}_{\text{Features}}$ (Figure 2-2 (b)) and $\text{jEdit}_{\text{Bugs}}$ (Figure 2-2 (c)). For all the datasets, the three FLTs produce almost identical results, regardless of the technique used for splitting the identifiers. For example, Figure 2-2 (a) shows that for the $\text{Rhino}_{\text{Features}}$ dataset, using CamelCase splitting ($IR_{CamelCase}Dyn$), the median and average are 9 and 30 respectively, whereas for Oracle splitting ($IR_{Oracle}Dyn$) the median and average are 8 and 32 respectively. The small differences observed on the IR based instances are also observed here. Even more so, for the other datasets, when incorporating dynamic information the differences produced by the feature location techniques seem to be less noticeable than the differences produced by IR-based feature location techniques. This fact may suggest that dynamic information has some influence and the splitting techniques used for identifiers may not be as important. It is also interesting to observe that feature location techniques applied on the datasets that use features as queries (i.e., $\text{Rhino}_{\text{Features}}$ and $\text{jEdit}_{\text{Features}}$) have lower effectiveness measures than the feature location techniques applied on the datasets that use bug descriptions as queries. For example, for Rhino, the median effectiveness when using feature descriptions as queries is about 21 (see

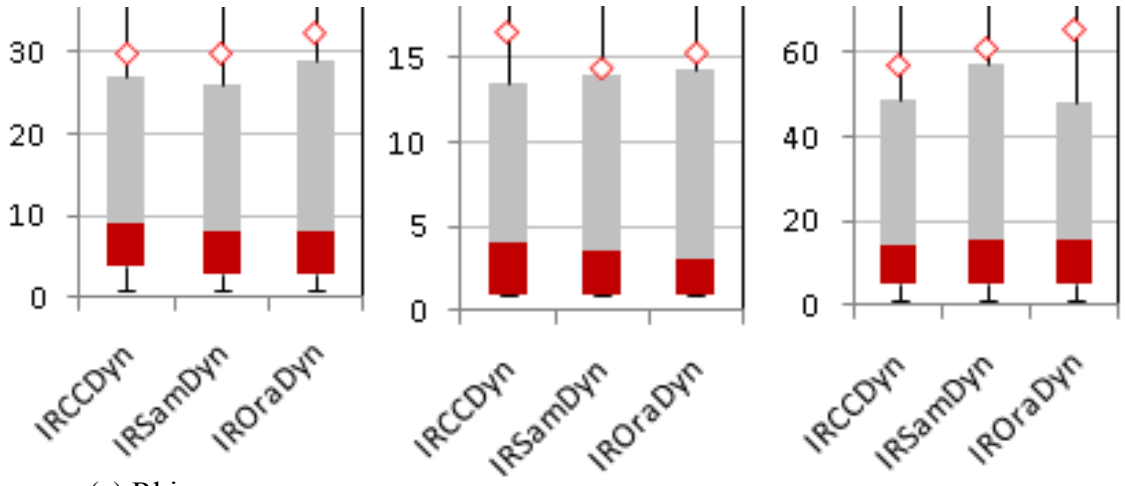
Figure 2-1 (a)), whereas the median effectiveness when using bug descriptions as queries is about 110 (see Figure 2-1 (b)). The same observation is valid for the jEdit when only textual information is used (see Figure 2-1 (c)(d)) as well as when textual and execution information are combined (see Figure 2-2 (b)(c)).

The results illustrated in Figure 2-1 and Figure 2-2 provide only a high level picture of the effectiveness measure. We now present results from a case by case comparison of the effectiveness measure. Table 2-4 presents the percentage of times an instance of the IR based FLT produces lower ranks than another instance of the IR based FLT. The first cell value represents the percentage of times the FLT from the corresponding row produces lower ranks than $IR_{CamelCase}$, whereas the number in parenthesis represents the percentage of times $IR_{CamelCase}$ produces lower ranks than the technique from the row (in the remaining percentages, the two techniques produce identical ranks). In this case, a higher percentage denotes a more effective technique. In addition, Table 2-4 shows the percentage of times the FLT form the row produces better results than $IR_{CamelCase}Dyn$.

We observe from Table 2-4 that comparing the effectiveness measures of $IR_{Oracle}$ and $IR_{CamelCase}$ side by side, $IR_{Oracle}$ produces lower ranks in 49% of cases, whereas $IR_{CamelCase}$ produces better results in 33% of cases. In the remaining 18% of cases (i.e., 100%-49%-33%) the two techniques produce identical ranks.

Similarly, from Table 2-4 we observe that when dynamic information is taken into account, for the $\text{Rhino}_{\text{Features}}$ dataset, $IR_{Oracle}Dyn$ produces lower ranks (i.e., better results) in 42% of cases, whereas $IR_{CamelCase}Dyn$ produces better results in 35% of cases. In the remaining 23% of cases (i.e., 100%-42%-35%) the techniques produce the same results.

It is interesting to observe that for both systems, $IR_{Oracle}$ and $IR_{Oracle}Dyn$ produce a higher percentage of good results than $IR_{CamelCase}$ and $IR_{CamelCase}Dyn$ respectively, when




(a) $\text{Rhino}_{\text{Features}}$ (b) $\text{jEdit}_{\text{Features}}$ (c) $\text{jEdit}_{\text{Bugs}}$

Figure 2-2 Box plots of the effectiveness measure of the three FLTs ($IR_{CamelCase}Dyn$ ($IR_{CC}Dyn$), $IR_{Samurai}Dyn$ ($IR_{Sam}Dyn$) and $IR_{Oracle}Dyn$ ($IR_{Ora}Dyn$)) for the 3 datasets: a) $\text{Rhino}_{\text{Features}}$, b) $\text{jEdit}_{\text{Features}}$ and c) $\text{jEdit}_{\text{Bugs}}$

these techniques are applied on the datasets that use features as queries (see columns two and four and rows two and four Table 2-4). However, when these techniques are applied on the datasets that use bug description as queries, the opposite phenomenon is observed. In other words $IR_{CamelCase}$ and $IR_{CamelCase}Dyn$ produce higher percentage of good results than $IR_{Oracle}$ and $IR_{Oracle}Dyn$ respectively (see columns three and five of and rows two and four Table 2-4).

The effectiveness measures presented as box plots and percentages are statistically analyzed using the Wilcoxon signed-rank test. Table 2-5 presents the p-values of the Wilcoxon signed-rank test for all the instances of the IR-based FLTs. The results that are statistically significant (i.e., the p-value is lower than $\alpha = 0.05$) are highlighted in bold. The table shows that there is only one instance when the Oracle splitting technique (i.e., $IR_{Oracle}$) produces results that are statistically significantly better than the technique that uses CamelCase splitting (i.e., $IR_{CamelCase}$). This is for the $\text{Rhino}_{\text{Features}}$ dataset and the p-value is equal to 0.005. We performed the same analysis between $IR_{Oracle}$ and $IR_{Samurai}$ and the results show that only for the $\text{Rhino}_{\text{Features}}$ dataset $IR_{Oracle}$ produces results that are statistically significantly better than $IR_{Samurai}$ (p-value=0.009). Refer to our online appendix for the data.

Similarly, Table 2-5 shows the p-values of the Wilcoxon signed-rank test applied on the effectiveness measures produced by the IRDyn FLTs. The results show that no technique produces statistically better results than any other technique. This observation helps in answering the research questions **RQ$_2$** and **RQ$_4$**, that the splitting technique used is not as important if dynamic information is considered. Refer to our online appendix for the results comparing $IR_{Oracle}Dyn$ and $IR_{Samurai}Dyn$. When dynamic information is involved, no technique produces statistically significant results than the other for any of the datasets.

If we look at the same results (i.e., the effectiveness measure) from three different points of view (i.e., box plots, percentages and statistical analysis), we derive the following conclusions. First, there are instances where a better identifier splitting technique (i.e., Oracle) improves feature location. This has been the case for the Rhino, for the Rhino$_{Features}$ dataset. Second, there are cases when even a perfect identifier splitting technique cannot help in the process of feature location. Such an example is given by the jEdit$_{Features}$ dataset, when the effectiveness measure is improved for a few cases, but the difference is not statistically significant. Moreover, there are instances where the perfect splitting technique can have negative impact on feature location, as it was the case for the jEdit$_{Bugs}$ dataset. In this case, the original CamelCase splitting technique produced better results than the Oracle in terms of percentages (see Table 2-4), but the difference is still not statistically significant. Finally, there is one instance, Rhino$_{Features}$, where splitting helps when textual information is used. However, when dynamic information is used, all the splitting techniques produce equivalent results from a statistical point of view.

Table 2-4 Percentages of times the effectiveness of the FLT from the row is higher than $IR_{CamelCase}$ (first two rows) and higher than $IR_{CamelCase}Dyn$ (last two rows), and vice-versa (see percentages from parenthesis)

| FLT | $\text{Rhino}_{\text{Features}}$ | $\text{Rhino}_{\text{Bugs}}$ | $\text{jEdit}_{\text{Features}}$ | $\text{jEdit}_{\text{Bugs}}$ |
|---|---|---|---|---|
| $IR_{Samurai}$ *vs* $IR_{CamelCase}$ | 39 (40) | 36 (48) | 33 (36) | 41 (41) |
| $IR_{Oracle}$ *vs* $IR_{CamelCase}$ | 49 (33) | 45 (48) | 44 (38) | 40 (55) |
| $IR_{Samurai}Dyn$ *vs* $IR_{CamelCase}Dyn$ | 33 (36) | N/A | 27 (22) | 28 (41) |
| $IR_{Oracle}Dyn$ *vs* $IR_{CamelCase}Dyn$ | 42 (35) | N/A | 34 (22) | 35 (50) |

Table 2-5 The p-values of the Wilcoxon signed-rank test for the FLT from the row compared with $IR_{CamelCase}$ (first two rows) and $IR_{CamelCase}Dyn$ (last two rows); statistical significance values are highlighted in **bold**

| FLT | $\text{Rhino}_{\text{Features}}$ | $\text{Rhino}_{\text{Bugs}}$ | $\text{jEdit}_{\text{Features}}$ | $\text{jEdit}_{\text{Bugs}}$ |
|---|---|---|---|---|
| $IR_{Samurai}$ *vs* $IR_{CamelCase}$ | 0.692 | 0.890 | 0.742 | 0.479 |
| $IR_{Oracle}$ *vs* $IR_{CamelCase}$ | **0.005** | 0.497 | 0.202 | 0.785 |
| $IR_{Samurai}Dyn$ *vs* $IR_{CamelCase}Dyn$ | 0.713 | N/A | 0.307 | 0.928 |
| $IR_{Oracle}Dyn$ *vs* $IR_{CamelCase}Dyn$ | 0.265 | N/A | 0.095 | 0.937 |

### 2.3.1 Qualitative Results

This section presents some observations after examining the results produced by the splitting techniques and after examining the queries.

One of the problems that we encountered using Samurai was that it tended to split certain types of identifiers into many meaningless terms, some of them having between one-three characters. Examples of identifiers from Rhino, where Samurai split them incorrectly were: *debugAccelerators*, *tolocale*, *imitating*, *imlementation*, etc. Their incorrect Samurai splitting was: *debug Ac ce le r at o rs*, *tol ocal e*, *imi ta ting*, *i ml eme n tat ion* (see Table 2-6). For these examples, CamelCase performed better, as it correctly split the first identifier (*debug accelerators*), but it left the other ones unaltered. Please refer to our online appendix for more information.

One of the benefits of using Samurai was that it accurately split same-case identifiers composed of multiple words. For these cases, CamelCase left the identifiers unmodified. Examples of such identifiers from Rhino include *SHORTNUMBER*, *readadapterobject*,

Table 2-6 Examples of splitted identifiers from Rhino using CamelCase and Samurai. The identifiers which are split correctly are highlighted in **bold**

| Original Identifier | CamelCase | Samurai |
|---|---|---|
| GETPROP | getprop | **GET PROP** |
| readadapterobject | readadapterobject | **read adapter object** |
| SHORTNUMBER | shortnumber | **SHORT NUMBER** |
| debugAccelerators | **debug accelerators** | debug Ac ce le r at o rs |
| tolocale | tolocale | tol ocal e |
| imitating | **imitating** | imi ta ting |

*GETPROP* which are correctly split by Samurai as *SHORT NUMBER*, *read adapter object*, and *GET PROP*, and are left unchanged by CamelCase (see Table 2-6).

However, there were some cryptic identifiers that were almost impossible to split using CamelCase or Samurai. Examples of such identifiers from Rhino include *ldbl*, *njm*, *pun*, *rve*, *wbdry*, etc. In these cases, inferring the meaning from the context in which these identifiers appeared was the only way to split or expand them correctly.

We observed a vocabulary mismatch problem, which produced inconsistencies between the identifiers used in the queries, and the identifiers used in the code.

This problem seemed to be less noticeable for features, and more severe for bugs. For jEdit, the issues that described features often contained terms that were later used in the code as identifiers for classes, methods, variables, etc. For example, jEdit's feature #1608486[6] ("Support 'thick' caret"), contained in its description many identifiers that were also found in the name of the methods (e.g., *thick*, *caret*, *text*, *area*, etc.). For features, their queries were expressive, and more consistent with the source code vocabulary, so they benefitted less from an Oracle splitting. Hence, when using feature descriptions as queries for both Rhino and jEdit, the median effectiveness of the FLTs, regardless of splitting, were about 20 for Rhino (see Figure 2-1 (a)) and about 10 for jEdit (see Figure 2-1 (c)).

[6] http://sourceforge.net/tracker/index.php?func=detail&aid=1608486&group_id=588&atid=300588

On the other hand, the vocabulary of the queries extracted from bug reports was less consistent with the source code vocabulary, and a splitting technique, helped bridge this gap. For example, jEdit's bug #1575505[7] ("C+j bug") reported a problem with the "join lines" implementation, yet nowhere in its description were the words *join* or *lines* mentioned. In general, the identifiers from the bug descriptions were less consistent with the code, and this issue was reflected in terms of the effectiveness measures produced by the FLTs, when these bug descriptions were used as queries. For example, in Figure 2-1 (b) the median effectiveness for Rhino system was about 110 (as opposed to a median of 20 when features were used as queries). Also, Figure 2-1 (d), shows that the median effectiveness of the techniques that used bugs as queries was around 67, as opposed to 10, which was the median effectiveness when features were used as queries.

Another problem with the queries is that some identifiers were used just for communication between developers, and no matter what splitting technique was used, these identifiers provided no useful information, because they appeared only in the query vocabulary, and did not appear at all in the source code vocabulary. Examples of such identifiers included words that are common in communication, such as *btw* (i.e., by the way), *thanks*, *hate*, *rant*, *greetings*, *fly*, *annoying*, etc., name of developers, *ApeHanger*, *Slava*, *Carlos*, etc.

### 2.3.2 Threats to Validity

In this section we present several threats to validity associated with the evaluation.

Threats to *construct validity* concern the relation between the theory and the observation. This threat is mainly due to mistakes in the Oracle and gold set. We cannot guarantee that no errors are present in the Oracle. As the intent of the Oracle is to explain identifier semantics, we cannot guarantee that some identifiers could have been split in

[7] http://sourceforge.net/tracker/index.php?func=detail&aid=1575505&group_id=588&atid=100588

different ways by developers that originally created them. This problem is difficult and it relates to guessing the developers' intent. To limit this threat, different sources of information such as comments, source code context, and online documentation were used when producing the Oracle. To minimize the risk on the accuracy of the gold set, we used data produced by other researchers, which was used in previous studies and made available to the research community.

Threats to *internal validity* concern any confounding factors that could have influenced our study results. In particular, these threats are due to the subjectivity of the manual building of the Oracle and to the possible biases introduced by manually splitting identifiers. To limit this threat, the Oracle was produced by a joint work among Latifa Guerrouj and Bogdan Dit (see Section 1.2), using CamelCase, Samurai and TIDIER. In addition, inconsistencies in splitting/mapping to dictionary words were discussed.

Threats to *conclusion validity* concern the relations between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used a non-parametric test (i.e., Wilcoxon signed-rank test), which does not make any assumptions on the underlying distributions of the data. Furthermore, as the only significant p-value is 0.005 (see Table 2-5), even with the conservative Bonferroni correction, it will remain significant as the limit in such case is equal to $\alpha$-value/number of tests (i.e., $\frac{0.05}{3} = 0.01666 < 0.05$).

Threats to *external validity* concern the possibility of generalizing our results. To make our results as generalizable as possible, we used two Java applications from two different application domains but we cannot be sure that our findings will be valid for other domains, applications, programming languages or software engineering tasks (i.e., different from feature location). More case studies are needed to confirm the results presented and to verify if indeed,

in the general case, dynamic information reduces the gain of more sophisticated identifier split techniques.

## 2.4 Related Work on Source Code Identifiers

Given the paramount role of source code identifiers in maintenance tasks such as traceability link recovery or feature and concept location, a large body of relevant work is available in this area. We divided this section into the related work on the role of unstructured information in program comprehension and approaches to feature location.

### 2.4.1 The Role of Unstructured Information in Program Comprehension

Takang et al. [172] attempted to determine the informativeness of identifiers. They conducted experiments to compare abbreviated identifiers to full-word identifiers and uncommented code to commented code. Their study results showed that commented programs are more understandable than non-commented programs and that programs containing full-word identifiers are more understandable than those with abbreviated identifiers.

Lawrie et al. [97, 98] have performed an empirical study to assess the quality of source code identifiers. Their study with over 100 programmers indicated that full words as well as recognizable abbreviations lead to better comprehension. Lawrie et al. [96] introduced GenTest, a splitting algorithm which by incorporating vocabulary normalization is able to outperform Samurai.

Binkley et al. [19] have investigated the use of different identifier separators in program comprehension. They found that the CamelCase convention led to better understanding than underscores and, when subjects are properly trained, they performed faster with identifiers in the CamelCase style rather than identifiers built using underscores. Binkley et al.'s study was replicated by Sharif and Maletic [165] using an eye tracking system. Their results indicate that

there was no difference in terms of accuracy between the CamelCase and underscore style, and that subjects recognized identifiers that used the underscore notation more quickly.

Caprile and Tonella [26] have analyzed the internal structure of identifiers. Their in-depth analysis showed that identifiers are one of the most important source of information about system concepts, and that the information carried by identifiers is often the starting point for program comprehension.

Deißenböck and Pizka [43] have provided guidelines for the production of high-quality identifiers. With such guidelines, identifiers should contain enough information for an engineer to understand the program concepts.

### 2.4.2 Related Work on Feature Location

Marcus et al. introduced LSI-based feature location technique [115]. This approach was later extended to include the Rocchio algorithm for relevance feedback [63] by allowing developers to reformulate search queries. Grant et al. [71] used Independent Component Analysis for feature location, by separating the features (modeled as input signals) into independent components and estimating the relevance to each source code method. Shepherd et al. [167] proposed an approach to feature location that is based on the program model that captures action-oriented relations between identifiers in a program.

There are several FLTs that use more than one type of information (or underlying analysis). For example, SITIR [105] and PROMESIR [143] both utilize textual and execution information. Eisenbarth et al. [59] proposed a technique that applies formal concept analysis to traces to generate a mapping between features and methods. Cerberus [57] is another hybrid technique which combines static, dynamic and textual analysis. Revelle et al. [156] incorporated the information resulting from web mining algorithms applied on execution traces

and combined it with textual information to support feature location. A recent survey summarizing feature location approaches can be found in [52].

## 2.5 Discussion

Perfecting splitting techniques can improve the accuracy of feature location, easing program comprehension and thus, software evolution. In situations where execution information cannot be collected (e.g., mission critical and time critical applications), the benefits of using advanced splitting techniques can be mostly visible. In fact, by splitting source code identifiers and mapping them to domain concepts, the localization of entities contributing to implementing some user observable functionality may be easier, which could minimize feature location effort.

In this chapter, we presented an exploratory study of two FLTs (i.e., IR and IRDyn) for locating bugs and features, utilizing three strategies for splitting identifiers: CamelCase, Samurai and manual splitting of identifiers. These FLTs and their preprocessing techniques were evaluated on two open-source systems, Rhino and jEdit, and compared in terms of their effectiveness measure.

The results of the IR-based FLT reveal that Samurai and CamelCase produce similar results. However, the $IR_{Oracle}$ outperforms $IR_{CamelCase}$ in terms of the effectiveness measure, on the $\text{Rhino}_{\text{Features}}$ dataset. This supports our conjecture that when only textual information is available, an improved splitting technique can help improve effectiveness of feature location. The results also show that when both textual and execution information are used, any splitting algorithm will suffice, as FLTs produce equivalent results. In other words, because execution information helps pruning the search space considerably, the benefit of an advanced splitting algorithm is comparably smaller than the benefit obtained from execution information; hence the splitting algorithm will have little impact on the final results. Overall, our findings outline

potential benefits of creating advanced preprocessing techniques as they can be useful in situations where execution information cannot be easily collected.

# 3 Configuring Latent Dirichlet Allocation: LDA-GA

A significant amount of research on applying Information Retrieval methods for analyzing textual information in software artifacts [21] has been conducted in the SE community in recent years. Among the popular and promising IR techniques used, we enumerate Latent Semantic Indexing [42] and Latent Dirichlet Allocation [22]. The latter is a probabilistic statistical model that estimates distributions of latent topics from textual documents. It assumes that these documents have been generated using the probability distribution of these topics, and that the words in the documents were generated probabilistically in a similar manner. A number of approaches using LSI and LDA have been proposed to support software engineering tasks: feature location [52], change impact analysis [64], bug localization [108], clone detection [171], traceability link recovery [9, 111], expert developer recommendation [89], code measurement [106, 148], artifact summarization [39], and many others [11, 81, 159]. In all these approaches, LDA and LSI have been used on software artifacts in a similar manner as they were used on natural language documents (i.e., using the same settings, configurations and parameters) because the underlying assumption was that source code (or other software artifacts) and natural language documents exhibit similar properties. More specifically, applying LDA requires setting the number of topics and other parameters specific to the particular LDA implementation. For example, the fast collapsed Gibbs sampling generative model for LDA requires setting the number of iterations $n$ and the Dirichlet distribution parameters $\alpha$ and $\beta$ [137]. Even though LDA was successfully used in the IR and natural language analysis community, applying it on software data, using the same parameter values used for natural language text, did not always produce the expected results [132]. As in the case of machine learning and optimization techniques, a poor parameter

calibration or wrong assumptions about the nature of the data could lead to poor results [7]. Recent research has challenged this assumption and showed that text extracted from source code is much more repetitive and predictable as compared to natural language text [80]. According to recent empirical findings, "*corpus-based statistical language models capture a high level of local regularity in software, even more so than in English*" [80]. This fundamental new research finding explains in part why these fairly sophisticated IR methods showed rather low performance when applied on software data using parameters and configurations that were generally applicable for and tested on natural language corpora. This dissertation builds on the finding that text in software artifacts has different properties, as compared to natural language text, thus, we need new solutions for calibrating and configuring LDA and LSI to achieve better (acceptable) performance on software engineering tasks. In addition, this chapter introduces LDA-GA, a novel solution that uses a Genetic Algorithm [82] to determine the near-optimal configuration for LDA in the context of three different software engineering tasks, namely (1) traceability link recovery, (2) feature location, and (3) software artifacts labeling. Our contributions are summarized as follows.

We introduced LDA-GA, a novel and theoretically sound approach for calibrating LDA on software text corpora using a GA, and we show that it can be applied successfully on three software engineering tasks: traceability link recovery, feature location and software artifact labeling.

We conducted several experiments to study the performance of LDA configurations based on LDA-GA with those previously reported in the literature; to perform such a comparison we replicated previously published case studies.

We compared LDA-GA with existing heuristics for calibrating LDA; the empirical results demonstrate that our proposed approach is able to identify LDA configurations that lead to better accuracy as compared to existing heuristics.

We make publicly available in our online appendix [1] all the data, results and algorithms used in our studies, for replication purposes and to support future studies.

## 3.1 Background and Related Work

This section provides background information on (i) LDA and (ii) its applications to software engineering tasks, as well as (iii) discussions about related approaches aimed at determining the best configuration for LDA.

### 3.1.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation [22] is an IR model that allows to fit a generative probabilistic model from the term occurrences in a corpus of documents. Specifically, given a collection of documents, the IR process generates a $m \times n$ term-by-document matrix $M$, where $m$ is the number of terms occurring in all artifacts, and $n$ is the number of artifacts in the repository. A generic entry $w_{ij}$ of this matrix denotes a measure of the weight (i.e., relevance) of the $i^{th}$ term in the $j^{th}$ document [10]. One of the most used weighting schemas, which we also applied in this dissertation, is the *tf-idf* since it gives more importance to words having high frequency in a document and appearing in a small number of documents [21]. Then, the term-by-document matrix is taken as an input by LDA, which identifies the latent variables (topics) hidden in the data and generates as output a $k \times n$ matrix $\theta$, called *topic-by-document matrix*, where $k$ is the number of topics and $n$ is the number of documents. A generic entry $\theta_{ij}$ of such a matrix denotes the probability of the $j^{th}$ document to belong to the $i^{th}$ topic. Since typically $k \ll m$, LDA is mapping the documents from the space of terms ($m$) into a smaller space of topics ($k$). The latent topics allow us to cluster them on the basis of their shared topics. More specifically, documents having the same relevant topics are grouped in the same cluster, and documents having different topics belong to different clusters.

LDA requires as input a set of hyper-parameters (i.e., a set of parameters that have a smoothing effect on the topic model generated as output). In this dissertation we used the fast collapsed Gibbs sampling generative model for LDA because it provides the same accuracy as the standard LDA implementation, yet it is much faster [137]. For such an implementation, the set of hyper-parameters are:

- $k$, which is the number of topics that the latent model should extract from the data. To some extent this is equivalent to the number of clusters in a clustering algorithm;
- $n$, which denotes the number of Gibbs iterations, where a single iteration of the Gibbs sampler consists of sampling a topic for each word;
- $\alpha$, which influences the topic distributions per document. A high $\alpha$ value results in a better smoothing of the topics for each document (i.e., the topics are more uniformly distributed for each document);
- $\beta$, which affects the term's distribution per topic. A high $\beta$ value results in a more uniform distribution of terms per topic.

Note that $k$, $\alpha$, and $\beta$ are the parameters of any LDA implementation, while $n$ is an additional parameter required by the Gibbs sampling generative model.

### 3.1.2 LDA Applications to Software Engineering

Some recent applications of LDA to SE tasks operate on models of software artifacts (e.g., source code) rather than directly on those artifacts. Approaches that generate these models require as input a corpus (i.e., a document collection) that represents the software artifacts being analyzed. The corpus is constructed from textual information embedded in the artifacts, including identifiers and comments. While a number of different SE tasks have been supported using advanced textual retrieval techniques, such as LDA, the common problem

remains: the way LDA is commonly configured is based on the assumption that the underlying corpus is composed of natural language text. In our survey of the literature, the following SE tasks have been supported using LDA and all of these papers and approaches used ad-hoc heuristics to configure LDA, perhaps resulting in suboptimal performance in virtually all the cases: feature location [17], bug localization [108], impact analysis [130], source code labeling [39], aspect identification [11], expert identification [104], software traceability [9, 35, 65, 67, 161], test case prioritization [177], refactoring [14, 15, 131], mining API usage examples [126] and discussion forums [103], and evolution analysis [81, 176].

### 3.1.3 Approaches for Estimating the Parameters of LDA

Finding an LDA configuration that provides the best performance is not a trivial task. Although heuristics have been proposed [70, 74], these approaches focus only on identifying the number of topics that would result in the best performance of a task, while ignoring all the other parameters that are required to apply LDA in practice. Moreover, such approaches have not been evaluated on real SE applications or have been defined for natural language documents only, thus, they may not be applicable for software corpora.

One such technique is based on a heuristic for determining the "optimal" number of LDA topics for a source code corpus of methods by taking into account the location of these methods in files or folders, as well as the conceptual similarity between methods [70],. However, the utility of this heuristic was not evaluated in the context of specific SE tasks.

On a more theoretical side, a non-parametric extension of LDA called Hierarchical Dirichlet Processes [175] tries to infer the optimal number of topics automatically from the input data. Griffiths and Steyvers [74] proposed a method for choosing the best number of topics for LDA among a set of predefined topics. Their approach consists of (i) choosing a set of topics, (ii) computing a posterior distribution over the assignments of words to topics

$P(z|w,T)$, (iii) computing the harmonic mean of a set of values from the posterior distribution to estimate the likelihood of a word belonging to a topic (i.e., $P(w|T)$), and (iv) choosing the topic with the maximum likelihood. In their approach, the hyper-parameters $\alpha$ and $\beta$ are fixed, and only the number of topics is varied, which in practice, is not enough to properly calibrate LDA. In our approach, we vary all the parameters (i.e., $k$, $n$, $\alpha$ and $\beta$), to find a (near) optimal configuration for LDA.

## 3.2 Finding a (near) Optimal LDA Configuration

As explained in Section 3.1, LDA (and in particular its implementation based on fast collapsed Gibbs sampling generative model) requires the calibration of four parameters, $k$, $n$, $\alpha$ and $\beta$. Without a proper calibration, or with an ad-hoc calibration of these parameters, LDA's performance may be sub-optimal. Finding the best configuration of these parameters poses two problems.

First, we need a measure that can be used to assess the performances of LDA before applying it to a specific task (e.g., traceability link recovery). This measure should be independent from the supported SE task. In other words, we cannot simply train an LDA model on the data for one particular task, since obtaining such data means solving the task. For example, for traceability link recovery, if we identify all the links to assess the quality of the LDA model for extracting the links themselves, then there is no need to have an LDA-based model to recover these links anymore. In other words, we need to build such a model on raw data (e.g., source code and documentation) without having additional information about the links.

Second, we need an efficient way to find the best configuration of parameters, as an exhaustive analysis of all possible combinations is impractical due to (i) the combinatorial nature of the problem (i.e., the large number of possible configuration values for the LDA

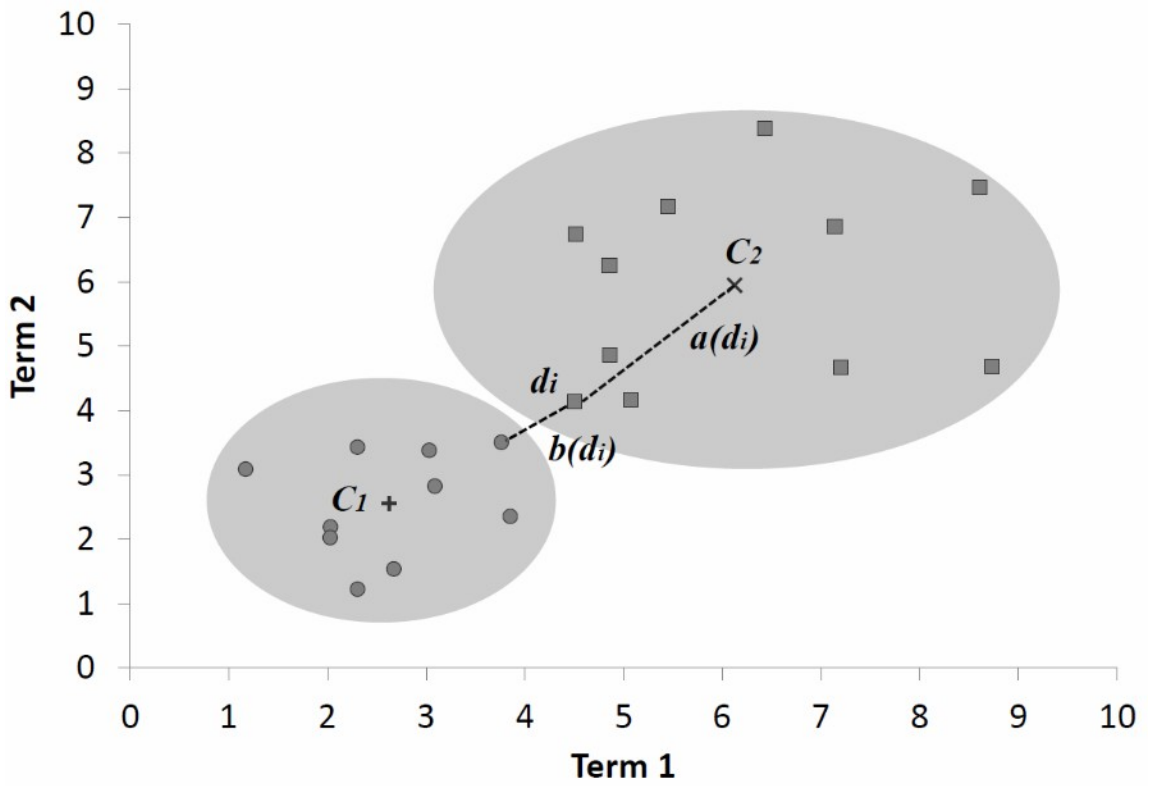



Figure 3-1 Example of Silhouette coefficient.

parameters), as well as (ii) the large amount of computational time required for even such a configuration. In the next subsections, we present our approach that addresses these problems, called LDA-GA, which is able to find a near-optimal configuration for the parameters of LDA.

### 3.2.1 Assessing the Quality of an LDA Configuration

LDA can be considered as a topic-based clustering technique, which can be used to cluster documents in the topics space using the similarities between their topics distributions. Our conjecture is that there is a strong relationship between the performances obtained by LDA on software corpora and the quality of clusters produced by LDA. Thus, measuring the quality of the produced clusters could provide some insights into the accuracy of LDA when applied to software engineering tasks. Indeed, if the quality of the clusters produced by LDA is poor, it means that LDA was not able to correctly extract the dominant topics from the software

corpus, because the documents, which are more similar to each other, are assigned to different clusters (i.e., LDA assigns different dominant topics to neighboring documents).

In this dissertation, we use the concept of a dominant topic to derive the textual clustering generated by a particular LDA configuration applied on a term-by-document matrix. Formally, the concept of a dominant topic can be defined as follows:

**Definition 1**. *Let $\theta$ be the topic-by-document matrix generated by a particular LDA configuration $P = [k, n, \alpha, \beta]$. A generic document $d_j$ has a dominant topic $t_i$, if and only if $\theta_{i,j} = \max\{\theta_{h,j}, h = 1 \dots k\}$.*

Starting from the definition of the dominant topic, we can formalize how LDA clusters documents within the topic space (the number of clusters is equal to the number of topics) as follows:

**Definition 2**. *Let $\theta$ be the topic-by-document matrix generated by a particular LDA configuration $P = [k, n, \alpha, \beta]$. A generic document $d_j$ belongs to the $i^{th}$ cluster, if and only if $t_i$ is the dominant topic of $d_j$.*

Thus, we can define a cluster as a set of documents in which each document is closer (i.e., shares the same dominant topic) to every other document in the cluster, and it is further from any other document from the other clusters. It is worth noting that the concept of a dominant topic is specific to software documents only. Collections of natural language documents are usually heterogeneous, meaning that documents can contain information related to multiple topics. In source code artifacts, heterogeneity is not always present, especially when considering single classes. More specifically, a class is a crisp abstraction of a domain/solution object, and should have a few, clear responsibilities. Hence, software documents should be clustered considering only the dominant topic, assuming that each document is related to only one specific topic.

Different LDA configurations provide different clustering models of the documents. However, not all clustering models that can be obtained by configuring LDA are good. There are two basic ways to evaluate the quality of a clustering structure: *internal criteria*, based on similarity/dissimilarity between different clusters and *external criteria*, which uses additional and external information (e.g., using judgment provided by users) [95]. Since the internal criterion does not require any manual effort and it is not software engineering task dependent, in this dissertation we use the *internal criteria* for measuring the quality of clusters. More specifically, we use two types of internal quality metrics: *cohesion* (similarity), which determines how closely related the documents in a cluster are, and *separation* (dissimilarity), which determines how distinct (or well-separated) a cluster is from other clusters [95].. Since these two metrics are contrasting each other, we use a popular method for combining both cohesion and separation in only one scalar value, called *Silhouette coefficient* [95]. The Silhouette coefficient is computed for each document using the concept of centroids of clusters. Formally, let $C$ be a cluster; its centroid $Centroid(C)$ is equal to the mean vector of all documents belonging to $C$:

$$Centroid(C) = \sum_{d_i \in C} \frac{d_i}{|C|}$$

Starting from the definition of centroids, the computation of the Silhouette coefficient consists of the following three steps:

1. For document $d_i$, calculate the maximum distance from $d_i$ to the other documents in its cluster. We call this value $a(d_i)$.
2. For document $d_i$, calculate the minimum distance from $d_i$ to the centroids of the clusters not containing $d_i$. We call this value $b(d_i)$.
3. For document $d_i$, the Silhouette coefficient $s(d_i)$ is:

$$s(d_i) = \frac{b(d_i) - a(d_i)}{\max\big(a(d_i), b(d_i)\big)}$$

The value of the Silhouette coefficient ranges between $[-1 \ldots 1]$. A negative value is undesirable, because it corresponds to the case in which $a(d_i) > b(d_i)$, i.e., the maximum distance to other documents in the cluster is greater than the minimum distance to other documents in other clusters. For measuring the distance between documents we used the Euclidean distance, since it is one of the most commonly used distance functions for data clustering [95]. Figure 3-1 provides a graphical interpretation of the Silhouette coefficient computed for a document $d_i$. In particular, it represents an example of a good Silhouette coefficient, since $d_i$ is close to the furthest document situated in its cluster, and far from the centroid of the nearest cluster. In the end, the overall measure of the quality of a clustering $C = \{C_1, \ldots, C_k\}$ can be obtained by computing the mean Silhouette coefficient of all documents. Let $C = \{C_1, \ldots, C_k\}$ be the clustering obtained using a particular LDA configuration, and let $M$ be a $m \times n$ term-by-document matrix. The mean Silhouette coefficient is computed as:

$$s(C) = \frac{1}{n}\sum_{i=1}^{n} s(d_i)$$

In this dissertation, we used the mean Silhouette coefficient as the measure for predicting the accuracy of LDA in the context of specific software engineering tasks.

### 3.2.2 Finding a (Near) Optimal LDA Configuration

Based on the conjecture that the higher the clustering quality produced by LDA, the higher the accuracy of LDA when used for software engineering tasks, we present an approach to efficiently identify the LDA configuration $P = [k, n, \alpha, \beta]$ that maximizes the overall quality (measured using the mean Silhouette coefficient) of the clustering produced by LDA. For solving such an optimization problem we applied Genetic Algorithms (GA) which is a stochastic search technique based on the mechanism of a natural selection and natural genetics.

Table 3-1 Characteristics of the systems used in the three evaluation scenarios: Traceability Link Recovery (top), Feature Location (middle) and Software Artifact Labeling (bottom)

Scenario I – Traceability Link Recovery

| System | KLOC | Source Artifacts (#) | Target Artifacts (#) | Correct Links |
|---|---|---|---|---|
| EasyClinic | 20 | Use Case (30) | Code Class (47) | 93 |
| eTour | 45 | Use Case (58) | Code Class (174) | 366 |

Scenario II – Feature Location

| System | KLOC | # Classes | # Methods | # Features |
|---|---|---|---|---|
| jEdit v4.3 | 104 | 503 | 6,413 | 150 |
| ArgoUML v0.22 | 149 | 1,439 | 11,000 | 91 |

Scenario III – Software Artifact Labeling

| System | KLOC | # Classes | Sampled Classes |
|---|---|---|---|
| JHotDraw | 29 | 275 | 10 |
| eXVantage | 28 | 348 | 10 |

Since the introduction of GA by Holland [82] in the 1970s, this algorithm has been used in a wide range of applications where optimization is required and finding an exact solution is NP-Hard. The advantage of GA with respect to the other search algorithm is its intrinsic parallelism, i.e., having multiple solutions (individuals) evolving in parallel to explore different parts of the search space.

The GA search starts with a random population of solutions, where each individual (i.e., *chromosome*) from the population represents a solution of the optimization problem. The population evolves through subsequent generations and, during each generation, the individuals are evaluated based on the *fitness* function that has to be optimized. For creating the next generation, new individuals (i.e., *offsprings*) are generated by (i) applying a *selection operator*, which is based on the fitness function, for the individuals to be reproduced, (ii) recombining, with a given probability, two individuals from the current generation using the *crossover operator*, and (iii) modifying, with a given probability, individuals using the *mutation operator*. More details about GA can be found in a book by Goldberg [68].

In this dissertation, we used a simple GA [68] with elitism of two individuals (i.e., the two best individuals are kept alive across generations). Individuals (solutions) are represented as an array with four floats, where each element represents $k$, $n$, $\alpha$ and $\beta$, respectively. Thus, an individual (or chromosome) is a particular LDA configuration and the population is represented by a set of different LDA configurations. The selection operator is the Roulette wheel selection, which assigns to the individuals with higher fitness a higher chances to be selected. The crossover operator is the arithmetic crossover, which creates new individuals by performing a linear combination (with random coefficients) of the two parents. The mutation operator is the uniform mutation, which randomly changes one of the genes (i.e., one of the four LDA parameter values) of an individual, with a different parameter value within a specified range. The fitness function that drives the GA evolution is the Silhouette coefficient described in Section 3.2.1.

Our GA approach can be briefly summarized as (i) generating LDA configurations, (ii) using them to cluster documents, (iii) evaluating the cluster quality using the Silhouette coefficient, and (iv) using that value to drive the GA evolution.

## 3.3 Empirical Study Design

This section describes the design of the empirical studies that we conducted in order to evaluate LDA-GA in the context of three software engineering tasks. The studies aim at answering the following research questions:

**RQ$_1$**: *What is the impact of the configuration parameters on LDA's performance in the context of software engineering tasks?* This research question aims at justifying the need for an automatic approach that calibrates LDA's settings when LDA is applied to support SE tasks. For this purpose, we analyzed a large number of LDA configurations for three software

engineering tasks. The presence of a high variability in LDA's performances indicates that, without a proper calibration, such a technique risks being severely under-utilized.

**$RQ_2$**: *Does LDA-GA, our proposed GA-based approach, enable effective use of LDA in software engineering tasks?* This research question is the main focus of our study, and it is aimed at analyzing the ability of LDA-GA to find an appropriate configuration for LDA, which is able to produce good results for specific software engineering tasks.

We address both research questions in three different scenarios, representative of SE tasks that can be supported by LDA: traceability link recovery, feature location, and software artifact labeling. LDA was previously used in some of these tasks [9, 39, 176]. For our data and results please visit our online appendix.

### 3.3.1 Scenario I: Traceability Links Recovery

In this scenario, we used LDA to recover links between documentation artifacts (e.g., use cases) and code classes. The experiment has been conducted on software repositories from two projects, EasyClinic and eTour. EasyClinic is a system used to manage a doctor's office, while eTour is an electronic touristic guide. Both systems were developed by the final year Master's students at the University of Salerno (Italy). The documentation, source code identifiers, and comments for both systems are written in Italian. The top part of Table 3-1 reports the characteristics of the considered software systems in terms of type, number of source and target artifacts, as well as Kilo Lines of Code (KLOC). The table also reports the number of correct links between the source and target artifacts. These correct links, which are derived from the traceability matrix provided by the original developers, are used as an *oracle* to evaluate the accuracy of the proposed traceability recovery method.

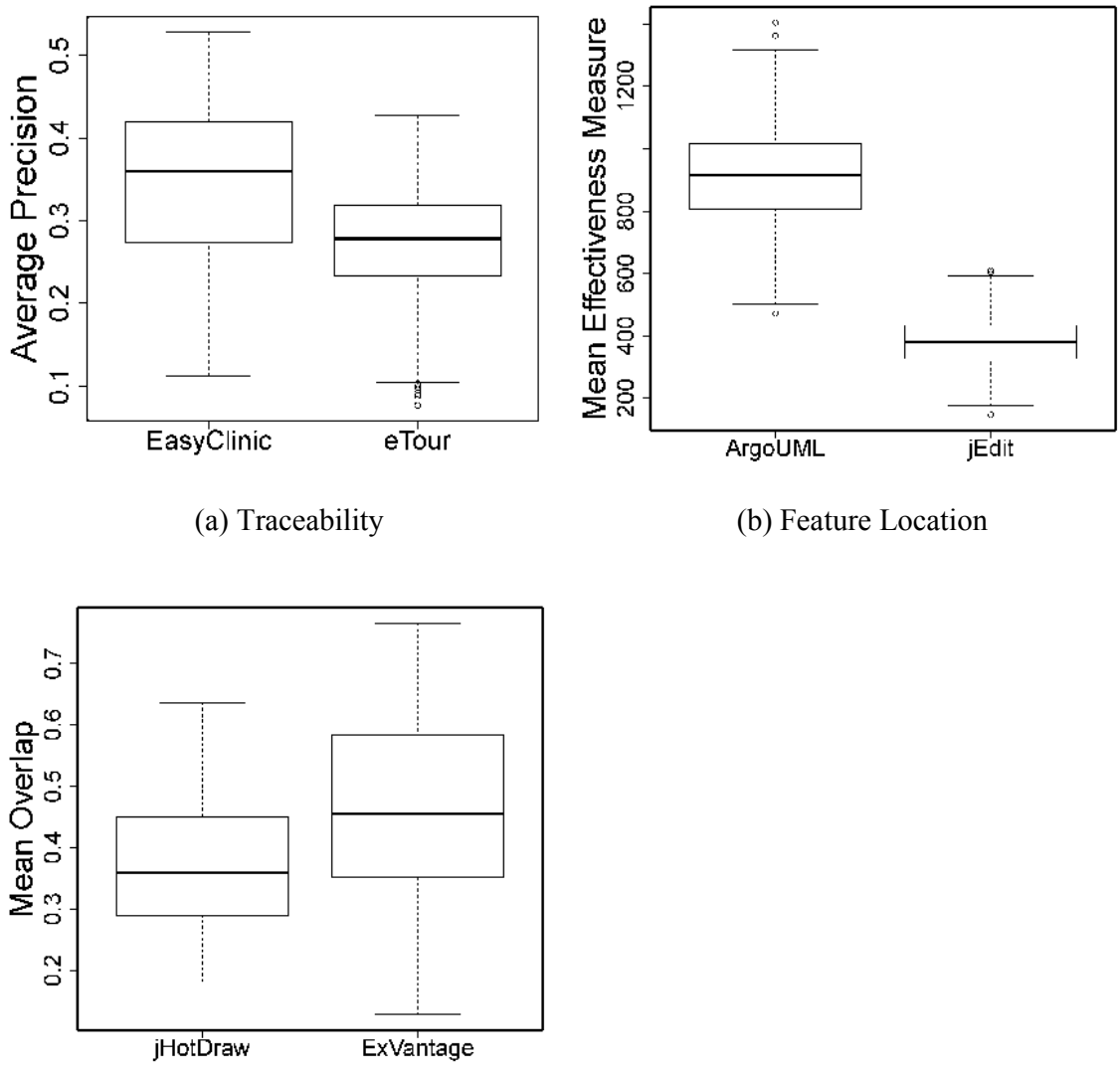
To address **$RQ_1$**, we compared the accuracy of recovering traceability links using different configurations for LDA. Specifically, we varied the number of topics from 10 to 100

with step 10 on EasyClinic, and from 10 to 200 with step 10 on eTour. We varied $\alpha$ and $\beta$ from 0 to 1 with 0.1 increments, and we exercised all possible combinations of such values. We fixed the number of iterations to 500, which resulted to be a sufficient number of iterations for the model to converge. Thus, the total number of trials performed on EasyClinic and eTour were 1,000 and 2,000, respectively. Clearly, although combinatorial, such an analysis is not exhaustive, as it considers a discrete set of parameter values and combinations. For **RQ$_2$**, we compared the accuracy achieved by LDA when the configuration is determined using LDAGA with (i) the best accuracy achieved by LDA (determined when answering RQ1) and (ii) the accuracy achieved by LDA on the same system in the previously published studies where an "ad-hoc" configuration was used [65]. While the former comparison is more of a sanity check aimed at analyzing the effectiveness of the GA in finding a near-optimal solution, the latter comparison was aimed at analyzing to what extent LDA-GA is able to enrich the effectiveness and usefulness of LDA in the context of traceability link recovery when properly calibrated.

When addressing **RQ$_1$**, we evaluated LDA's recovery accuracy using the average precision metric [10], which returns a single value for each ranked list of candidate links provided. For **RQ$_2$**, we used two well-known IR metrics: precision and recall [10]. The precision values achieved for different configurations (over different levels of recall) are then pairwise-compared using the Wilcoxon rank sum test. Since this requires performing three tests for each system, we adjusted the p-values using Holm's correction procedure [83]. This procedure sorts the p-values resulting from $n$ tests in ascending order, multiplying the smallest by $n$, the next by $n - 1$, and so on.

### 3.3.2 Scenario II: Feature Location

In this scenario, we used LDA to locate features within the textual corpus of source code. The context of this scenario is represented by two software systems, jEdit v4.3 [85] and




(a) Traceability

(b) Feature Location

(c) Artifact Labeling

Figure 3-2 Variability of performance achieved by LDA configurations for (a) traceability link recovery, (b) feature location, and (c) software artifact labeling

ArgoUML v0.22 [8]. jEdit is an open-source text editor for programmers, while ArgoUML is a well-known UML editor. Table 3-1 (middle) reports the characteristics of the considered software systems in terms of number of classes, number of methods, as well as KLOC and the number of features to be located. These software systems have been used in previous studies on feature location [17, 53]. For more information about how the datasets were generated refer to [46, 52] and Appendix A.

To answer **RQ1**, we compared the effectiveness measure of LDA using different configurations. Specifically, we varied the number of topics from 50 to 500 with step 50 for both ArgoUML and jEdit. We varied $\alpha$ and $\beta$ from 0 to 1 with 0.1 increments. Similarly to the traceability task, we fixed the number of iterations to 500. We exercised all possible combinations of such values. Thus, the total number of trials performed on both software systems consisted of 1,000 different LDA combinations. For **RQ2**, similarly to the previous scenario, we compared the performance achieved by LDA-GA with (i) the best performance achieved by LDA when answering **RQ1** and (ii) the performance obtained by LDA using the *source locality* heuristic proposed by Grant and Cordy for the feature location task [70]. The performance of LDA in this scenario was analyzed using the *effectiveness measure* (*EM*) [143]. Given a feature of interest, this measure estimates the number of methods a developer needs to inspect before finding a method relevant to that feature (the list of methods are ranked by their similarity to the description of the feature). A lower value for the EM indicates less effort (i.e., fewer methods to analyze before finding a relevant one). The EM computed for different configurations on different queries (i.e., feature descriptions) were then pairwise-compared using a Wilcoxon rank sum test, similarly to the evaluation from Scenario I and, also in this case, the p-values were adjusted using Holm's procedure.

### 3.3.3 Scenario III: Software Artifact Labeling

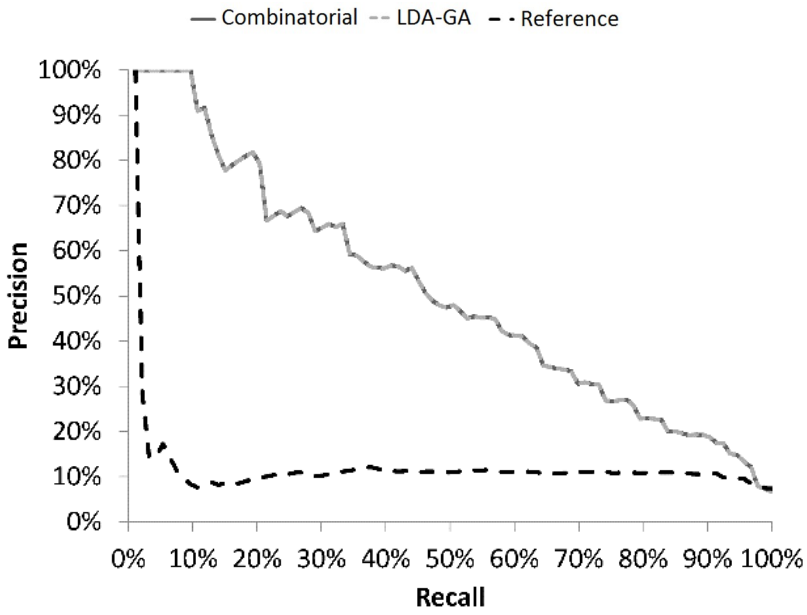
In this scenario, we used LDA to automatically "label" source code classes using representative words. Specifically, we extracted topics from a single class (using LDA), and then we ranked all the words characterizing the extracted topics according to their probability in the obtained topic distribution. The top ten words belonging to the topic with the highest probability in the obtained topic distribution were then used to label the class [39].

The study was conducted on ten classes from JHotDraw and ten classes from eXVantage. The former is an open-source drawing tool, and the latter is a novel testing and generation tool. Their characteristics are summarized in the bottom part of Table 3-1. For the sampled classes, we had user-generated labels from a previously published work [39], and these represented our "ideal" labels. After obtaining the LDA labels we compared them to the user-generated ones and computed the overlap between them. The overlap was measured using the asymmetric Jaccard measure [10]. Formally, let $K(C_i) = \{t_1, \ldots, t_m\}$ and $K_{m_i}(C_i) = \{t_1, \ldots, t_h\}$ be the sets of keywords identified by subjects and the technique $m_i$, respectively, to label the class $C_i$. The overlap was computed as follows:

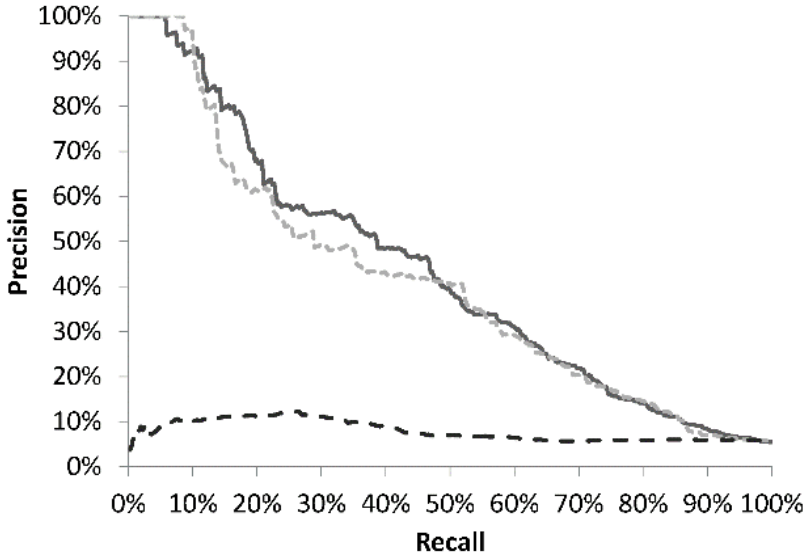
$$overlap_{m_i}(C_i) = \frac{\left|K(C_i) \cap K_{m_i}(C_i)\right|}{K_{m_i}(C_i)}$$

Note that the size of $K(C_i)$ might be different from the size of $K_{m_i}(C_i)$. In particular, while the number of keywords identified by LDA is always ten (by construction we set $h = 10$), the number of keywords identified by subjects could be more or less than ten (generally it is ten, but there are few cases where the number is different). For this reason, we decided to use the asymmetric Jaccard to avoid penalizing too much the automatic method when the size of $K(C_i)$ is less than ten.

Also in this scenario, in order to address **RQ1** we compared the recovery accuracy of LDA using different settings. Specifically, we varied the number of topics from 10 to 50 with step 10 for both JHotDraw and eXVantage. As for $\alpha$ and $\beta$, we varied them between 0 and 1 by increments of 0.1. We fixed the number of iterations to 500 as in the previous two tasks. We exercised all possible combinations of such values. Thus, the total number of trials performed on JHotDraw and eXVantage was 500 on both systems. For RQ2, we compared the




(a) EasyClinic




(b) eTour

Figure 3-3 Precision and Recall graphs for traceability link recovery systems: (a) EasyClinic and (b) eTour

accuracy achieved by LDA-GA with (i) the best accuracy achieved by LDA while iterating through the parameters and (ii) the accuracy achieved by LDA reported by De Lucia et al. [39].

### 3.3.4 LDA-GA Settings and Implementation

The LDA-GA has been implemented in `R` [174] using the `topicmodels` and `GA` libraries. The former library provides a set of routines for computing the fast collapsed Gibbs sampling generative model for LDA, while the latter is a collection of general purpose functions that provide a flexible set of tools for applying a wide range of GA methods. For GA, we used the following settings: a crossover probability of 0.6, a mutation probability of 0.01, a population of 100 individuals, and an elitism of two individuals. As a stopping criterion for the GA, we terminated the evolution if the best results achieved did not improve for ten generations; otherwise we stopped after 100 generations. All the settings have been calibrated using a trial-and-error procedure, and some of them (i.e., elitism size, crossover and mutation probabilities) were the values commonly used in the community. To account for GA's randomness, for each experiment we performed 30 GA runs, and then we took the configuration achieving the median final value of the fitness function (i.e., of the Silhouette coefficient).

## 3.4 Empirical Study Results

This section discusses the results of our experiments conducted in order to answer the research questions stated in Section 3.3. We report the results for each LDA application scenario.

### 3.4.1 Scenario I: Traceability Link Recovery

As for **RQ$_1$**, Figure 3-2(a) shows boxplots summarizing the average precision values obtained using the 1,000 and 2,000 different LDA configurations (described in Section 3.3) on EasyClinic and eTour, respectively. We used these boxplots to highlight the variability of the average precision values across different configurations. As shown, the variability of LDA's performance is relatively high: the average precision ranges between 11% and 52% on

Table 3-2 The results of the Wilcoxon test for Traceability Link Recovery

| Comparison | EasyClinic | eTour |
|---|---|---|
| LDA-GA < Combinatorial | 1 | 1 |
| LDA-GA < Oliveto et al. [132] | **< 0.01** | **< 0.01** |
| Combinatorial < Oliveto et al. [132] | **< 0.01** | **< 0.01** |
| Combinatorial < LDA-GA | 1 | **< 0.01** |

EasyClinic and between 7% and 43% on eTour. For EasyClinic, more than 75% of the different LDA configurations obtained an average precision lower than 45% (see first three quartiles in Figure 3-2(a)). Moreover, only a small percentage of the configurations executed in the combinatorial search (about 3.6%) obtained an average precision greater than 50%. In the end, only one of them achieved the highest value, 52%. Similarly for eTour, the configurations placed in the first three quartiles (about 75% of the set) obtained an average precision lower than 40%, while less than 1% of the total amount of executed configurations in the combinatorial search (2,000 configurations) achieved an average precision greater than the 40%. Only one configuration achieved the highest average precision (43%).

In summary, for **RQ$_1$**we can assert that for traceability recovery, LDA shows high variability. Thus, LDA's efficiency for establishing links between software artifacts depends on the particular configuration $P = [k, n, \alpha, \beta]$ used to derive latent topics. Indeed, "bad" configurations can produce poor results while "optimal" configurations (which represent a small portion of all possible LDA configurations) can lead to very good results.

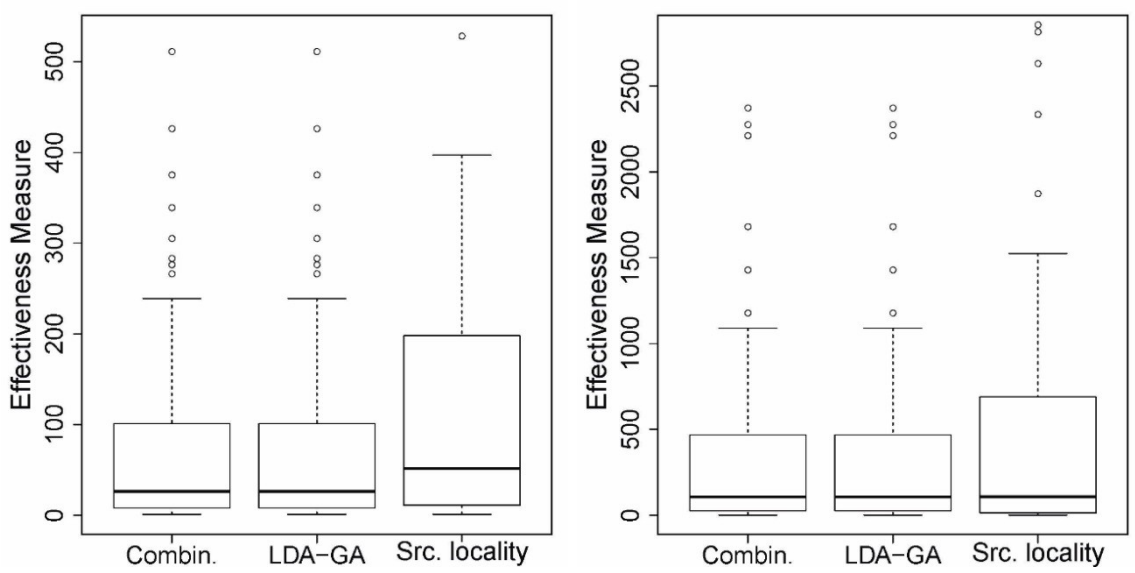
Regarding **RQ$_2$**, Figure 3-3 reports the precision/recall graphs obtained by LDA using (i) the best configuration across 1,000 and 2,000 different configurations executed in the combinatorial search; (ii) the configuration identified by LDA-GA; and (iii) an "ad-hoc" configuration used in a previous study where LDA was used on the same repositories [132]. For both EasyClinic and eTour, LDA-GA was able to obtain a recovery accuracy close to the accuracy achieved by the optimal configuration across 1,000 and 2,000 different configurations

executed in the combinatorial search. In particular, for EasyClinic LDA-GA returned exactly the configuration identified by the combinatorial search (i.e., the two curves are perfectly overlapped) while on eTour the two curves are comparable. Moreover, the average precision achieved by the configuration provided by LDA-GA is about 41%, which is comparable with the average precision achieved with the optimal configuration, which is about 43% (only a small difference of 2%). Among 2,000 different configurations tried for the combinatorial search, only five configurations obtained an average precision comparable or greater than the one achieved by LDA-GA, i.e., the configurations obtained by LDA-GA belong to the 99% percentile for the distribution reported in Figure 3-2(a). Finally, comparing the performance achieved by LDA-GA with the performance reached by other LDA configurations used in previous work [132], we can observe that the improvement is very substantial for both software systems.

Table 3-2 reports the results of the Wilcoxon test (i.e., the adjusted p-values) for all combinations of the techniques (statistically significant results are highlighted in bold face). As observed, there is no statistically significant difference between the performance obtained by LDA-GA and the combinatorial search for EasyClinic. However, for eTour the combinatorial search performs significantly better than LDA-GA. However, considering the precision/recall graph reported in Figure 3-3, we can observe that the difference is relatively small.

### 3.4.2 Scenario II: Feature Location

Figure 3-2(b) shows the boxplots summarizing the variability of the average effectiveness measure (EM) values obtained using 1,000 different LDA configurations, as explained in Section 3.3. As in the previous task, the feature location results show high variability in their EM, which ranges between 472 and 1,416 for ArgoUML and between 145




(a) jEdit (b) ArgoUML

Figure 3-4 Box plots of the effectiveness measure for the feature location task for systems: (a) jEdit and (b) ArgoUML

and 600 for jEdit. For ArgoUML, we observed that more than 90% of different configurations produced an average EM ranging between 600 and 1,200, while only a small percentage (about 3%) produced an optimal average EM lower than 600. Within this small number of optimal configurations only one configuration obtains the lowest (i.e., the best) EM of 472. Similarly, for jEdit, 95% of different configurations produced an average EM that ranges between 200 and 1,600, while only one achieved the smallest average EM of 145. These results for **$RQ_1$** suggest that without a proper calibration, the performance of LDA risks of being unsatisfactory.

For **$RQ_2$**, Figure 3-4(b) shows boxplots for ArgoUML of the EM values achieved by three different configurations: (i) the best configuration obtained by a combinatorial search across 1,000 different LDA configurations (combinatorial search); (ii) the configuration obtained using LDA-GA; and (iii) the best configuration obtained using the source locality heuristic [70]. First, we can note that the configuration obtained via LDA-GA is exactly the same as the one obtained from the combinatorial search, thus LDA-GA was able to find the best configuration (i.e., with the lowest average EM). Comparing the performance of LDA-GA

Table 3-3 Results of the Wilcoxon test for Feature Location

| Comparison | jEdit | ArgoUML |
|---|---|---|
| LDA-GA < Combinatorial | 0.09 | 1 |
| LDA-GA < Source Locality Heuristic [70] | **0.02** | **0.02** |
| Combinatorial < Source Locality Heuristic [70] | **0.02** | **0.02** |

with the source locality heuristic, we can observe that for the first two quartiles, there is no clear difference (the median values are 107 and 108 for LDA-GA and source locality heuristic respectively). Considering the third and fourth quartiles, the difference becomes substantial: the third quartile is 467 for LDA-GA and 689 for the previous heuristic, while for the fourth quartiles we obtained 4,603 for LDA-GA and 7,697 for source locality heuristic. Overall, LDA-GA reached an average EM equal to 473, as opposed to EM equal to 707 obtained using the source locality heuristic. The same trend is observed for the jEdit system illustrated in Figure 3-4(a).

Table 3-3 reports the results of the Wilcoxon test (i.e., the adjusted p-values) for all combinations of the techniques (statistically significant results are shown in bold face). As we can see, there is no statistical difference between the performance obtained by LDA-GA and the combinatorial search. Based on the results of the statistical tests, we can assert that LDA-GA is able to find the optimal or the near-optimal configurations. Moreover, LDA-GA significantly outperforms the previously published source locality heuristic (p-value< 0.02).

### 3.4.3 Scenario III: Software Artifact Labeling

For **$RQ_1$**, Figure 3-2(c) shows boxplots for the average percentage overlap (AO) values obtained using 500 different LDA configurations, as explained in Section 3.3. Even if in this case the corpus of documents (the total number of classes and the vocabulary size) is really small, as compared to the size of the repository considered for the other tasks, LDA also shows a high variability of performances, ranging between 18% and 66% on JHotDraw, and between 13% and 77% on eXVantage. For JHotDraw, it can be noted that more than 72% of the different

Table 3-4 Average Overlap between Automatic and Manual Labeling for the two systems: eXVantage (top) and JHotDraw (bottom)

eXVantage

| Descriptive Statistics | LDA | | De Lucia et al.[39] | | |
|---|---|---|---|---|---|
| | LDA-GA | Combinatorial | $n = M$ | $n = M/2$ | $n = 2$ |
| **Max** | **100%** | **100%** | 100% | 100% | 100% |
| **3rd Quartile** | **95%** | **95%** | 71% | 70% | 69% |
| **Median** | 67% | **70%** | 59% | 60% | 54% |
| **2nd Quartile** | 60% | **67%** | 34% | 50% | 41% |
| **Min** | **50%** | **50%** | 0% | 0% | 40% |
| **Mean** | 74% | 77% | 52% | 56% | 60% |
| **St. Deviation** | 19% | 17% | 31% | 34% | 23% |

JHotDraw

| Descriptive Statistics | LDA | | De Lucia et al.[39] | | |
|---|---|---|---|---|---|
| | LDA-GA | Combinatorial | $n = M$ | $n = M/2$ | $n = 2$ |
| **Max** | **100%** | **100%** | 100% | 100% | 100% |
| **3rd Quartile** | 81% | **82%** | 73% | 70% | 66% |
| **Median** | 71% | **75%** | 65% | 61% | 56% |
| **2nd Quartile** | 47% | **50%** | 46% | 45% | 41% |
| **Min** | 14% | 14% | 0% | 38% | **29%** |
| **Mean** | 65% | 66% | 59% | 60% | 59% |
| **St. Deviation** | 28% | 26% | 28% | 20% | 24% |

configurations obtained an AO value ranging between 25% and 55%, while only a small percentage (about 1%) obtains an optimal AO greater than 60%. Within this small number of optimal configurations, only one achieves the highest AO of 64%. Similarly, for eXVantage the majority (about 79%) of the different configurations obtained an AO ranging between 10% and 70%, while only one configuration achieved the highest AO of 77%.

For **$RQ_2$**, Table 3-4 reports the statistics of the overlap between the user-based labeling and the automatic labeling obtained using (i) LDA-GA; (ii) the best configuration achieved using the combinatorial search, i.e., the configuration which has the higher percentage overlap among 500 different configurations; and (iii) the LDA configuration used in the previous work [39] for the same task. For both systems, LDAGA obtains a percentage overlap with the user labeling that is close to the combinatorial search, with a difference from the best LDA configuration (obtained by the combinatorial search) of about 3% for eXVantage and 1% for

JHotDraw. For eXVantage, among the 500 different LDA configurations computed in the combinatorial search, only 12 configurations have an average overlap greater or equal to 74.33%. We can also observe that there are only small differences for the median and second quartile between LDA-GA and the global optimum, while for the other quartiles there is no difference. Similarly, among 500 different configurations evaluated for JHotDraw, only one configuration is comparable with LDAGA. By comparing the quartile values obtained for JHotDraw, we can note that the difference between LDA-GA and the combinatorial search optimum is about 2%-3% on average. Finally, we can observe how the performances of LDA configured using LDA-GA are significantly better than those reported in the previous work [39] (where $\alpha$ and $\beta$ were set to default of $50/k$ and 0.1 respectively). For eXVantage we obtain an improvement in terms of mean overlap of about 14-20%, while for JHotDraw we get an improvement of about 5-6%.

## 3.5 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. For tasks such as traceability link recovery and feature location, we used well-established metrics (i.e., precision, recall and effectiveness) and oracles used in previous studies [17, 53, 65, 66], thereby ensuring that the error-proneness is limited. For the labeling task, we compared LDA-based labeling with a user-generated labeling, using, again, the dataset previously verified and published [39].

Threats to *internal validity* can be related to co-factors that could have influenced our results. We limited the influence of GA randomness by performing 30 GA runs and considering the configuration achieving the median performance. We also observed that the configuration that we obtained did not substantially vary across GA runs.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Wherever appropriate, we used non-parametric statistical tests (the Wilcoxon test rank sum test in particular) to support our claims.

Threats to *external validity* concern the generalization of our results. Firstly, it is highly desirable to replicate the studies carried out on three scenarios on other datasets. Secondly, although the proposed approach can be applied in principle to other LDA-based solutions to support SE tasks, specific studies are needed to evaluate their feasibility and performances.

## 3.6 Discussion

In this chapter we proposed LDA-GA, an approach based on Genetic Algorithms that determines the near-optimal configuration for LDA in the context of three important software engineering tasks, namely (1) traceability link recovery, (2) feature location, and (3) software artifact labeling. We also conducted several experiments to study the performance of LDA configurations based on LDA-GA with those previously reported in the literature (i.e., existing heuristics for calibrating LDA) and a combinatorial search. The results obtained indicate that (i) applying LDA to software engineering tasks requires a careful calibration due to its high sensitivity to different parameter settings, that (ii) LDA-GA is able to identify LDA configurations that lead to higher accuracy as compared to alternative heuristics, and that (iii) its results are comparable to the best results obtained from the combinatorial search.

Overall, our empirical results warn the researchers about the dangers of ad-hoc calibration of LDA on software corpora, as was predominantly done in the SE research community, or using the same settings and parameters applicable only to natural language texts. Without a sound calibration mechanism for LDA on software data, which might require using approaches such as the one proposed in this chapter, the potential of such a rigorous statistical method as LDA can be seriously undermined, as shown in our empirical study.

# 4 Configuring and Assembling IR Techniques: IR-GA

Prior research in software engineering (SE) highlighted the usefulness of conceptual (or textual) unstructured information to capture the knowledge and design decisions of software developers. Identifiers and comments encoded in class names, method names, or attributes in source code or other artifacts contain information often indispensable for program understanding [6, 25, 43, 77, 98, 154] and account for approximately half of the source code in software systems [43]. This conceptual information plays a paramount role as a data source, which is used by (semi-) automatic techniques to support software maintenance and evolution tasks.

In the last decade a lot of effort in the SE community has been devoted to the problem of extracting, representing, and analyzing conceptual information in software artifacts. Specifically, Information Retrieval methods were proposed and used to support practical tasks. Early approaches aimed at constructing software libraries [109] and supporting reuse tasks [125, 187], while more recent work focused on addressing software maintenance and development tasks including feature location (e.g., [16, 141, 142, 143, 145, 146, 147, 149, 150, 151, 162, 186]), traceability link recovery (e.g., [5, 41, 94, 99, 112, 123]), change impact analysis (e.g., [54, 64, 87, 88]), source code search [72, 119, 120, 121, 122], detecting similar applications [73, 118], code measurement [13, 66, 113, 114, 144], identification of duplicate bug reports (e.g., [159, 182]) among many other applications [20, 51, 140].

All these IR-based techniques that support SE tasks, such as Latent Semantic Indexing [42] or Latent Dirichlet Allocation [22], require configuring different components and their respective parameters, such as type of pre-processors (e.g., splitting compound or expanding abbreviated identifiers; removing stop words and/or comments), stemmers (e.g., Porter [139]

or Snowball [170]), indexing schemata (e.g., term frequency - inverse document frequency), similarity computation mechanisms (e.g., cosine, dot product, entropy-based), etc.. Nevertheless, despite this overwhelming popularity of IR methods in SE research, most of the proposed approaches are based on ad-hoc methods and guidelines to choosing, applying, and configuring IR techniques. Moreover, *recent studies demonstrate that effectiveness of these IR-based approaches not only depends on the design choices behind the IR techniques and their internals, but also on the type of software artifacts used in the specific SE tasks and more in general on the project datasets*. [132, 134].

Most of existing approaches to SE tasks using IR methods rely on ad-hoc methods to configure these solutions, components, and their configurations, thus resulting oftentimes in suboptimal performance of such promising analysis methods to deal with unstructured software data. Moreover, a SE literature analysis (reported in the Related Work on choosing the different phases on an IR process – Section 4.6) has shown that a large number of papers do not even provide the details on how certain IR-based techniques have been instantiated; whereas the remaining set of papers use ad-hoc configurations, component settings, thus, significantly underachieving potential of IR methods to solve SE tasks. All in all, our conjecture is that existing methods for designing IR-based solutions for SE tasks is currently based on art, rather than science. This also makes the practical use of IR-based processes quite difficult and undermines the technology transfer to software industry.

In this chapter we propose a novel approach to solve the problem of assembling IR-based solutions for a given SE task and accompanying dataset. Our underlying assumption, which is supported by a large body of empirical research in the field, is that it is not possible to build a set of guidelines for assembling IR-based solutions for a given set of tasks as some of these solutions are likely to underperform on previously unseen datasets.

Our solution, named IR-GA, aims at enabling automatic search and assembly of parameterized IR-based solutions for given SE tasks that take into account not only task specific components and data sources (i.e., different parts of software artifacts related to solving a particular SE task), but also internal properties of the IR model built from the underlying dataset using a large number of possible components and configurations. We use Genetic Algorithms to effectively explore the search space of possible combinations of instances of IR process components (e.g., pre-processors, stemmers, indexing schemata, similarity computation mechanisms) to select the candidates with the best expected performance for a given dataset used for a SE task. Noticeably, during the GA evolution, the quality of a solution (represented as a GA individual) is evaluated based on the quality of the clustering of the indexed software artifacts. *For this reason, our approach is **unsupervised** and **task-independent**, whereas the instantiated process is **dataset-specific**. Thus, it can be used to select and generate on demand an adequate IR-based solution given a dataset provided as input. Moreover, IR-GA could potentially support any IR-based software engineering task (e.g., traceability link recovery, feature location, impact analysis, detection of duplicate bug reports, developer recommendations, source code search, bug triaging, clone detection, etc.).*

We empirically show that using IR-GA it is possible to automatically assemble a near-optimal configuration of an IR-based solution for datasets related to three kinds of software engineering tasks, namely (i) traceability link recovery, (ii) feature location, and (iii) duplicate bug report identification. The evaluation shows that IR processes instantiated by IR-GA outperform previously published results related to the same tasks and the same datasets, and that the performances of IR-GA do not significantly differ from an "ideal" upper bound, obtained by means of a combinatorial search, and with the availability of an oracle (which is not required by our approach).

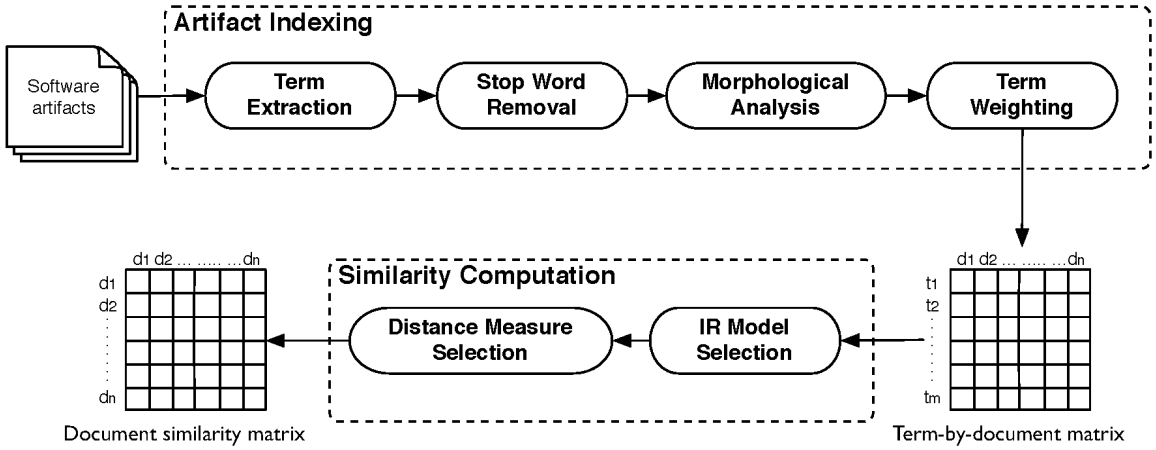



Figure 4-1 Outline of a generic IR Process to solve SE problems

## 4.1 Background

This section provides some backgrounds on a generic IR process for solving SE problems, and how such a process can be instantiated for solving three problems that we believe are relevant IR applications to SE: traceability link recovery, feature location, and identification of duplicate bug reports.

### 4.1.1 A generic IR process

Let us consider a generic IR process, as the one shown in Figure 4-1. The next paragraphs describe in details each of these steps.

**Step 1. Term extraction** This step consists of removing elements (e.g., special characters) that are not relevant to the IR process, and extracting portions of software artifacts that are considered relevant for the task at hand. For example, when performing requirement-to-source traceability recovery or feature location, one may or may not decide to consider source code comments [5], may or may not decide to split compound identifiers [45], or may decide to consider only certain parts-of-speech (e.g., nouns [24]). Similarly, when comparing bug reports for duplicate detection, contributor comments' (other than the initial description), stack traces, and source code fragments may (or may not) be considered.

**Step 2. Stop word removal** This step aims at removing common terms that often do not contribute to discern one document from another. This can be done using list-based stop word removals, e.g., by removing English (or other languages) stop words (e.g., articles, prepositions, common use verbs), programming language keywords, or recurring domain-specific terms.

**Step 3. Morphological analysis** This step is often performed to bring back words to the same root (e.g., by removing plurals to nouns, or verb conjugations). The simplest way to do morphological analysis is by using a stemmer (e.g., Porter [138], Snowball [170]).

**Step 4. Term weighting** The information extracted in the previous phase is stored in a $m \times n$ matrix, called *term-by-document matrix* (*TDM*), where $m$ is the number of terms occurring in all the artifacts, and $n$ is the number of artifacts (i.e., documents) in the repository. A generic entry $w_{ij}$ of this matrix denotes a measure of the weight (i.e., relevance) of the $i^{th}$ term in the $j^{th}$ document [10]. Different measures of relevance can be used: the simplest one is the *Boolean* weighting, which just indicates whether a term appears in a document or not; other measures are the term frequency, which accounts for the *term frequency* (*tf*) in a document, or the *tf-idf* (*term frequency-inverse document frequency*), which gives more importance to words having high frequency in a document (high *tf*) and appearing in a small number of documents, thus having a high discriminating power (high *idf*). In general, one can use a combination of a local weight of the term in a specific document (e.g., *tf*) and a global weight of the term in the whole document collection (e.g., *idf*). A more sophisticated weighting schema is represented by *tf-entropy* [56], where the local weight is represented by the term frequency scaled by a logarithmic factor, while the entropy of the term within the document collection is used for the global weight.

Applying one of these term weighting schemas (e.g., *Boolean, tf, tf-idf, log, tf-entropy*) on the TDM results in a *weighted term-by-documents matrix (WTDM),* which has the same size as the TDM. The WTDM will be used as input for the algebraic model as illustrated in the next step.

**Step 5. Application of an algebraic model** After having built a WTDM, different algebraic models (e.g., Vector Space Model [160], Latent Semantic Indexing [42], Latent Dirichlet Allocation [22]) can be applied on this matrix in order to build a corresponding IR model.

The simplest approach would require analyzing the WTDM "as is" using the traditional Vector Space Model [160]. In VSM, artifacts (i.e., documents) are represented as vectors of terms (i.e., columns of the TDM) that occur within artifacts in a repository [10]. Textual similarities between documents can be computed using distance metrics between vectors (see next step). It is important to know that VSM does not take into account relations between terms of the artifacts vocabulary. For instance, having "automobile" in one artifact and "car" in another artifact does not contribute to the similarity measure between these two documents.

Alternatively, one can use Latent Semantic Indexing [42], which is an extension of the VSM. It was developed to overcome the synonymy and polysemy problems, which occur with VSM model. LSI explicitly takes into account the dependencies between terms and between artifacts, in addition to the associations between terms and artifacts. For example, both "car" and "automobile" are likely to co-occur in different artifacts with related terms, such as "motor" and "wheel". To exploit information about co-occurrences of terms, LSI applies Singular Value Decomposition (SVD) [42] to project the original WTDM into a reduced space of concepts, and thus limit the noise that the terms may cause. A crucial factor that determines

the performances of LSI is the choice of the number of "concepts" known as dimensionality reduction factor (i.e., the $k$ parameter).

More advanced methods, such as Latent Dirichlet Allocation [22] or Jensen–Shannon [32], treat documents as probability distributions of terms. LDA requires a calibration of different parameters, such as number of topics and the Dirichlet distribution parameters $\alpha$ and $\beta$. For a detailed description of LDA and its parameters please refer to Chapter 3.1.1.

**Step 6. Use of a distance (or similarity) measure** The last step of the IR process aims at comparing documents (e.g., requirements and source code in traceability recovery, queries and source code in feature location, bug report pairs in duplicate bug report detection, etc.). This can be done using different similarity measures. For example, one can use the cosine similarity, Euclidian distance, the Jaccard similarity, or the Dice (symmetric or asymmetric similarity) coefficient.

### 4.1.2 Traceability Link Recovery

For traceability link recovery tasks, the typical artifacts used for generating the corpus consist of documentation artifacts, such as requirements, use cases or class diagrams and source code components, such as classes or methods. These artifacts depend on the traceability recovery task. For example, for recovering traceability links between use cases and source code classes the use cases are used as queries (or *source artifacts*) and the classes are used as *target artifacts*. All these artifacts are typically preprocessed by (i) removing special characters, (ii) splitting identifiers, (iii) removing stop words that are common in language as well as words that appear frequently in the templates of the source artifacts (e.g., "use case number", "actor", etc.) and (iv) stemming. The most used weighting schema for the TDM is the standard *tf-idf*, while VSM and LSI are among the most used IR techniques. The cosine similarity between all the source artifacts and all the target artifacts is used to rank the potential candidate links. The

Table 4-1 IR-GA chromosome representation (left) and values of the genes (i.e., steps of the IR process) for IR-GA

| Gene Type | Possible Values for Gene Type |
|---|---|
| Character Pruning | No Removing |
| | Remove Special Characters, but Keep Digits |
| | Remove Special Characters and Remove Digits |
| Identifier Splitting | No Splitting |
| | Camel Case Split |
| | Camel Case Split and Keep Original Compound Identifiers |
| Stop Word Removing | No Stop Word Removal |
| | Removal of Standard Stop Words |
| | Removal of Standard Stop Words; Remove $\leq 2$ characters words |
| | Removal of Standard Stop Words; Remove $\leq 3$ characters words |
| Morphologic Analysis (Stemming) | No Stemming |
| | Snowball Stemming |
| | Porter Stemming |
| Term Weighting | Boolean |
| | tf |
| | tf-idf |
| | log |
| | tf-entropy |
| IR Technique | VSM |
| | LSI |
| | LDA |
| LSI Settings | $k$, the dimensionality reduction factor, where $10 \leq k \leq rank(WTDM)$ |
| LDA Settings | $k$, the number of topics, where $10 \leq k \leq rank(WTDM)$ |
| | $n$, the number of Gibbs iterations, where $10 \leq n \leq 2000$ |
| | $\alpha$, which influences the topic distributions per document; $0 \leq \alpha \leq 5$ |
| | $\beta$, which influences the term's distribution per topic; $0 \leq \beta \leq 5$ |
| Metric To Compute Documents Similarities | Cosine |
| | Euclidian |
| | Jaccard |
| | Dice |

list of candidate links is presented to the developer for investigation to decide if they are correct or not.

### 4.1.3 Feature Location

The process of applying IR techniques to support feature location is similar to traceability link recovery. The artifacts used for generating the corpus are typically short textual descriptions (e.g., queries) of the bug or the change request (i.e., the *source artifacts*) and program elements, such as classes or methods (i.e., the *target artifacts*). The queries can be

formulated manually by the developer, or they can be extracted from issue tracking systems, such as Bugzilla. For the latter case, the query can consist of the title (e.g., short summary), or the combination of the title and the description of the issue. For the target artifacts, the typical information associated with a method consists of comments, type, name, signature and body and the information associated with a class consists of all the comments, methods and fields. The corpus is preprocessed using the standard steps (e.g., removing special characters, splitting compound identifiers, removing stop words and stemming), and weighting schemata (e.g., tf-idf). Also for feature location, LSI is one of the most used techniques. The similarity measure is also in this case the cosine similarity. Using such a similarity measure, the list of target artifacts is ranked descending and presented to the developer, which manually investigates these methods and decides if they are relevant or not to the query.

### 4.1.4 Identification of Duplicate Bug Reports

For the task of detecting duplicate bug reports, the primary source of information for constructing the corpus consists of the information extracted from issue tracking systems. Each document of the corpus (i.e., each bug) typically consists of the title (e.g., short description of the issue), the description, and in some cases by the project name, component name, severity, priority, etc. In these documents, different weights could be assigned to the previously enumerated elements (e.g., title could be weighted more than description). The *source artifacts* are new, unassigned bugs for which the developer is trying to find similar bugs, and the *target artifacts* are existing bugs, which were assigned to developers or resolved. Similarly to the other tasks, the corpus is preprocessed using the standard steps (e.g., removing any sentence punctuation marks, splitting identifiers, removing stop words and stemming). The cosine similarity of an IR technique (e.g., VSM) between the source (i.e., new) bugs and the target

(i.e., existing) bugs is used to rank the list of bugs presented to the developer for the manual inspection.

## 4.2 The Proposed Approach: IR-GA

Genetic Algorithms [82] are a stochastic search technique inspired by the mechanism of a natural selection and natural evolution. A GA search starts with a random population of solutions, where each individual (i.e., *chromosome*) of a population represents a solution of the optimization problem. The population is evolved toward better solutions through subsequent generations and, during each generation, the individuals are evaluated based on the *fitness* function that has to be optimized. For creating the next generation, new individuals (i.e., *offsprings*) are generated by (i) applying a *selection operator*, which is based on the fitness function of the individuals to be reproduced, (ii) recombining, with a given probability, two individuals from the current generation using the *crossover operator*, and (ii) modifying, with a given probability, individuals using the *mutation operator*.

In this dissertation we propose to use a GA to automatically assemble and instantiate parameterized IR-based solutions for SE tasks. Specifically, we use GA to (i) instantiate the IR process represented in the chromosome of an individual, (ii) executing it (i.e., processing the documents using that IR process), and (iii) computing the GA fitness function by clustering the processed documents and evaluating the clustering quality.

### 4.2.1 Use GA to instantiate IR processes

The first choice in the design of a GA for an optimization problem is the representation of candidate solutions that must be encoded in a suitable form, known as *chromosome representation*. As explained in Section 4.1, an IR process (see Figure 4-1) consists of a sequence of given steps or components, thus it naturally lends itself to be represented as a chromosome. In IR-GA, the chromosome (see Table 4-1) is a vector, where each cell (*gene*)

represents a phase of the IR process, and can assume as a possible value any of the techniques (or approaches) available for that phase. For example, for the morphological analysis we could have "no stemming", "Porter" stemmer [139], or "Snowball" stemmer [170]. Note that, since some steps require multiple decisions, they are represented as two genes. In particular, the term extraction has two genes, one related to what kind of characters to prune, another related to how splitting compound identifiers. In principle, further, more complex configurations can be foreseen. The right-side of the chromosome encodes the parameter settings for the used IR methods. While in principle an approach like IR-GA could also search for the IR method (e.g., LSI, LDA or VSM) achieving the best performances (see Section 4.2.2), the fitness function we defined requires at the moment the use of an IR method that clusters documents. For example, LSI [42] and LDA [22] are two such techniques that represent documents using a fixed number of concepts or topics respectively. For this reason, we only considered LSI, which already proves to be very successful for investigated software engineering tasks [65, 143]. In terms of calibration, LSI requires to set the number of concepts ($k$) to which the term-by-document space will be reduced.

IR-GA is based on a simple GA with elitism of two individuals (i.e., the two best individuals are kept alive across generations). The GA's initial population is randomly generated (i.e., by randomly choosing the value of each gene of each individual). The *selection* of the individuals to be reproduced is performed using the *Roulette Wheel* selection operator, which elects individuals to reproduce pseudo-randomly, giving higher chances to individuals with higher fitness. The *crossover* operator is the *single-point* crossover, which, given two individuals (parents) $p_1$ and $p_2$, randomly selects a position in the chromosome, and then creates two new individuals (the offspring) $o_1$ composed of the left-side of $p_1$, and the right-side of $p_2$, and $o_2$ composed of the left-side of $p_2$ and the right-side of $p_1$. The mutation

operator is the uniform mutation, which randomly changes one of the genes with one of the admissible values. The GA terminates after a fixed number of generations or when the fitness function cannot be improved further (i.e., GA converged to a local or global maximum/minimum).

### 4.2.2 Measuring the quality of IR processes

Another important step in the design of a GA is the definition of the fitness function. In IR-GA we need to define a fitness function able to estimate the performances of an IR process. Thus, the fitness function evaluation is ***unsupervised*** *(i.e., it does not require a labeled training set or an oracle), making it* ***task-independent***.

When applying IR methods such as LSI or LDA to extract textual information from software artifacts, such techniques implicitly cluster the software documents on the basis of their textual similarities. Such different clusterings can be obtained by using various numbers of latent concepts/topics used for modeling the concept/topic space, independently from the used pre-processing techniques. We conjecture that there is a strong relationship between the performances obtained by an IR process on software corpora and the quality of produced clusters. Indeed, if the quality of the clusters produced by an IR-process is poor, this means that the IR process was not able to correctly extract the most important concepts from the software corpus and the documents (i.e., the documents which are more similar to each other, are assigned to different clusters). Similarly to what has been done in previous work when calibrating LDA [134] (see Chapter 3), we use the Silhouette coefficient [95] to measure the quality of clusters, since it provides only one scalar value combining *cohesion* and *separation* of clusters. In particular, the Silhouette coefficient is computed for each document using the concept of centroids of clusters. Let $C$ be a cluster; its centroid $Centroid(C)$ is equal to the mean vector of all documents belonging to $C$:

$$Centroid(C) = \sum_{d_i \in C} \frac{d_i}{|C|}$$

Starting from the definition of centroids, the Silhouette coefficient is computed for each document $d_i$ as:

$$s(d_i) = \frac{b(d_i) - a(d_i)}{\max\big(a(d_i), b(d_i)\big)}$$

where $a(d_i)$ is the separation (measured as the maximum distance from $d_i$ to the centroid of its cluster) and $b(d_i)$ is the cohesion (represented as the minimum distance from $d_i$ to the centroids of the clusters not containing $d_i$). The value of the Silhouette coefficient ranges between $[-1 \dots 1]$.

A good cluster has a positive Silhouette coefficient because it corresponds to the case in which $a(d_i) > b(d_i)$, i.e., the maximum distance to other documents in the cluster is greater than the minimum distance to other documents in other clusters. We used the cosine of the angle between vectors for measuring the distance between documents, since in LSI the documents are represented as vectors in the concepts space. In the end, the overall measure of the quality of clustering $C = \{C_1, \dots, C_k\}$, that is our fitness function, is computed by the mean Silhouette coefficient of all the documents.

## 4.3 Empirical Evaluation Design

The *goal* of our study is to investigate whether IR-GA it is able to instantiate IR processes that are able to effectively solve SE tasks, while the *quality focus* is represented by the performances of the IR-based processes in terms of accuracy and completeness. The *perspective* is of researchers interested in developing an automatic approach to assemble IR processes for solving specific SE tasks. The *context* of the study consists of (i) *three SE tasks*, namely traceability links recovery, feature location, and identification of duplicate bug reports, and (ii) their corresponding *objects* (i.e., datasets on which the tasks are experimented).

Specifically, the study aims at addressing the following research questions (RQs) that have been addressed in the context of the three different SE tasks considered in our study.

**RQ$_1$**: *How do the processes instantiated by IR-GA compare with those previously used in literature for the same tasks?* This research question aims at justifying the need for an automatic approach that calibrates IR processes for SE tasks. Specifically, we analyzed to what extent the process instantiated by IR-GA for solving a specific task is able to provide better performances than a process with an *ad-hoc* setting. Our conjecture is that, with a proper setting, the performances could be sensibly improved because in many cases, the IR-based techniques have been severely under-utilized in the past.

**RQ$_2$**: *How do the processes instantiated by IR-GA compare with an "ideal" configuration?* We empirically identified the configuration that provided the best results as compared to a specific oracle. For instance, in the case of traceability recovery, we identified the configuration that provided the best performances in terms of correct and incorrect links recovered. Clearly, one can build such a configuration only with the availability of a labeled data set, by using a combinatorial search among different treatments and by evaluating each combination against the oracle in terms of precision and recall. We call such a configuration *ideal*, because it is not possible to build a priori (i.e., without the availability of a labeled training set) a configuration providing better performances than that. The "ideal" configuration will be identified by means of a combinatorial search on discretized values of the search space of the parameters. The performances achieved by the process instantiated by IR-GA are then compared with those achieved by the combinatorial search, to investigate how far off is the IR-GA configuration from the best possible performances that one can achieve.

**RQ$_3$**: *Does a higher Silhouette coefficient value correlate with better performance for the maintenance tasks?* Our underlying assumption is that an IR model with a higher Silhouette coefficient will return a better result for a maintenance task, and subsequently, an IR model

Table 4-2 Characteristics of the systems used in the three evaluation tasks: Traceability Link Recovery (top), Feature Location (middle) and Detecting Duplicate Bug Reports (bottom)

Task 1 – Traceability Link Recovery

| System | KLOC | Source Artifacts (#) | Target Artifacts (#) | Correct Links |
|---|---|---|---|---|
| EasyClinic | 20 | Use Case (30) | Code Class (47) | 93 |
| eTour | 45 | Use Case (58) | Code Class (174) | 366 |
| iTrust | 10 | Use Case (33) | Java Server Page (47) | 58 |

Task 2 – Feature Location

| System | KLOC | # Files | # Methods | # Features |
|---|---|---|---|---|
| jEdit v4.3 | 104 | 503 | 6,413 | 150 |
| JabRef v2.6 | 74 | 579 | 4,607 | 39 |

Task 3– Detecting Duplicate Bug Reports

| System | Period | # Bugs & Traces | Duplicate Pairs | Traces Type | Method Signature |
|---|---|---|---|---|---|
| Eclipse 3.0 | June 2004 | 225 | 44 | Marked | No |
| Eclipse 3.0 [182] | June 2004 | 220 | 44 | Full | Yes |

with a low Silhouette coefficient will yield poor results. In order to evaluate the relation between the Silhouette coefficient and the maintenance task performance, we evaluated 1,000 different IR models using random values for the preprocessing steps, IR technique and IR settings. The performance of these IR models were evaluated in the context of a maintenance task.

### 4.3.1 Task 1: Traceability Link Recovery

For this task, we used IR-GA to recover traceability links between high level artifacts (e.g., use cases) and source code classes. The experiment has been conducted on the software repositories of three projects, EasyClinic, eTour and iTrust. EasyClinic is a system used to manage a doctor's office, while eTour is an electronic touristic guide. The documentation, source code identifiers, and comments for both systems are written in Italian. Both EasyClinic and eTour have been developed by final year Master students at the University of Salerno (Italy). iTrust is a medical application used as a class project for Software Engineering courses

at the North Carolina State University[8]. All artifacts consisting of use cases and Java Server Pages are written in English. Table 4-2 (top) summarizes the characteristics of the considered software systems: the number and type of source and target artifacts, the source code size in Kilo Lines of Code (KLOC), and the number of correct links between the source and target artifacts. These correct links are derived from the traceability matrix built and validated by the original developers. We consider such a matrix as the oracle to evaluate the accuracy of the different traceability recovery processes.

To answer **$RQ_1$**, we compared the accuracy of recovering traceability links achieved by the IR process assembled by IR-GA with the accuracy achieved by a baseline (or reference) in which LSI was applied on the same systems using an "ad-hoc" corpus pre-processing and LSI configuration [37, 40]. We also compared the accuracy of recovering traceability links using different combinations of pre-processing steps ($charPrunning \times splitting \times stopWordsRemoval \times stemming \times termWeighting = 3 \times 3 \times 3 \times 3 \times 5 = 405$, see Table 4-1). In addition, we used considered three IR techniques, namely VSM, LSI and LDA with different configurations, while considering four metrics to compute the similarities between documents (see Table 4-1).

Specifically, for LSI we varied the number of concepts from 10 to maximum number of topics, which is 77 for EasyClinic, 176 for eTour and 80 for iTrust. For VSM we fixed the number of concepts to the maximum number of topics. For LDA we varied the number of concepts from 10 to maximum number of topics in increments of 10. In addition, for LDA we varied $\alpha$ and $\beta$ from 0 to 5 with 0.1 increments.

We also exercised all possible combinations of preprocessing steps with such values. Thus, the total number of trials performed on EasyClinic were $405 \times 4 \times (77 - 10) =$

[8] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=tracing

$108{,}540$ for LSI, $405 \times 4 \times 1 = 1{,}620$ for VSM and $405 \times 4 \times \left\lceil \frac{(77-10)}{10} \right\rceil \times 10 \times 10 = 1{,}134{,}000$, for a total number of $108{,}540 + 1{,}620 + 1{,}134{,}000 = 1{,}244{,}160$ trails. Similarly, for eTour and iTrust there were 3,024,540 and 1,249,020 trails respectively.

Using this **combinatorial** search, we are able to identify the configuration that provides the best recovery accuracy (as compared with our oracle) among a large sample of possible configurations aiming at estimating the "ideal" configuration of the IR-based traceability recovery process. We then compared the performances achieved with the best configuration (identified with this combinatorial search) with the performances achieved with the configuration identified by IR-GA in order to answer **RQ$_2$**.

For both **RQ$_1$** and **RQ$_2$**, the performances of the IR-GA approach, of the baseline approach, and of the combinatorial approach are evaluated and compared by using two well-known metrics in the IR filed, namely precision and recall [10]. The precision values achieved for different levels of recall (for each correct link) by the different IR processes are then pairwise-compared using the Wilcoxon rank sum test. Since this requires performing three tests for each system, we adjusted the p-values using Holm's correction procedure [83]. Finally, we use the average precision and f-measure metrics [10] for comparing the performances of the different IR processes. We used the average precision since it provides a single value that is proportional to the area under precision-recall curve achieved for each ranked list of candidate links, and the f-measure since it encapsulates the harmonic mean between precision and recall. This average precision is the mean of the precision over all the correct links. Hence, it combines both precision and recall into a single performance scalar value.

### 4.3.2 Task 2: Feature Location

For this task, we used IR-GA to locate bugs and features in a corpus consisting of source code methods. The experiment has been conducted on two Java software systems,

namely jEdit v4.3 [85], an open-source text editor for programmers, and JabRef v2.6 [84], a bibliography reference manager. Table 4-2 (middle) reports the system size (in KLOC), the number of source files and methods, and the number of bugs and features to be located. These systems have been used in previous studies on feature location [17, 53]. For more details about how we generated the datasets for these systems refer to our Appendix A.

Regarding **$RQ_1$**, for the same systems we compared the performances obtained by the IR process assembled by IR-GA with the performances achieved by LSI in a previously published study, where an "ad-hoc" IR technique was used with a default (or standard) configuration [105]. The latter is used as a baseline. To address **$RQ_2$**, we compare the process instantiated by IR-GA with an "ideal" process, which was determined by using a **combinatorial** search similar to the one from Task 1 (see Section 4.3.1). The obtained similarities are then evaluated using the *effectiveness measure (EM)* [143]. The EM estimates the number of methods a developer needs to analyze before finding the first relevant method. More specifically, the EM is computed as the lowest rank of a relevant method that was found in the list of methods that were sorted in a descending order based on their textual similarity to the description of the feature of interest. A high EM value suggests sub-optimal performance for the IR technique and indicates a greater effort from the part of the developer, due to the large number of false positive methods to be analyzed before finding a relevant one. The EM computed for different IR process on different feature descriptions were then pairwise-compared using the Wilcoxon rank sum test, and similarly to the evaluation from Task 1, the p-values were adjusted using Holm's correction procedure.

### 4.3.3 Task 3: Duplicate Bug Report Identification

For this task, we used IR-GA to identify duplicate bug reports from a set of existing reports. More specifically, we used IR-GA to compute the textual similarity between new bug

reports and existing bug reports using their description. The textual corpora is the one composed of (i) the title of the bug, and (ii) the double weighted title and the description of the bug. We used these corpora in order to compare our results with the results of the experiment introduced by Wang et al. [182], which we use as a baseline.

Based on the study by Wang et al. [182], in addition to the textual similarity, for each bug report in the analyzed corpus, we also generated an execution trace by following the steps to reproduce the bug. These steps to reproduce were available in the bug description. Using the information from the traces we build a bug-to-method matrix, where each bug represents a column, and each method represents a row. The matrix has binary values, where an entry $e_{i,j}$ represents whether or not the $i^{th}$ method in the corresponding row appears in the execution trace of the $j^{th}$ bug report.

The bug-to-method matrix can be used to identify bugs having similar execution traces and can complement the textual similarity information (identified with an IR process) in order to identify duplicate bug reports. The intuition behind this process is that bugs having similar execution traces are more likely to be duplicates. We then apply IR-GA on the bug-to-method matrix as well and we compute the similarity between each bug (in terms of execution trace) using the *Execution-information-based Similarity*. The final similarity between each pair of bug report is given by averaging the textual similarity and the similarity of the execution traces of the two bugs.

The design of our study is based on the study introduced by Wang et al. [182], but is different in several important aspects. First, the IR technique is potentially different: we used IR-GA which instantiates LSI, VSM and LDA, while the evaluation of Wang et al. used VSM. Second, the datasets used are different, including the type of execution traces and method signatures. For example, Wang et al. used 220 bug reports related to Eclipse 3.0 posted on June

2004 and 44 duplicate pairs of bug reports. Moreover, their execution traces corresponding to the bug reports were *full execution traces*[9] and they also contained method signatures. For our evaluation, we used 225 bugs for the same system (i.e., Eclipse 3.0) that were posted in the same period (i.e., June 2004) with 44 duplicate pairs, and *marked execution traces*[10] without method signatures. For collecting the data, even though we strictly followed the methodology described in their approach, we did not have the exact set of bug reports used in their evaluation. Moreover, since the process of collecting the traces was manual, it is likely that the content of our traces will be different than the content of their traces. In addition, since collecting data is manual, collecting two separate traces for the same bug will likely produce different results, even if they are collected by the same user. In addition, our JPDA [133] instrumentation did not record the method signatures for the executed methods. In summary, we followed the design by Wang et al. to generate the dataset that we can use for comparison with their approach for Task 3, however our dataset does not fully correspond to the one by Wang et al. (see Table 4-2 (bottom)).

For each duplicate pair of bugs, we compute the similarity between the oldest submitted bug (among those two) and the remaining 224 bug report in the corpus. We computed the accuracy of detecting all the pairs of bugs using the Recall Rate (*RR*) [159, 182]. To address **RQ$_1$**, we compare the *RR* of the configuration produced by IR-GA against the *RR* of a "baseline" configuration produced by using the preprocessing steps described by Wang et al., and by applying LSI with an "ad-hoc" number of concepts used in traceability link recovery (i.e., $k = 50\%$ of total number of documents [40]). For **RQ$_2$**, we compared the *RR* generated

[9] A full execution trace is an execution trace that records all executed methods from the start of the application until the application is closed. Full traces usually capture more information than marked traces.
[10] A marked execution trace is a trace where the user has control over the beginning and the end of the trace recording process. Usually the user starts the trace before exercising the feature of interest and stops the trace immediately after exercising the scenario associated with the feature of interest

by the configuration suggested by IR-GA against the *RR* of the best configuration produced by performing a **combinatorial** search on the preprocessing steps, IR techniques, and their parameter values (similar to the combinatorial search for Task 1, traceability link recovery and Task 2, feature location). Similarly to the other two tasks, the *RR* values at different cut points (i.e., suggested list sizes) are pairwise-compared using the Wilcoxon rank sum test, adjusting the corresponding p-values using Holm's correction procedure.

### 4.3.4 IR-GA Implementation and Settings

IR-GA has been implemented in `R` [174] using the `GA` library. Every time an individual needs to be evaluated, we process documents using features available in the `lsa` package which allows applying all the pre-processing steps, while for computing the SVD decomposition we used the a fast procedure provided by the `slam` package for large and sparse matrices. We used the `topicmodels` package for LDA, and the `GA` package for the genetic algorithm.

As for the GA settings, we use a crossover probability of 0.8, a uniform mutation with probability of $1/n$, where $n$ is the chromosome size. We set the population size equals to 50 individuals with elitism of two individuals. As stop condition for GA, we terminate the evolution if the best fitness function value does not improve for 10 consecutive generations or when reaching the maximum number of generations equals to 100 (which was never reached in our experiments).

We want to emphasize that we choose these parameter values, as these settings are commonly used in the genetic algorithm community. We acknowledge the fact that there is no "silver bullet" configuration for a search algorithm that would work well on all types of optimization problems, as stated by the no free lunch theorem [185]. In other words, the same set of parameters have the potential to provide different results (e.g., optimal vs. sub-optimal) for different search spaces.

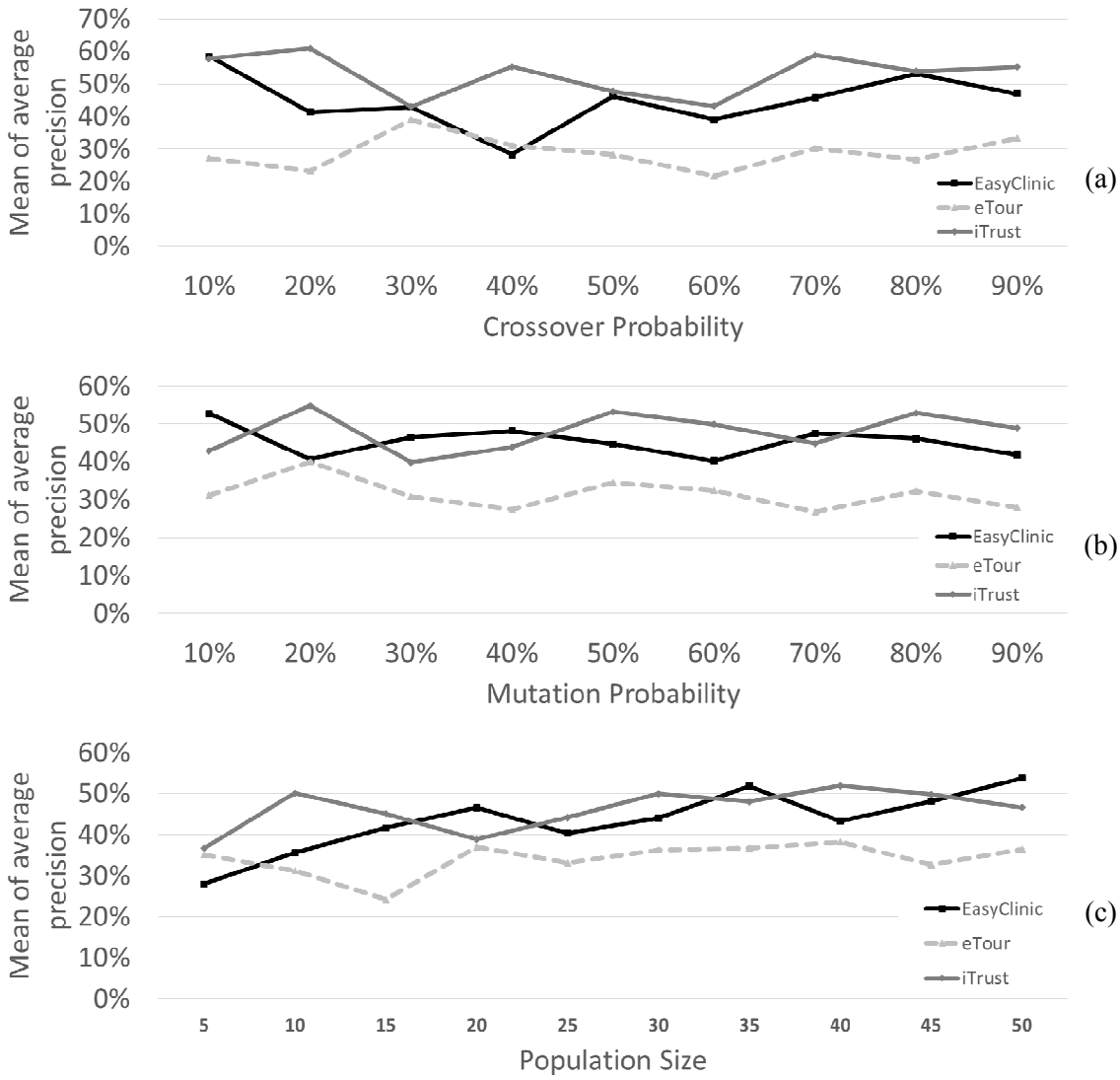



Figure 4-2 Mean of average precision (aggregated over thirty runs) that was obtained by running IR-GA on the EasyClinic, eTour and iTrust traceability link datasets while considering different values for the genetic algorithm parameters, namely: (a) crossover probability, (b) mutation probability and (c) population size.

When developing IR-GA our focus was to create a technique that identifies a good IR configuration for a dataset from a software maintenance task. Moreover, IR-GA was designed to be general enough to be applied on a wide range of maintenance tasks that have their particular characteristics, which in turn will generate a specific search space for the genetic algorithm. Thus, our IR-GA approach focuses on *generality* (i.e., it could be applied on a various number of maintenance tasks) rather than specificity (i.e., it was not designed to be

applied on a particular maintenance task, and it was not optimized for a specific maintenance task), and fine tuning the parameters of the genetic algorithm to obtain better results for one specific set of problems is beyond the scope of this dissertation.

Our IR-GA approach relies on a search heuristic provided by a genetic algorithm to identify a set of parameters (i.e., the configuration parameters for an IR solution) over a search space (i.e., the search space denoted by the available IR configurations that can be instantiated to create an IR based solution). Since genetic algorithms are typically instantiated with some specific parameter values, we investigated the impact of a subset of these parameters on the results produced by IR-GA. More specifically, we investigate the influence on the results for three important genetic algorithms parameters, namely the *crossover probability*, the *mutation probability* and the *population size*. Figure 4-2 illustrates the mean of average precision (that was aggregated over thirty runs) that was obtained by running IR-GA on the EasyClinic, eTour and iTrust traceability link datasets (see Table 4-2) while considering different values for the genetic algorithm parameters, namely: crossover probability (Figure 4-2 (a)), mutation probability (Figure 4-2 (b)) and population size (Figure 4-2 (c)).

The results indicate a variability in the results produced using different genetic algorithms values. For example, in Figure 4-2 (a), which analyzes the variability of the mutation probability, for the EasyClinic system, we observe a difference between the maximum and minimum value of the mean of average precision of about 30% (i.e., at 40% crossover probability the mean average precision is 28%, whereas at 10% crossover probability the mean average precision is at 58%). This difference of 30% is the most pronounced for the EasyClinic dataset, and for eTour and iTrust the difference between the maximum and minimum mean average precision is about 17% and 18%, respectively. When considering the mutation probability (see Figure 4-2 (b)), the differences between the maximum and minimum values of the mean average precision is approximately 12%, 13% and 15% for EasyClinic,

eTour and iTrust respectively. Finally, when varying the population size from 5 to 50 in increments of 5 for the genetic algorithm parameter, we obtain a difference for the mean of average precision values of approximately 26% for EasyClinic, 14% for eTour and 15% for iTrust.

It is clear from Figure 4-2 that the stochastic nature of the genetic algorithm, combined with variations on the genetic algorithm parameters will produce results that will vary. While this is an expected outcome, in order to address this intrinsic problem of genetic algorithms, for each task and for each dataset we perform 30 independent runs, storing the best configuration and the relative best fitness function value (i.e., the Silhouette coefficient) for each run. Among the obtained configurations, we consider the one that achieves the *median fitness function across the 30 independent runs* for the best individual in the last generation. Table 4-1 summarizes the possible values for each gene of the chromosome used in our experimentation. Clearly, the number of possible values can easily be extended (e.g. different stemming, different weighting schemas, etc.).

## 4.4 Empirical Evaluation Results

This section describes the results of our experiments conducted in order to answer the research questions stated in Section 4.3. The results are reported in different subsections for each SE task.

### 4.4.1 Task 1: Traceability Link Recovery

Figure 4-3 reports the precision/recall graphs obtained using (i) the *combinatorial* IR configuration; (ii) the IR configuration identified by IR-GA; and (iii) an "ad-hoc" configuration (i.e., *reference*) used in a previous study, where LSI was used on the same dataset and for the same traceability recovery task. For all three systems, namely EasyClinic, eTour and iTrust, IR-GA was able to obtain a precision and recall rate close to the one obtained by the

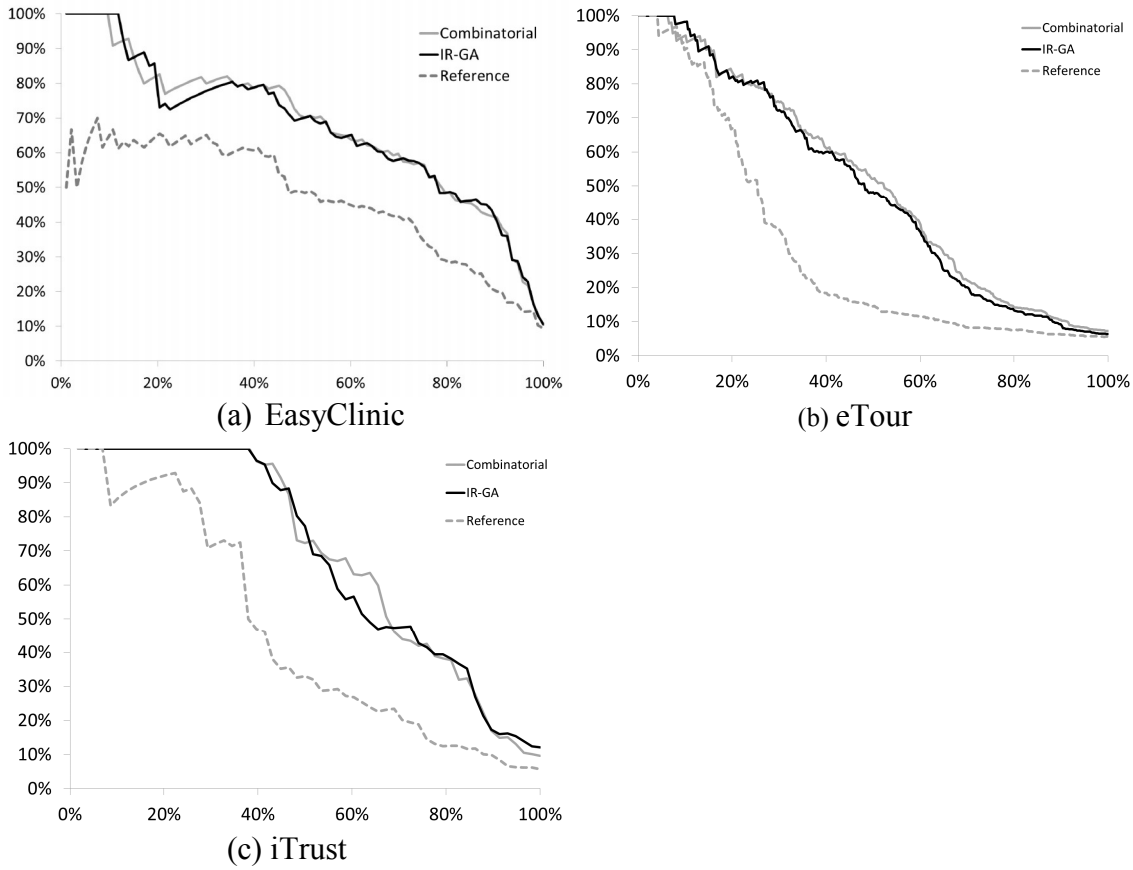



Figure 4-3 Traceability recovery: precision/recall graphs for (a) EasyClinic, (b) eTour and (c) iTrust

combinatorial configuration. It is important to emphasize that the configuration identified by IR-GA used no information about the oracle for computing the result, whereas the combinatorial search used 1,244,160 different configurations for EasyClinic, 3,024,540 for eTour and 1,249,020 for iTrust to identify the best configuration, and its performance was evaluated based on the oracle. In other words, among those >1 million configurations, the one that produced the best results based on the oracle was chosen for comparison.

Based on Figure 4-3, when comparing the performance achieved by IR-GA with those of the reference configuration, we can observe a significant improvement in all cases. These results are also confirmed by the average precision and f-measure obtained by the three different treatments (see Table 4-3). Indeed, the average precision and f-measure obtained by IR-GA is very close to the "ideal" (combinatorial) one. For example, for EasyClinic the average

Table 4-3 Comparison of the average precision values and f-measures (in parenthesis) for the traceability link recovery approaches: Combinatorial, IR-GA and Reference

| System | Combinatorial | IR-GA | Reference [37, 40] |
|---|---|---|---|
| EasyClinic | 67.47% (46.48%) | 66.92% (46.37%) | 46.78% (37.95%) |
| eTour | 50.48% (32.58%) | 49.02% (31.27%) | 30.93% (19.53%) |
| iTrust | 68.84% (42.70) | 68.13% (42.65%) | 45.47% (29.25%) |

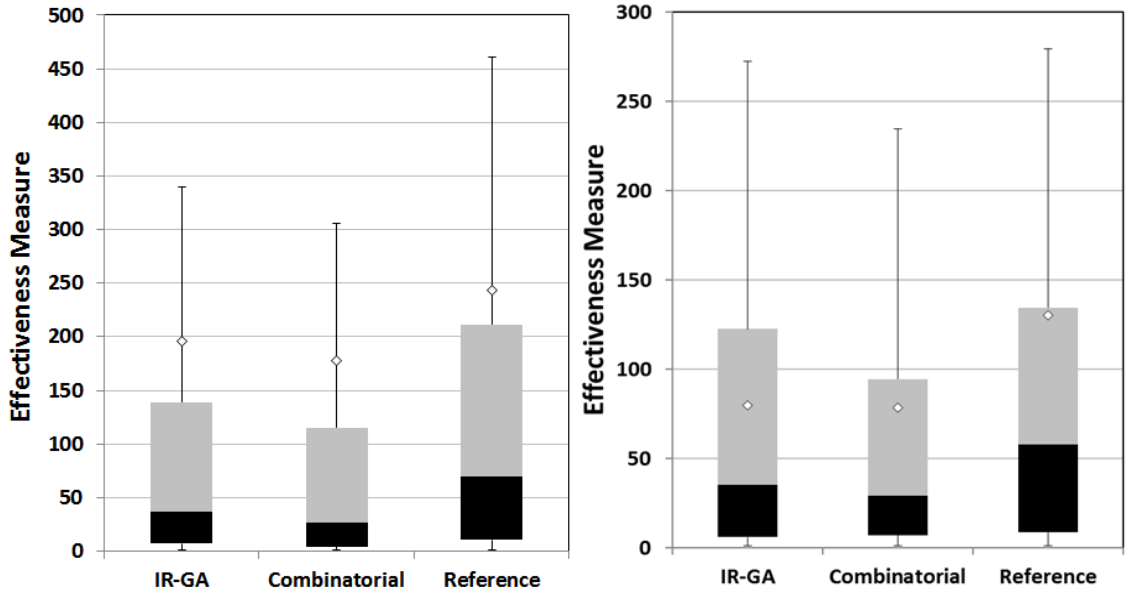
Table 4-4 Results of the Wilcoxon test for the Traceability Link Recovery task

| Comparison | EasyClinic | eTour | iTrust |
|---|---|---|---|
| IR-GA > Combinatorial | 0.99 | 1 | 0.99 |
| IR-GA > Reference [37, 40] | **<0.001** | **<0.001** | **<0.001** |
| Combinatorial > Reference [37, 40] | **<0.001** | **<0.001** | **<0.001** |

precision obtained with IR-GA is 66.92% (f-measure of 46.37%) , which is slightly lower than the average precision obtained by the combinatorial search, which is 67.47% (f-measure 46.48%). The same can be observed for eTour and iTrust. Moreover, the difference in terms of average precision and f-measure with respect to the combinatorial configuration is lower than 1%. However, the improvement obtained with respect to the reference configuration is of about 20% in terms of average precision and between 9%-13% in terms of f-measure. For example, for EasyClinic the average precision obtained by IR-GA is 66.92% (f-measure of 46.37%), whereas the average precision obtained by the reference configuration (i.e., the one that used an "ad-hoc" configuration of LSI) is 46.78% (f-measure of 37.95%). The findings presented in Figure 4-3 and Table 4-3 are also confirmed by our statistical analysis test (see Table 4-4), which illustrate that for all three systems, there is no statistical difference between the results produced by IR-GA and the combinatorial search, but there is a statistical difference between the results produced by IR-GA and the reference (baseline). In other words, IR-GA outperforms the baseline, and the difference is statistically significant.

### 4.4.2 Task 2: Feature location

Figure 4-4 reports the boxplots of the effectiveness measure values for feature location that were computed using (i) the combinatorial IR configuration (middle box-plot); (ii) the IR




(a) jEdit (b) JabRef

Figure 4-4 Box plots of the effectiveness measure for feature location on (a) jEdit and (b) JabRef

configuration identified and assembled by IR-GA (left box-plot); and (iii) the "ad-hoc" IR configuration that was used in a previous study [105], which also served as a reference point (right box-plot). For both systems, namely jEdit and JabRef, our IR-GA technique was able to produce an effectiveness measure that was close to the one identified in the combinatorial search. Moreover, as illustrated in Figure 4-4, when comparing the results achieved by IR-GA with the results produced by the reference configuration, we observe an improvement for both systems for all the quartiles. More specifically, for jEdit (see Figure 4-4 (a)) the median and mean values of IR-GA for the effectiveness measure are 36 and 196, whereas the corresponding median and mean values achieved by the reference configuration are 69 and 244, respectively. A similar trend is observed for the first and third quartiles where IR-GA obtains an effectiveness measure of 7 (vs. 10.25 for the reference) and 139 (vs. 211 for the reference) respectively.

Similarly, for JabRef (see Figure 4-4 (b)) the median and mean values of IR-GA for the effectiveness measure are 35 and 80, whereas the corresponding median and mean values

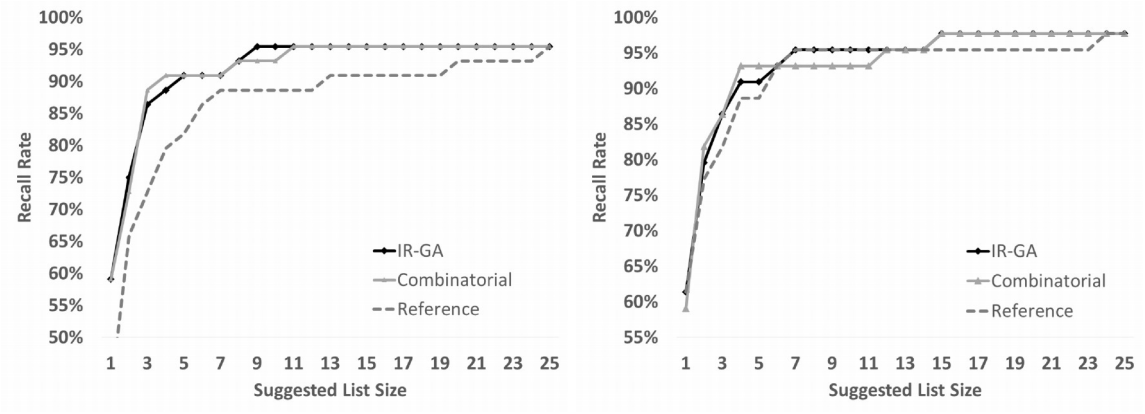
Table 4-5 Results of the Wilcoxon test for the Feature Location task

| Comparison | jEdit | JabRef |
|---|---|---|
| IR-GA < Combinatorial | 1 | 0.83 |
| IR-GA < Reference [105] | **<0.001** | 0.37 |
| Combinatorial < Reference [105] | **<0.001** | 0.37 |

achieved by the reference configuration are 58 and 130, respectively. In terms of the first and third quartiles, IR-GA obtains an effectiveness measure of 6 (vs. 8.5 for the reference) and 122 (vs. 134 for the reference) respectively.

For both jEdit and JabRef, IR-GA produced results that are very close to the ones obtained using a combinatorial search that iterated through more than one million configurations of preprocessing techniques, IR techniques (i.e., LSI, VSM and LDA) and their corresponding parameters. Among these configurations, the one that produced the best results in terms of the oracle was used as comparison with IR-GA and the reference. From Figure 4-4 we observe that distribution of the effectiveness measure provided by IR-GA is comparable to the distribution obtained by the combinatorial search. Moreover, for jEdit, the first quartile, median, third quartile and mean of the effectiveness measure obtained using the combinatorial search are 4.25 (vs. 7 for IR-GA), 27 (vs. 36.5 for IR-GA), 115 (vs. 139 for IR-GA) and 178 (vs. 195 for IR-GA) respectively. Similarly for JabRef, the combinatorial search and IR-GA produced very similar results. More specifically, the first quartile, median, third quartile and mean of the effectiveness measure obtained using the combinatorial search are 7 (vs. 6 for IR-GA), 29 (vs. 35 for IR-GA), 94.5 (vs. 122 for IR-GA) and 78 (vs. 80 for IR-GA) respectively. In fact, for the first quartile IR-GA had a better effectiveness measure than the combinatorial search (six as opposed to seven).

Table 4-5 reports the results of the Wilcoxon test (i.e., the adjusted p-values) for all combinations of the evaluated techniques. For jEdit IR-GA and the combinatorial search achieve statistically significantly better results than the reference, while at the same time we




(a) Short Corpus (b) 2ShortLong Corpus

Figure 4-5 Recall Rate graphs for Eclipse, with suggested list size raging between 1 and 25 for the: (a) Short Corpus and (b) 2ShortLong Corpus

observe that there is no significant difference between IR-GA and the combinatorial search. For JabRef, even if the boxplots from Figure 4-4 (b) reveal a better effectiveness measure distribution for both IR-GA and the combinatorial search as compared to the reference (which are backed up by values of the mean, median, and the quartiles), the results of the Wilcoxon tests indicate that these differences are not statistically significant. This suggests that the "ad-hoc" configuration (that was used a reference [105]) applied on the JabRef system is providing results very close to the ones obtained from the combinatorial search. However, the same reference approach did not provide the same level of accuracy on jEdit. In other words, the ad-hoc configuration was not able to reach the same level of accuracy achieved by both the combinatorial search and IR-GA configurations.

### 4.4.3 Task 3: Duplicate Bug Report Identification

For the identification of duplicate bug reports, we used two different corpora, referred as *Short* and *2ShortLong* (as suggested by [182]). In the former case, each bug is characterized by the bug title only (also known as short description), while in the latter we used both the title and the bug description (the title is weighted twice as much as the bug description). Note that in both cases we combined the textual information with dynamic information extracted from execution traces.

Table 4-6 Results of the Wilcoxon test for the Detection of Duplicate Bug Reports task

| Comparison | Short Corpus | 2ShortLong Corpus |
|---|---|---|
| IR-GA < Combinatorial | 1 | 0.68 |
| IR-GA < Reference [182] | **<0.001** | 0.20 |
| Combinatorial < Reference [182] | **<0.001** | 0.64 |

Figure 4-5 reports the recall rate for the results produced by using three different IR configurations that were identified using (i) IR-GA, (ii) a combinatorial search using various preprocessing steps, IR techniques and IR parameters, and (iii) a reference configuration which we used as a baseline. The reference configuration was instantiated using LSI where (i) $k$ (i.e., the dimensionality reduction factor for LSI) was set to half the number of documents [40]; and (ii) by applying a standard corpus preprocessing that is typical to bug duplicates [182] and other SE tasks [105].

For both the *Short* and *2ShortLong* corpora, IR-GA achieved virtually the same recall rate values as the combinatorial search. Moreover, IR-GA produced better results in terms of recall rate as compared to the reference (see Figure 4-5). In particular, when the corpus consists of bug report titles only (i.e., the *Short* corpus), the recall rate for IR-GA is significantly higher than the recall rate of the reference. For a cut point equal to 1, IR-GA has 59% recall rate (same as the combinatorial search) which is 18% greater than the reference recall rate of 41%. For cut points ranging from 2 to 5, the IR-GA recall rate is approximately 9-14% higher than the reference recall rate, whereas for cut points ranging from 6 to 24, IR-GA's recall rate is approximately 2-7% higher than the reference recall rate. The only case where both IR-GA's and reference's recall rates are the same is for cut point 25 (i.e., the last one) as observed from the graph in Figure 4-5 (a).

When the textual corpus is represented by the bug reports titles and their descriptions (i.e., the *2ShortLong* corpus), IR-GA is slightly better than the reference configuration. More specifically, the IR-GA recall rate is either equal to the reference rate, slightly higher by

approximately 2-5%. The gap between IR-GA and the reference is much higher for the *Short* corpus as opposed to the *2ShortLong* corpus, and one explanation for this finding could be the fact that when additional information is added to the corpus (i.e., a long description), the benefits of properly calibrating the an IR technique (i.e., using IR-GA) on a corpus with limited information are mitigated by the richness of additional textual information that is introduced, which could be sufficient even on an IR technique using ad-hoc configuration (i.e., as it was in the case of the reference). A similar phenomenon was observed in Chapter 2.3, when the benefits of utilizing a better splitting techniques were mitigated by the introduction of an additional source of information, namely dynamic information from execution traces.

It is worth emphasizing that the results produced when applying a properly calibrated IR process (such as the one assembled by IR-GA) on the *Short* corpus, are approximately the same results as the ones obtained using the *2ShortLong* corpus on an IR technique without calibration. Moreover, IR-GA produced approximately the same results on both the *Short* and *2ShortLong* corpora.

Table 4-6 reports the adjusted p-values of the Wilcoxon test for all combinations of the techniques. The results indicate that for the *Short* corpus IR-GA statistically outperforms the reference. However, for the *2ShortLong* corpus the small improvement introduced by IR-GA over the reference configuration (see Figure 4-5 (b)) is not statistically significant. When comparing IR-GA with the combinatorial search, we observe no significant difference for both *Short* and *2ShortLong* corpora.

### 4.4.4 Detailed description of the experimented IR Processes

In Sections 4.4.1, 4.4.2 and 4.4.3 we presented the results of our evaluation for the maintenance tasks of traceability link recovery, feature location and identification of duplicate bug reports, respectively. More specifically, we detailed the results of comparing the

Table 4-7 Comparison of different IR processes provided by IR-GA, combinatorial and reference. Table abbreviations: Rem. = Remove; CC = Camel Case; CC & KC = Camel Case & Keep-Compound Identifier; SL = remove stop words using standard list; $\leq X$ chars = remove words with less than (or equal to) $X$ characters; Cos. = Cosine

| Task System | Method | LSI k | Special Chars. | Digits | Term Splitting | Remove Stop Words | Stemmer | Weight Schema | Doc. Sim. |
|---|---|---|---|---|---|---|---|---|---|
| **Task 1: Traceability Link Recovery** | | | | | | | | | |
| EasyClinic | Combinatorial | 60 | Rem. | Include | CC | SL | Snowball | tf-idf | Cos. |
| | IR-GA | 53 | Rem. | Remove | CC | SL & ≤ 2 chars. | Snowball | tf-idf | Cos. |
| | Reference | 37 | Rem. | Remove | CC | SL | Snowball | tf-idf | Cos. |
| eTour | Combinatorial | 170 | Rem. | Include | CC | SL | Snowball | tf-entropy | Cos. |
| | IR-GA | 149 | Rem. | Remove | CC | SL | Porter | tf-idf | Cos. |
| | Reference | 87 | Rem. | Remove | CC | SL | Snowball | tf-idf | Cos. |
| iTrust | Combinatorial | 75 | Rem. | Include | CC & KC | SL & ≤ 2 chars. | Snowball | tf-idf | Cos. |
| | IR-GA | 79 | Rem. | Include | CC | SL & ≤ 3 chars. | Porter | log | Cos. |
| | Reference | 40 | Rem. | Remove | CC | SL | Snowball | tf-idf | Cos. |
| | | | | | | | | | |
| **Task 2: Feature Location** | | | | | | | | | |
| jEdit | Combinatorial | 1,150 | Rem. | Remove | CC & KC | SL & ≤ 3 chars. | Snowball | tf-idf | Cos. |
| | IR-GA | 1,028 | Rem. | Remove | CC & KC | SL & ≤ 3 chars. | Snowball | tf-idf | Cos. |
| | Reference | 300 | Rem. | Remove | CC | SL | Porter | tf-idf | Cos. |
| JabRef | Combinatorial | 416 | Rem. | Remove | CC | SL & ≤ 3 chars. | Snowball | tf-idf | Cos. |
| | IR-GA | 402 | Rem. | Remove | CC | SL & ≤ 2 chars. | Snowball | tf-idf | Cos. |
| | Reference | 300 | Rem. | Remove | CC | SL | Porter | tf-idf | Cos. |
| | | | | | | | | | |
| **Task 3: Detecting Duplicate Bug Reports** | | | | | | | | | |
| Eclipse Short | Combinatorial | 169 | Rem. | Include | CC | SL | Porter | tf-idf | Cos. |
| | IR-GA | 174 | Rem. | Include | CC & KC | SL | Snowball | tf-idf | Cos. |
| | Reference | 112 | Rem. | Remove | CC | SL | Porter | tf-idf | Cos. |
| Eclipse 2ShortLong | Combinatorial | 180 | Rem. | Include | CC | SL | Porter | tf-idf | Cos. |
| | IR-GA | 182 | Rem. | Include | CC | SL & ≤ 3 chars. | Snowball | tf-idf | Cos. |
| | Reference | 112 | Rem. | Remove | CC | SL | Porter | tf-idf | Cos. |

performances achieved by IR-GA, with the ones generated by a combinatorial search (over different preprocessing steps, IR techniques and parameters), and a baseline (or reference). In this section, we discuss the specific IR-configurations (i.e., the preprocessing steps, IR technique and IR-technique parameter values) that were identified by IR-GA, the combinatorial search and the reference. The IR-configurations of these approaches are presented in Table 4-7, where the rows are grouped by *task* (i.e., traceability link recovery, feature location and identification of duplicate bug reports), *dataset* (or systems) and *approach* (i.e., IR-GA, combinatorial, and reference), and the columns represent the steps of the IR-process, or the gene type as discussed in Section 4.2.1 and illustrated in Table 4-1.

From the third column of Table 4-7 we notice that for all tasks, datasets and approaches LSI was the IR-technique that was chosen by IR-GA and the combinatorial search, which tested LSI, VSM and LDA. For most baselines, LSI was the suggested technique and was configured with the dimensionality reduction factor $k$ of either 300 (for feature location tasks), or half the number of documents from the corpus (for traceability link recovery and identification of duplicate bug reports). Since LSI was identified to be the technique that produced the best results, the other IR techniques were not included in the Table 4-7. It is important to mention that there were other IR-configurations that used VSM and produced results close to the ones presented in Sections 4.4.1, 4.4.2 and 4.4.3, however, in the end LSI-based configurations outperformed them. On the other hand, IR-configurations that used LDA generated results that were not as close to the ones produced by LSI-based configurations. We identify two implications based on these observations. First, it is possible that there are other maintenance techniques or other datasets where LDA configurations will outperform LSI configurations, however, we did not observe this phenomenon in our experiments. Second, from a practical point of view, as LDA computations are in general more CPU and time consuming, the IR-GA technique could be configured to search through all the viable LSI and VSM configurations first, and then include searching through LDA configurations if necessary. This is a decision that is left to the researcher or practitioner.

When comparing LSI's dimensionality reduction factor $k$ for all three approaches we observe a clear pattern. For all tasks and dataset, the reference configuration uses a much lower $k$ value than the combinatorial or IR-GA. For example, for jEdit, the system with the highest number of documents in our experiments, the reference used $k = 300$, which is much lower than $k = 1{,}150$ (for combinatorial) or $k = 1{,}028$ (for IR-GA). Since the reference k is lower than the combinatorial and IR-GA $k$ value, and since in most cases IR-GA and the

combinatorial approach produced better results than the reference configuration, our findings suggest that a higher value for $k$ will produce better results. In terms of comparing the $k$ values for the combinatorial and IR-GA approaches, we observe that these values are very close to each other. This would indicate that IR-GA allows us to instantiate an IR process that is close to the one identified by the combinatorial one.

When comparing the preprocessing steps for character pruning (see Table 4-7 columns four and five), we observe that in all cases the special characters were removed, and in some cases, the combinatorial and IR-GA approaches chose to include the digits in the corpus. The reference always removed the digits because the assumption was that their contribution to providing meaning to the IR model that analyzes a corpus from source code was limited. This configuration step of removing digits was adapted to corpora that did not include source code. The assumption of removing digits from source code and the implication of always removing the digits from corpora that is not composed only of source code is illustrated in our findings. For the feature location datasets, where the corpus is composed of source code, all approaches removed the digits, which seems to enforce the assumption that digits do not carry enough information to warrant their inclusion. However, for the traceability link recovery and identification of duplicate bug report tasks, which include in their corpora use cases and bug descriptions respectively, the combinatorial and IR-GA approaches often choose the option to include the digits, as they might carry some meaning, which in turn would help improve the results over the baseline (which always removed the digits).

For the choice of splitting identifiers, in the majority of cases (i.e., 17 out of 21), the standard Camel Case splitting algorithm was applied (see Table 4-7 column six). For jEdit, both IR-GA and the combinatorial search chose the option to split identifiers using Camel Case, as well as keeping the original (compound) identifier, which has the potential to carry an

important meaning especially in the cases of large systems such as jEdit. For example if identifier $ab$ is always split into $a$ and $b$, some if its meaning is lost in a large system where there is a potential to be numerous $a$'s and $b$'s. However if the original compound identifier $ab$ is kept, it can be easily matched, because odds are that $ab$ appears less often (thus has more discriminative power) than $a$ and $b$.

If we compare the preprocessing steps of removing stop words and stemming (see Table 4-7 columns seven and eight) we observe that in all cases a standard list of stop words was used (i.e., no approach was configured by keeping programming language keywords or identifiers frequently used in English, such as *a*, *the*, *in*, etc.). In addition, in some cases IR-GA and the combinatorial search choose to also remove identifiers with less than two or three characters, in order to reduce the noise from the IR model. In terms of stemmers, every approach used either the Snowball or Porter stemmer (i.e., no approach was configured with no stemming at all).

For all approaches the chosen metric to compute the document to document similarity was the *cosine similarity* (see Table 4-7 last column). Moreover, for the majority of cases (i.e., 19 out of 21) the *tf-idf* measure was chosen as the preferred weighting schema (see Table 4-7 column nine). However, the combinatorial approach chose *tf-entropy* as the weighting schema for eTour, and IR-GA choose the *log* weighting schema for iTrust. Previous studies [33] proposed sophisticated techniques to search for adequate indexing schemas, and our results reflect that by showing that in most cases *tf-idf* is the preferred one, but there are also cases where other weighting schemas are most suited.

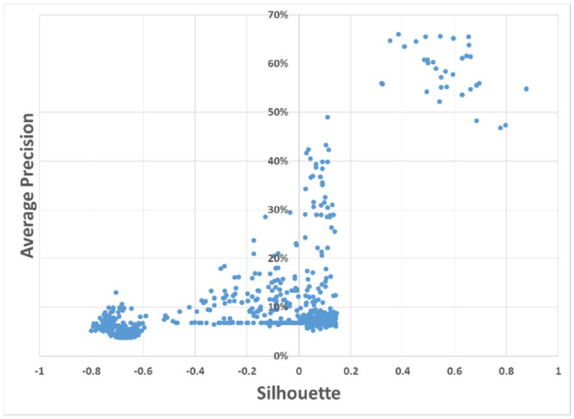
Although we can identify some patterns when analyzing each preprocessing step individually, we cannot determine a clear pattern when we take all the preprocessing steps and the configuration of the IR techniques as a whole, but we can observe a clear variation in the

choices. The fact that not all the tasks and/or datasets require the same preprocessing steps confirm the findings of Falessi et al. [61] that there is no unique IR configuration that can be efficiently applied to all the tasks and all the datasets. Thus, IR-GA plays a key role into considering all the potential configurations of an IR-technique, and all their potential interactions as a whole (i.e., as opposed to considering them individually) and recommends the configuration that is most suited for a dataset, as each dataset is unique. We observed that besides recommending the preprocessing steps, IR-GA was able to recommend the dimensionality reduction factor $k$, and the overall IR configuration was able to produce better results than the baseline. Moreover, our comparison between IR-GA and the combinatorial search indicate that IR-GA was able to find a suitable configuration (using the Silhouette coefficient of the underlying model) that is close to the configuration identified by the combinatorial search, which used the oracle to identify the best configuration.

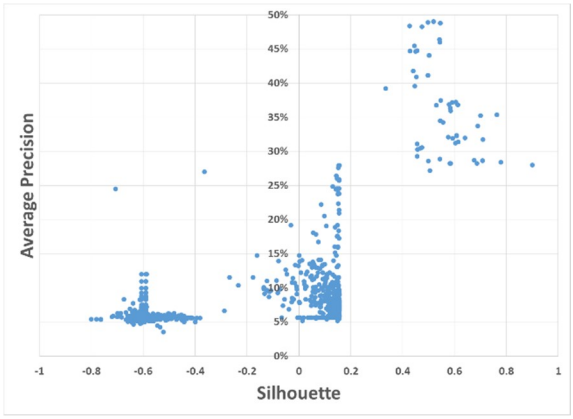
### 4.4.5 Relation between the Silhouette coefficient and performance of maintenance tasks

Figure 4-6 presents scatter plots that illustrate the relation between the Silhouette coefficient and the average precision for the three traceability link recovery systems, namely EasyClinic, eTour and iTrust. Each point in the graph corresponds to one IR configuration that was generated by randomly selecting its preprocessing options, IR technique and IR settings. The search space for choosing this random configuration is the same search space of parameters that IR-GA uses to find the configuration with the highest Silhouette coefficient. In each graph, the x-axis corresponds to the value of the Silhouette coefficient (which is between -1 and 1) and the y-axis corresponds to the value of the average precision generated by the IR model. For each graph, there are 1,000 points corresponding to random IR instances.

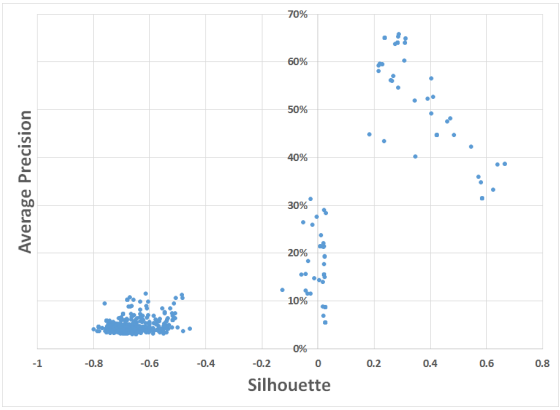
For each system in Figure 4-6, we can observe around three "clusters" forming approximately in the lower-left (i.e., low Silhouette and low average precision), lower-center




(a) EasyClinic




(b) eTour




(c) iTrust

Figure 4-6 Scatter plots that illustrate the relation between the Silhouette coefficient and the average precision for (a) EasyClinic, (b) eTour and (c) iTrust

(i.e., medium Silhouette and low-medium average precision) and upper-right (i.e., high Silhouette and high average precision) of the graph.

The first cluster is in the lower-left corner of the graph and captures instances of IR models where the Silhouette coefficient is low (e.g., less than -0.6 for EasyClinic and less than -0.4 for eTour and iTrust) and the average precision is low as well (e.g., less than 15% for all three systems). These finding aligns with one of our initial hypotheses that a lower Silhouette coefficient leads to lower performance in results.

The second cluster is located approximately in the lower-center portion of the graph and corresponds to IR models where the Silhouette coefficient has a medium value (e.g., between -0.4 and 0.2 for EasyClinic, between -0.1 and 0.2 for eTour and between -0.2 and 0.1

for iTrust) and the average precision is in general low (e.g., less than 20% for EasyClinic and less than 15% for eTour). However, for all three systems we also observe instances where for a small range of Silhouette coefficient values we have low to medium average precision values. For example, for EasyClinic we have an average precision of about 5%-50% for a Silhouette coefficient ranging between 0 and 0.2. In addition, for eTour the average precision ranges between 5%-30% for Silhouette values between 0.1 and 0.2. Finally, for iTrust we observe a range of average precision values between 0%-30% that correspond to a Silhouette coefficient value of approximately 0 value. For all three systems, the high variation in the average precision for a small range of Silhouette values could be attributed to the property of the Silhouette coefficient, which characterizes the cohesiveness and separation of clusters. A medium Silhouette value (e.g., around 0) will correspond to clusters that are not well defined, meaning that they are not very cohesive and not well separated from one another. This could explain the high variability in average precision for IR models with Silhouette coefficient close to 0 (i.e., the medium value).

The third cluster is located in the upper-right corner and concentrates values where the silhouette coefficient is high (e.g., more than 0.3 for EasyClinic, more than 0.4 for eTour and more than 0.2 for iTrust) and the average precision is also consistently high (e.g., more than 50% for EasyClinic, more than 30% for eTour and iTrust). We can observe that although a high Silhouette value corresponds to a high average precision, there is also a variability in the results and there is no clear linear relationship between the Silhouette and the average precision. In other words, the IR model with the highest, second highest, third highest, etc. Silhouette coefficient does not correspond to the highest, second highest, third highest, etc. average precision. However, considering all the results, a high Silhouette coefficient (see upper-right corner) corresponds to a high average precision as compared to a medium or low Silhouette coefficient. We acknowledge that the lack of a clear linear relation between the

Silhouette coefficient and the performance in a maintenance task is a limitation of the presented approach.

## 4.5 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. For the three tasks investigated, we evaluated the performances of IR-GA and of alternative approaches using well-established metrics, namely precision, recall, f-measure, effectiveness measure and Recall Rate, and oracles already used and validated in previous studies [17, 53, 65, 66, 182]. Finally, we used as baseline of comparison performances achieved using IR processes and calibrations used in previous papers. As for detecting duplicate bug reports, it was not possible to fully replicate the approach of Wang et al. [182] due to the unavailability of all the required information. However, Section 4.3.3 explains the details and the rationale for using a baseline for comparison for such a task.

Threats to *internal validity* are related to co-factors that could have influenced our results. As discussed in Section 4.3.4, different values of the GA parameter were shown to produce different results. We limited the influence of GA randomness by performing 30 GA runs, and considering the configuration achieving the median performance.

Threats to *conclusion validity* concern the relationship between treatment and outcome. To support our claims, we used non-parametric statistical tests (i.e., Wilcoxon rank sum test). In terms of the relation between the Silhouette coefficient and the performance in a maintenance task (see Section 4.4.5), although our experiments indicate high performance for high values of the Silhouette (and vice-versa), this relation is not clearly linear.

Threats to *external validity* concern the generalization of our results. First, we consider only a subset of the possible treatments for the various phases such as term extraction, stop words removal, stemming, and term weighting. Although the chosen treatments are well

representative of most of the ones used in literature, it is worthwhile to investigate further possibilities. Second, we applied IR-GA to calibrate IR processes for artifacts to be used in three tasks that we believe are crucial in many SE activities, however it would be worthwhile to experiment it on further tasks.

## 4.6 Related Work on Configuring IR Techniques for SE Tasks

This section describes related work concerning the importance of choosing the right treatments for different phases of an IR process when applied to SE tasks. It also reports approaches aimed at suggesting calibrations for specific IR techniques and IR phases.

Falessi et al. [61] empirically evaluated the performance of IR-based duplicate requirement identification on a set of over 983 requirement pairs coming from industrial projects. To this aim, they instantiated 242 IR processes, using various treatments for stemming, term weighting, IR algebraic method, and similarity measure. Their study shows how the performances of the duplicate requirement identification significantly vary for different processes. The work by Falessi et al. motivates our research, as it empirically shows that instantiating an appropriate IR process is crucial to achieving good performances. However, while they do not propose an approach to choose the most suitable process, our IR-GA approach searches for a (near) optimal IR process using GAs, and above all, it is able to do it without the availability of an oracle.

Previously we proposed LDA-GA [134] (see Chapter 3), a GA approach to automatically calibrate the parameters of LDA in the context of three software engineering tasks, namely traceability link recovery, feature location and software artifact labeling. In contrast with LDA-GA, in the approach introduced in this chapter we leveraged GAs to instantiate the entire IR process, rather than just tuning the parameters of a specific IR algebraic method. More specifically, our results are in concordance with the findings of Falessi et al.

[61], which illustrate that performances of IR processes depend on the choices made for the various phases of IR processing, rather than on the appropriate choice and calibration of the IR algebraic model.

Lohar et al. [107] introduced a traceability link recovery approach that is configured at runtime for a particular dataset. Their approach utilizes machine learning to identify the best configuration for traceability link recovery (among a set of traceability link recovery techniques and their specific configurations) using a training set of validated traceability links. The main difference between our proposed approaches, namely LDA-GA and IR-GA, and Lohar et al.'s [107] approach is that our techniques are unsupervised and they do not require a-priori knowledge of the oracle. In other words, our techniques will generate a configuration based on the input dataset set alone, whereas their technique requires knowledge of part of the oracle to be used as input for training their approach.

The literature also reports approaches for calibrating specific stages of an IR process, or parameters of specific algebraic IR techniques. Cordy and Grant have proposed heuristics for determining the "optimal" number of LDA topics for a source code corpus of methods, by taking into account the location of these methods in files or folders, as well as the conceptual similarity between methods [70]. Cummins [33] proposed to use genetic programming (GP) to automatically build term weighting formulae, using different combinations of *tf* and *idf* that can be altered using functions such as logarithm. The similarity between our approach and Cummins' approach is the use of search-based optimization techniques to calibrate IR processes. Their approach was evaluated on a set of 35,000 textual documents, for a document search task. In contrast to our technique, their approach (i) focuses on term weighting only (whereas we focus on the whole process), and (ii) their approach is supervised, as the fitness function evaluation requires the availability of a training set (e.g., labeled traceability links). Griffiths and Steyvers [74] propose a method for choosing the best number of topics for LDA

among a set of predefined topics. Their approach consists of (i) choosing a set of topics, (ii) computing a posterior distribution over the assignments of words to topics, (iii) computing the harmonic mean of a set of values from the posterior distribution to estimate the likelihood of a word belonging to a topic, and (iv) choosing the topic with the maximum likelihood.

In the context of clone detection, Wang et al. [181] introduced EvaClone (and CloudEvaClone) an approach that uses a genetic algorithm to identify a suitable configuration for a clone detection tool, which produces better results that the default settings of clone tools. Similarly to our work, their approach also aims at addressing the confounding configuration choice problem (i.e., the problem of not knowing apriori which values to choose for the parameters of clone detection tools).

With respect to the previously described approach, we claim this is the first approach to propose an automatic calibration of an entire IR process. Also, with respect to many other approaches, IR-GA is task independent and does not require any oracle or training set to perform the calibration. In addition, the outcome of the calibration only depends on the specific artifacts provided as inputs, while it does not depend on the specific task.

## 4.7 Discussion

The application of IR techniques to software engineering problems requires a careful construction of a process consisting of various phases, i.e., term extractions, stop word removal, stemming, term weighting, and application of an algebraic IR method. Each of these phases can be implemented in various ways, and requires careful choice and settings, because the performances significantly depend on such choices [61].

This chapter proposes the use of Genetic Algorithms to assemble a (near) optimal IR process to be applied to given software artifacts, e.g., when processing such artifacts to solve problems such as traceability link recovery or feature location. Noticeably, the proposed

approach is unsupervised and task independent, as it evaluates the extent to which the artifacts can be clustered after being processed.

We applied the proposed approach IR-GA to three software engineering tasks, namely traceability link recovery, feature location, and detection of duplicate bug reports. Results of our empirical evaluation indicate that for traceability recovery and feature location, the IR processes assembled by IR-GA significantly outperform those assembled according to what previously done in literature. For duplicate bug report detection, the obtained results do not always significantly improve the performances of the baseline approach, as such a baseline is already close to the "ideal" optimum, which was identified by a combinatorial search over a discretized search space for the configuration parameters. However, in most cases, the performances achieved by IR-GA are not significantly different from the performances of an "ideal" IR process that can be combinatorially built by considering all possible combinations of treatments for the various phases of the IR process, and by having a labeled training set available (i.e., by using a supervised approach). The raw data used for the three tasks and working data sets used for the statistical analysis are available in a replication package [1].

# 5 Supporting Reproducible Empirical Research using TraceLab Component Library

In the previous chapters we introduced LDA-GA (Chapter 3) and IR-GA (Chapter 4), which are two examples of approaches that were designed to support software maintenance tasks, that contain numerous implementation details, and that were evaluated in large empirical studies. In software maintenance, oftentimes research is driven by empirical studies, and advancing this field requires researchers not only to come up with new, more efficient and effective approaches that address software maintenance problems, but most importantly, to compare their new approaches against existing ones in order to demonstrate that they are complementary or superior and under which scenarios. We address this problem by facilitating the reproducibility of the approaches presented in this dissertation, and we present the details of a framework that was specifically designed to support creating, running and sharing empirical research across various software engineering tasks. Before providing the details of this framework, we emphasize the current problem of comparing an approach against existing ones, which is a requirements for advancing the field. Comparing approaches is not only time consuming but also error-prone. For instance, existing approaches may be hard to reproduce because the datasets used in their evaluation, the tools and implementation, or the implementation details (*e.g.*, specific parameter values, environmental factors) are not available [12, 23, 34, 52, 69, 129, 158].

These problems are illustrated through a survey on feature location (FL) techniques by Dit *et al.* [52], which revealed that only 5% of the papers surveyed (i.e., three out of 60 papers) evaluated their approach using the same dataset used to evaluate other techniques, and that

only 38% of the papers surveyed (i.e., 23 out of 60 papers) compared their proposed feature location technique against any previously introduced feature location techniques. In addition, these findings are consistent with the ones from the study by Robles [158], which determined that among the 154 research papers analyzed, only two made their datasets and implementation available, and the vast majority of the papers describe evaluations that cannot be reproduced, due to lack of data, details, and tools. Furthermore, a study by González-Barahona and Robles [69] identified the factors affecting the reproducibility of results in empirical software engineering research and proposed a methodology for determining the reproducibility of a study. Similarly, Borg et al. [23] conducted a mapping study investigating the relationship between evaluation criteria and results for traceability link recovery approaches based on information retrieval. Their findings revealed that most studies were evaluated against datasets with fewer than 500 artifacts and, as a result, they identified the need for performing case studies on industrial-size datasets. They encouraged researchers to publicly provide the datasets and tools used in their evaluations and also provided a set of guidelines to raise the quality of publications in the field of software engineering research. In another study, Mytkowicz *et al.* [129] investigated the influence of the omitted-variable bias (*i.e.*, a bias in the results of an experiment caused by omitting important causal factors from the design) in compiler optimization evaluation. Their study showed that factors such as the environment size and the link order, which are often not reported and are not explained properly in the research papers, are very common, unpredictable, and can influence the results significantly. Moreover, D'Ambros *et al.* [34] argued that many approaches in bug prediction have not been evaluated properly (*i.e.*, they were either evaluated in isolation, or they were compared against a limited set of other approaches), and highlighted the difficultness of comparing results.

This issue of the reproducibility of experiments and approaches has been discussed and investigated in different areas of software maintenance research [12, 23, 34, 52, 69, 129, 158,

168], and some initial steps have been taken towards solving this problem. For example, efforts for establishing datasets or benchmarks that can be used uniformly in evaluations have resulted in online benchmark repositories such as PROMISE [124, 163], Eclipse Bug Data [188], SEMERU feature location dataset [52], Bug Prediction Dataset [34], SIR [55], and others. In addition, different infrastructures for running experiments in SM and other fields were introduced, such as TraceLab [28, 30, 91], RapidMiner [153], Simulink [117], Kepler [92], and others. However, among these, a good candidate framework for facilitating and advancing research in software engineering and maintenance is TraceLab (see Section 5.2.2 for an in-depth comparison and discussion of TraceLab's features with other tools). More specifically, unlike the other frameworks, TraceLab is a plug-and-play framework that was *specifically designed for facilitating creating, evaluating, comparing, and sharing experiments* in software engineering and maintenance (see Section 5.2.1 for a detailed description of its features). These characteristics ensure that TraceLab makes experiments *reproducible*.

The goal of this chapter is to ensure that a large portion of existing and future experiments in software maintenance research that are designed and implemented with TraceLab will be *reproducible*. To accomplish this, we analyzed the approaches presented in 27 SM research papers, identified their common building blocks, and we implemented them as components in a well-organized, structured, documented and comprehensive *Component Library* for TraceLab. In addition, we used the *Component Library* to assemble and replicate a subset of the existing SM techniques, and exemplified how these components and experiments could be used as starting points for creating new and reproducible experiments. Moreover, we illustrate how the *Component Library* can be used to reproduce LDA-GA or IR-GA experiments.

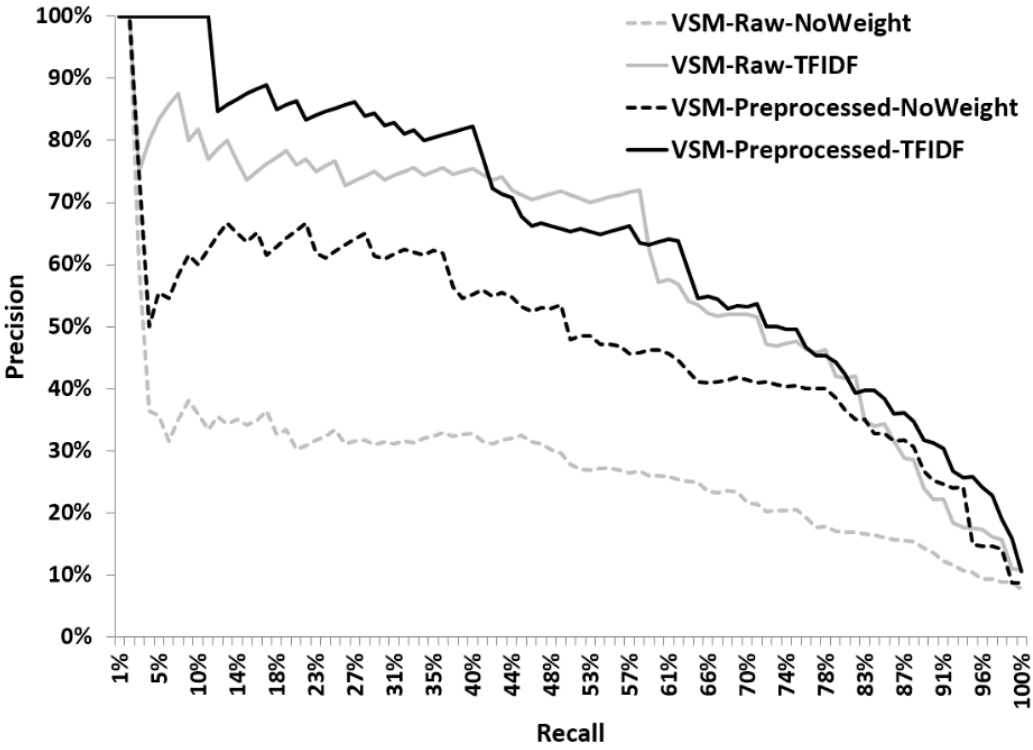



Figure 5-1 Precision-Recall curves for EasyClinic for recovering traceability links between use cases and classes using a VSM-based traceability technique and different preprocessing techniques (raw – gray color, preprocessed – black color) and weighting schemes (no weight – dash line, tf-idf – solid line)

## 5.1 Motivating Example

When new approaches are introduced, in general, authors rightfully focus more on describing the important details of the new techniques, and due to various reasons (*e.g.*, space limitations) they may present only in passing the details of applying well-known and popular techniques (*e.g.*, VSM), as they rely on the conventional wisdom and knowledge (or references to other papers for more details) about applying these techniques [52, 158].

However, for a researcher who tries to reproduce the results exactly, it might be difficult to infer all the assumptions the original authors took for granted and did not explicitly state in their publication. Therefore, the reproducer's interpretation of applying the approach could have a significant impact on the results.

To illustrate this point with a concrete example, we applied the popular IR technique Vector Space Model [160] on the EasyClinic system from TEFSE 2009[11] challenge to recover traceability links between use cases and class diagrams. We configured the VSM technique using four treatments consisting of all the possible combinations of two corpus preprocessing techniques and two VSM weighting schemes. The preprocessing techniques were *raw preprocessing* (*i.e.*, only the special characters were removed) and *basic preprocessing* (*i.e.*, special characters were removed, identifiers were split and stemmed). The weighting schemes used were *no weighting* and *term frequency-inverse document frequency* (*tf-idf*) weighting [160].

Figure 5-1 shows the precision and recall curves for recovering traceability links between use cases and classes on the EasyClinic dataset, using a VSM-based traceability technique and different preprocessing techniques (i.e., *raw preprocessing* in gray color and *basic preprocessing* in black color) and weighting schemes (i.e., *no weighting* in dash line and *tf-idf weighting* in solid line). The results in Figure 5-1 show a high variety in the precision and recall values, based on the type of preprocessing and weighting schemes used. Assuming these details are not clearly specified in the paper, any of these configurations or variations of these configurations can be chosen while reproducing an experiment, potentially yielding completely unexpected and drastically different results. It is worth emphasizing that in our example we picked a small subset of the large number of weighting schemes and preprocessing techniques that can be found in the literature, and these options were deliberately picked to illustrate an example, as opposed to conducting a rigorous experiment to identify the configuration of factors that could produce the best results.

[11] http://web.soccerlab.polymtl.ca/tefse09/Challenge.htm

The main point of this example is that even in this simple scenario of using VSM for a typical traceability task, there are many options on how we can instantiate and use this technique, which leads to completely different results. However, all these problems could be eliminated if all these details are encoded in the experiment description, for example, by designing an experiment in TraceLab.

## 5.2 Background and Related Work

This section provides the background details about TraceLab as an environment for SM research and compares and contrasts TraceLab to other research tools specific to other domains.

### 5.2.1 TraceLab

TraceLab [28, 30, 91] is a framework designed to support the reproducibility of experiments in software engineering and software maintenance. More specifically, it provides a visual workbench (see Figure 5-2) that allows researchers to create, evaluate, compare, and most importantly share experiments in SM research. TraceLab was developed at DePaul University in collaboration with researchers at the College of William and Mary, Kent State University, and University of Kentucky, and it is already being used by numerous users throughout the world.

The heart of a TraceLab experiment lies in its workflow of *components* (see Figure 5-2 (1)). Components are reusable user-defined code units that are designed to accomplish a very specific task. They exchange data with other components through their inputs and outputs via shared memory. The components are represented in TraceLab as ovals (see examples from Figure 5-2 (1) and Figure 5-3).

An *experiment* is a collection of components (or nodes) connected in the form of a precedence graph. The execution of an experiment begins at the "Start" node and continues

along every path until the “End” node is reached, thus completing the experiment. Since it is a precedence graph, unless otherwise specified, each node must wait for all of the incoming edges to complete before executing. This ensures that the previous techniques have completed their execution and the correct data is available. The execution of components in TraceLab was designed to be parallelizable. Each component is given its own copy of the data and is run in a separate thread. The main reason for this design decision was to ensure that running experiments will not encounter any errors caused by nondeterministic behavior triggered by race conditions, which means that developers do not have to worry about race conditions when designing their own custom components. Therefore, when two components branch out from a parent component (e.g., components "Import Use Cases" and "Import Code Classes" in Figure 5-2 (1)) they each will run concurrently and independently. This feature is built into the TraceLab framework, and it will be automatically applied to all execution paths of an experiment.

TraceLab was designed to run experiments fully automatically and without any interaction from the user. Moreover, the same experiment can be applied to multiple datasets, making it a great tool for batch experiment processing. A TraceLab experiment essentially takes as input (i) data (which, depending on the type of experiment was extracted from software systems, version control systems, issues tracking systems, execution traces, etc.) and (ii) an oracle (or ground truth) that was also generated using external tools or human support. Next, this data is typically imported, converted to appropriate internal TraceLab datatypes, processed, and the evaluated based on a set of predefined metrics and the oracle. The results of an experiment could be visualized or exported for further analysis.

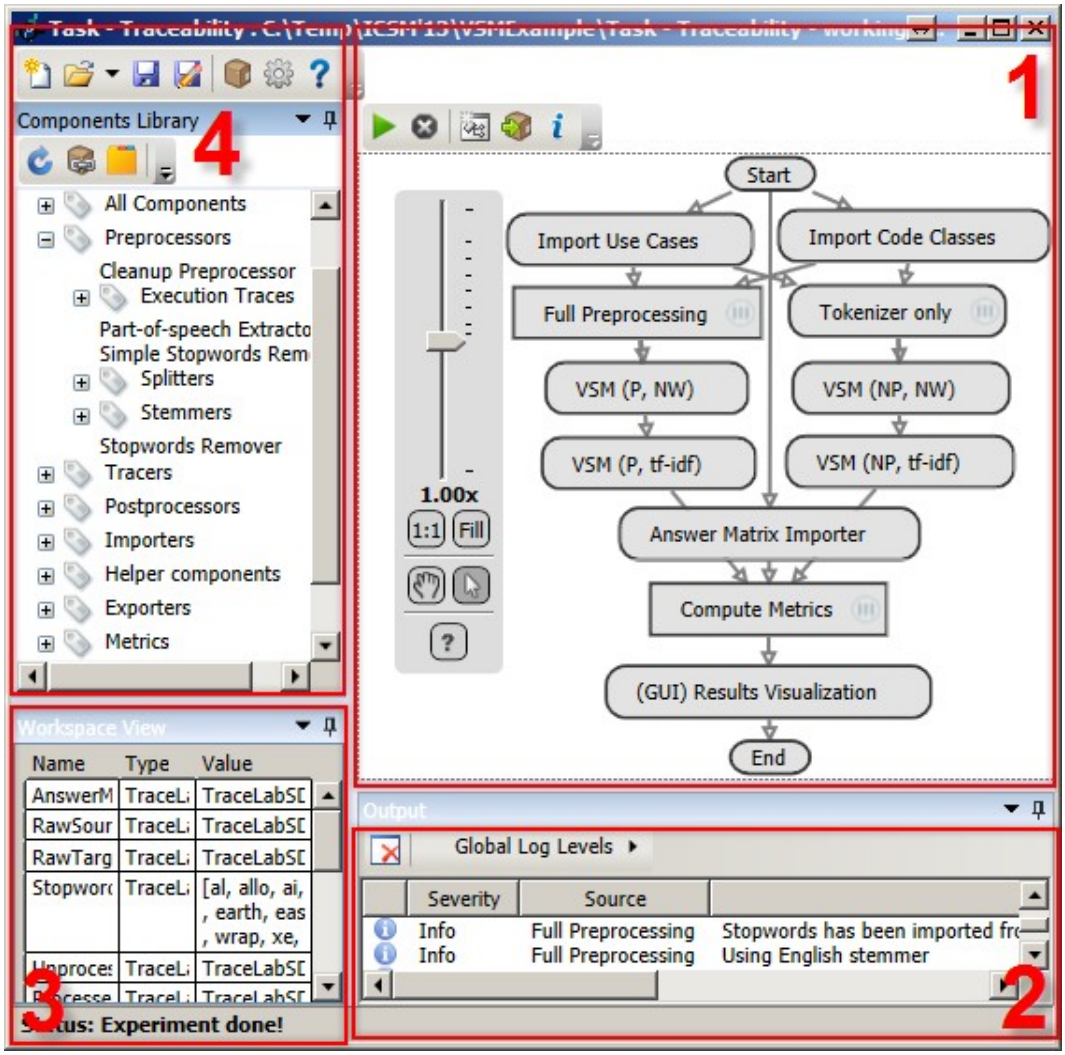



Figure 5-2 The four "quadrants" of TraceLab in clockwise order from top-right are (1) the sample TraceLab experiment that implements our motivating example in Section 0; (2) an output window for reporting execution status of an experiment; (3) the Workspace containing the data and the values of the experiment; and (4) the *Component Library*

#### *5.2.1.1 TraceLab Features*

TraceLab provides many control elements that allow flexibility when designing an experiment. For example, *Goto decisions* (see Figure 5-3 (a)) allow execution redirection to any of the outgoing nodes based on a given condition. *If statement decisions* (see Figure 5-3 (b)) provide additional control, by allowing execution of one of a number of sub-graphs (called scopes) based on a given condition. Scopes provide independent experiment sub-graphs that execute in their own namespace and once completed, provide control to the parent graph.

Similarly, *While loops* (see Figure 5-3 (c)) repeatedly execute the scope as long as the given condition is true.

The *workspace* (see Figure 5-2 (3)) is the data-sharing interface that allows components to communicate with the preceding and following nodes. Components can load and store data to and from the workspace only for their declared inputs and outputs. Data may also be read from the workspace for use in a control-flow node. Any information in the workspace may be serialized to disk as an XML file for later use or debugging purposes. Additionally, any data type with a viewer (including all standard types) can be viewed from the workspace by clicking on their workspace entry. There are already a large number of predefined datatypes and components to handle these datatypes (e.g., importers, exporters, etc.), but if needed, researchers can adapt existing datatypes or create custom ones to fit their needs. Some of the predefined data types include `TLArtifact` (i.e., a generic data type that can represent any textual software artifact, such as requirements, design specifications, UML diagrams and defect logs, test cases, or software code; it has two fields, namely ID and textual information), `TLArtifactsCollection` (i.e., a collection of `TLArtifacts`), `TLSimilarityMatrix` (i.e., a datatype that represents the set of links from source artifacts to target artifacts with assign probability score of their relationship; this matrix can be used either as standard similarity matrix, an answer matrix, a traceability matrix, etc.). Moreover, the predefined data types include `Lists`, `HashTables`, `Dictionaries`, etc., which can be used for various tasks ranging from representing stop words to storing different values for the Box Plot points used to represent the results. In our experiments, we used a Program Dependence Graph (PDG), which stores basic information about the nodes and dependencies in the graph. However, since most applications that use a PDG require various types of information, practitioners are free (i) to use our PDG as is, (ii) to customize it for their unique needs (i.e., refining it), or (iii) to create

a brand new PDG data type. In either case, an existing experiment that was exported and shared through TraceLab's packaging feature will still reference the correct component, regardless of the existence of other versions of the component. In other words, a shared experiment will still reference the "old" PDG (i.e., the one that was used for creating the experiment), and will not be affected by the newly created PDG (which can be part of future experiments).

The status of an executing experiment is reported in the *Output view* (see Figure 5-2 (2)) in the form of messages displayed to the user. The messages have different levels of severity, such as info, trace, debug, warning, and error. Each message displays the component name, severity, custom message provided by the author of the component (if any), and optionally an exception dialogue describing an uncaught exception and a stack trace.

A major contribution of this chapter is a *Component Library*, designed to implement a wide range of SM techniques that can be easily accessed from TraceLab (see Figure 5-2 (4)) to build and execute experiments. The *Component Library* is included in the distribution of the official TraceLab release.

The component library provides a set of tools and techniques to researchers for using in experiments. Components may be categorized by multiple tags, both by component developers and users. In order to use a component in an experiment, the user only needs to drag-and-drop the component from the component library into the experiment (see Figure 5-2 (1)) and connect it with the other components. Each component has a set of metadata that uniquely identifies it within TraceLab. The primary identifier is the component's name, which appears in the component library and on the component node within the experiment. Components contain additional information such as a description, author, versioning information (see Figure 5-4), inputs and outputs. For example, if a component takes in two sets of artifacts and produces a ranked list of similarities between the two, it must explicitly declare

two `TLArtifactsCollection` objects as input and declare a `TLSimilarityMatrix` as output (see Figure 5-4). This allows TraceLab to evaluate the experiment graph before running it, checking for valid inputs and control-flow errors. For example, if a component declares an input that is not an output of any preceding components, TraceLab will catch the error before the experiment starts. In addition to inputs and outputs, a component can allow the configuration of specific settings (e.g., weighting scheme for the Vector Space Model – see Figure 5-4).

An important feature of TraceLab consists of generating *composite components*. More specifically, a group of individual components that are often used together as a group to accomplish a specific task can be combined to form *composite components*. This feature provides an additional level of abstraction for common functionality, it improves the reusability of components in the same experiment and across different experiments, and it improves the readability of the graph experiment. An example of such a composite component is the node with rectangular edges labeled *Compute Metrics* in Figure 5-2 (1), which evaluates the performance of several techniques by encapsulating the functionality of various components, such as computing the precision and computing the recall.

In addition to all these features, arguably the most important and distinctive feature provided by TraceLab is the *packaging* feature for encapsulating and sharing experiments. In order to share a TraceLab experiment, all of the necessary information (i.e., data, components and settings) must be included. Therefore, the packaging feature of TraceLab allows a user to encapsulate all the datasets and custom components used in the experiment, including all the dependencies and specific versions of datasets and components. The resulting self-contained experiment can be shared with the research community. The original experiment along with exact data and settings can be run by other researchers by unpacking and installing the original

package experiment, using the associated functionality in TraceLab. Paths are relative to the package and they reference the data "inside" the package. Therefore, a shared package can be used as is by other researchers regardless of the location on their machine, because the paths are relative to the content of the package. The packaging feature not only allows to include data, but it can also reference existing packages which contain experiments and/or data. In other words, a researcher does not have to include the same data in different experiments, because she can choose to reference them. Once an experiment has been imported from a package, prior to running the experiment, one can change the configuration of the components that load/save data (i.e., if they want to use the same workflow on different data), or one can add/remove components to alter the workflow of the experiment if needed. In fact this one of the strengths of TraceLab. It ships with multiple importers, capable of importing data from multiple formats and converting it into standard data structures, while, at the same time, allowing the creation of new customized data importers. As a result, experiments can easily be run against different datasets. We would like to emphasize that there are two major reasons we include datasets in the packages. First, it ensures that the experiments are executable "out of the box", and we have observed that new users find it far easier to change a data source of an experiment that actually runs correctly, rather than having to find and configure the data to run the experiment. Second, one of the goals of TraceLab is to help new researchers get started, which implies providing them not only with the experiments, but also with the data.

It is important to highlight that researchers can create as many component types as they want (although we encourage reuse of components if possible). Each experiment consists of a particular dataset (which has a particular format and version), and a specific set of components (with their own version, configuration options) and dependencies between components. Once an experiment is created, and shared, it will reference specific versions of components and data

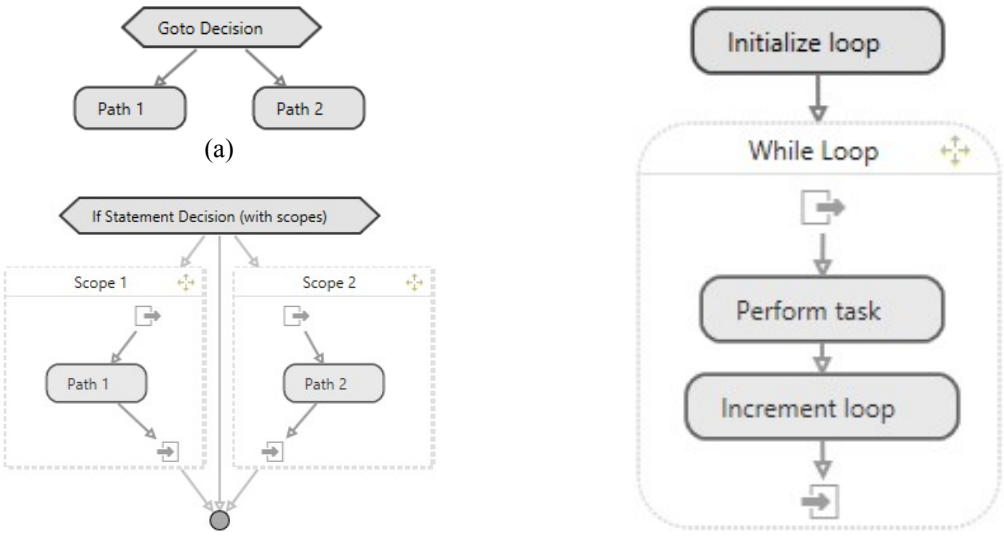



Figure 5-3 Control flow options provided by TraceLab: (a) Goto decision, (b) If statement and (c) While loop

types, even though some components may have evolved in the meanwhile. Therefore, the evolution of existing techniques will not affect the reproducibility of existing experiments.

We also note that our philosophy in creating TraceLab was to allow flexibility in the way components are designed, and we illustrated this with an example. We recently developed a new Weka component for a series of experiments, and this component is in effect a wrapper for the underlying Weka library. We needed to choose between (i) creating multiple TraceLab-Weka components to perform specific classification functions (e.g., a J48 component, a Bayes Regression Classifier, etc.) or (ii) creating one generic component, which served as a general Weka wrapper. We opted for the second choice, which means that the component can be configured to run any classifier. This reduced the proliferation of components, but at the same time placed more onus on the component user to understand Weka's configuration parameters. The point is that TraceLab allows the experimenter to make such choices, and this was a deliberate design decision on our part.

#### *5.2.1.2 Supported Languages*

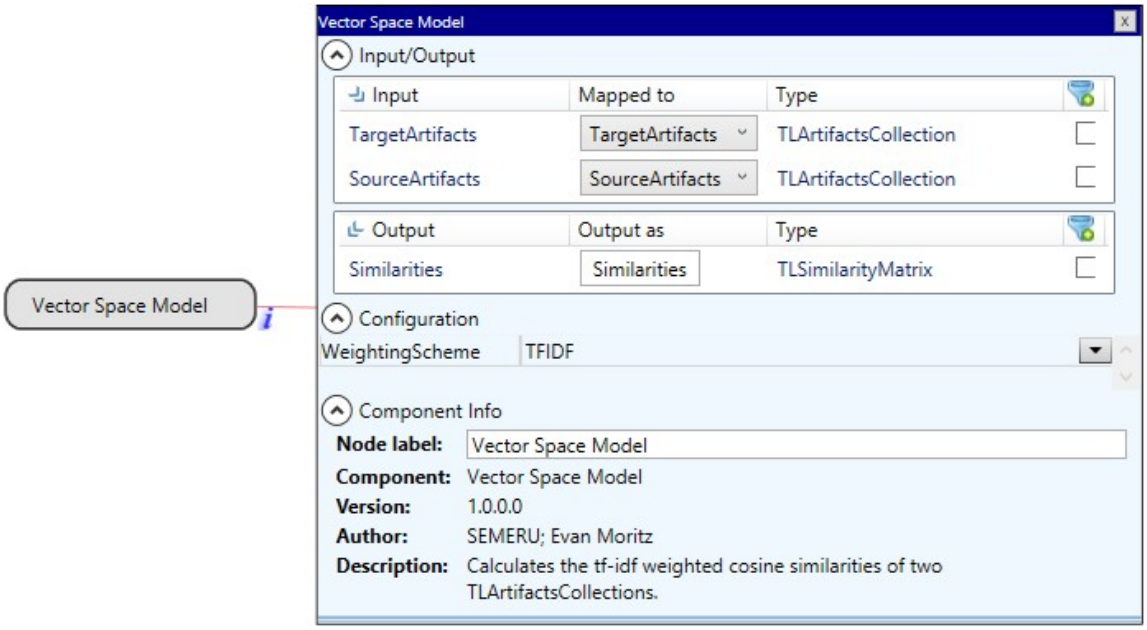



Figure 5-4 Information pane for the Vector Space Model TraceLab component (left). The information pane shows the inputs and outputs, settings (e.g., weighting scheme) and other metadata information.

TraceLab ships with a software development kit (SDK) that allows users to define their own custom components and types in .NET languages, Java, and via plugins, R [152], Matlab [116] and Weka [180].

Any .NET language that compiles to a Dynamic Linked Library (DLL) may be used to create user-defined components and types. This includes Visual Basic, C++, C#, and F#.

Developers can create user-defined TraceLab components and types in Java using IKVM.NET[12]. After compiling the Java components, the JAR file is converted to a DLL through IKVM. Thus, when called in TraceLab, the Java code is actually run in the IKVM virtual machine.

In addition to .NET and Java, TraceLab components can execute R code. Although tools like R.NET[13] exist for running R code in .NET languages, they impose additional external dependencies on TraceLab and the development environment. Moreover, TraceLab has no

[12] http://www.ikvm.net/
[13] https://rdotnet.codeplex.com/

built-in mechanism for recognizing components written in R. To overcome this issue, we have created a lightweight language plugin for R (named RPlugin) that allows R scripts to be run from TraceLab. The component classes are written normally in .NET, and any R scripts that need to be run interface with the plugin. RPlugin makes calls to an existing implementation of R and has a framework for passing data and running scripts in R. RPlugin is included with the TraceLab Component Library described section 5.4.

The developers of TraceLab have created a Matlab plugin similar to RPlugin that can run Matlab scripts from .NET.

### 5.2.2 Comparing TraceLab with Other Tools

There are many other frameworks and tools that have been designed to support research in other domains, such as information retrieval, machine learning, data mining, and natural language processing, among others. Consequently, reuse of third party tools or APIs is a common practice for constructing experiments and building research infrastructure in software evolution and maintenance. For example, a common scenario is to reuse WEKA for implementations of machine learning classifiers, R for statistical analysis, or MALLET for topic modeling. However, these tools/APIs were not built to support research on software evolution and maintenance. Moreover, most of the tools were conceived as extensible APIs and only few of them provide features such as experiment composition by using a data-flow GUI, new components implementation, or easy sharing/publishing of experiments; moreover, not all of them can be used across multiple platforms. Table 5-1 compares TraceLab to some similar tools that also use a data-flow oriented GUI.

The R Project [152] is a programming language and environment designed to perform statistical computing tasks on large-scale data. The tool is primarily command based, with the ability to produce charts and graphs. There are a multitude of user-contributed libraries for

performing specialized tasks, including a variety of common software engineering research tasks. However, R does not feature an environment for designing experiments using a GUI-based workflow and can be difficult to reproduce when shared due to dependencies on a large number of libraries and different versions. Additionally, researchers must learn a new programming language when performing experiments in R.

Matlab [116] offers a programming language in an interactive environment geared towards numerical computation, data analysis, visualization (e.g., 2D and 3D visualization) and programming. Matlab was designed to be used in diverse areas, ranging from signal processing, image processing, testing and measuring, computational finance and many others.

Simulink [117] is a Matlab-based tool for simulation and model-based design of embedded systems. In Simulink, a model is composed of subsystems (*i.e.*, a group of blocks) or individual blocks, and the blocks can be implemented using Matlab, C/C++, or Fortran. Building the model can be accomplished using drag-and-drop of blocks and making connections between them, which is similar to the way TraceLab allows to build experiments.

WEKA [180] is a collection of machine learning algorithms that are packaged as an open source Java library that also allows running the algorithms using a graphical user interface (GUI). One of the WEKA modules is the *KnowledgeFlow*, which provides the user with a data-flow oriented GUI for designing experiments. As in TraceLab, the components in the KnowledgeFlow are categorized by tasks (DataSources, DataSinks, Filters, Classifiers, Clusterers, Associations, Evaluation, Visualization), and there is a layout canvas for designing experiments by dragging, dropping, and connecting components. New components can be added to WEKA by extending or modifying the library using Java, and the experiments can be saved and loaded for being executed in the WEKA Experimenter module.

Table 5-1 Comparison of TraceLab with other related tools (columns). The features (rows) are as follows: 1) data-flow oriented GUI [Yes / No]; 2) Type of application [Desktop / Web / API]; 3) License type [Commercial / Open source / Free online access]; 4) Tool allows saving and loading experiments [Yes / No]; 5) Tool allows creating composite components [Yes / No / Programmatically]; 6) Tool has a component "market" where developers can contribute with their own components [Yes / No]; 7) Programming language that can be used to build new components; 8) The platforms were the tool could be used [Software As A Service, Windows, Linux, Mac]

| Tool / Feature | R Project | Matlab | Simulink | Weka/ Rapid. Miner | Gate | Kepler | FETCH | Taverna | TraceLab |
|---|---|---|---|---|---|---|---|---|---|
| **GUI** | N | Y | Y | Y | N | Y | N | Y | **Y** |
| **Type** | API, D | API, D | D | API, D | API, D | API, D | D | D, W | **API, D** |
| **License** | O | C | C | O | O | O | O | O | **O** |
| **Save/Load exp.** | Y | Y | Y | Y | Y | Y | N | Y | **Y** |
| **Composite component** | P | P | Y | N | P | Y | N | Y | **Y** |
| **Component Market** | Y | Y | Y | N | Y | Y | N | Y | **Y** |
| **Programming Language** | R | Matlab<br>Java<br>C | C/C++<br>Matlab<br>.<br>Fortran | Java | Java | R<br>C<br>Matlab<br>Java | RML<br>(Other) | Any | **Java<br>R<br>.NET<br>(e.g., C#)<br>Matlab** |
| **Platforms** | W, L, M | W, L, M | W, L, M | W, L, M | W, L, M | W, L, M | W, L | W, L, M | **W, L, M** |

RapidMiner [153] is a data mining application that provides an improved GUI for designing and running experiments. It includes a reusable library for designing experiments and running them and it fully integrates WEKA as the machine learning library.

GATE [166] provides an environment for text processing that includes an IDE with components for language processing, a web application for collaborative annotation of document collections, a Java library, and a cloud solution for large scale text processing.

Kepler [92] is a tool that follows the same philosophy as TraceLab. By using Kepler, it is possible to build, save, and publish experiments/components using a data-flow oriented GUI. It is also possible to extend Kepler because of its collaborative-project nature. However,

Table 5-2 List of Journals and Conferences for which we identified at least one paper in our mapping study

| Abbreviation | Venue Name | # papers |
|---|---|---|
| ASE | Automated Software Engineering | 1 |
| CSMR | European Conference on Software Maintenance and Reengineering | 2 |
| EMSE | Empirical Software Engineering | 1 |
| ICSE | International Conference on Software Engineering | 6 |
| ICPC | International Conference on Program Comprehension | 9 |
| ICSM | International Conference on Software Maintenance | 3 |
| MSR | Working Conference on Mining Software Repositories | 2 |
| TEFSE | International Workshop on Traceability in Emerging Forms of Software Engineering | 1 |
| TSE | Transactions in Software Engineering | 1 |
| WCRE | Working Conference on Reverse Engineering | 1 |
| | **Total** | 27 |

the main difference with TraceLab is that Kepler was conceived as a tool for experiments in sciences such as Math or Physics.

FETCH [62] is a set of third-party open source tools linked in a pipeline to support program analysis of large C/C++/Java software systems. Fetch does not allow researchers to design experiments or extend components. Instead, it is a command-line based tool that applies several analyses to a software system and generate reports (i.e., charts, tables, files) describing the results of the analysis.

Taverna [173] is a workflow-based tool for designing experiments, by connecting components deployed as web services. The components are imported into Taverna through the web service's WSDL (Web Service Description Language). Therefore, Taverna is independent of the programming language, and researchers have to write their components on any language and publish them as a discoverable web service. However, Taverna does not provide an IDE for implementing/publishing web services. Taverna has a workbench for designing the workflows, a server for the remote execution of workflows when required, and a command line tool for workflow execution from a terminal.

Although TraceLab is not specialized on simulation, natural language processing, or machine learning, it was specifically designed to allow software engineering and maintenance researchers the possibility to (i) *develop* and *share* their own components/experiments, and (ii) to ensure the *reproducibility* of their results. During the design of TraceLab, a traditional desktop application was preferred over a Service-Oriented Architecture solution because we believed that the overhead of services (i) may not suit the kinds of experiments that might be conducted in SE, and (ii) could have introduced an additional burden for the user to create and maintain services. Furthermore, service composition adds additional overheads, which are not suited to some experiments in the traceability domain as well as other SE domains. Hence, our goal was to create a local solution (to avoid confidentiality issues) to allow users to quickly and easily create and compose components. Although TraceLab was initially implemented in C# and supported only Windows, the latest version of TraceLab is cross-platform and supports all major OS platforms (*e.g.*, Window, MacOS and Linux). However, we made publicly available on GitHub the Windows version of TraceLab, and we will open-source the cross-platform version of TraceLab after it goes through the incubation stage. In order to achieve this support for all OS platforms, TraceLab was compiled using Mono[14], the open source, cross platform .NET framework.

To implement the TraceLab components, researchers can use Java, any .NET language (*e.g.*, C#, VB, C++), R or Matlab. For the .NET languages, either Microsoft Visual Studio or Mono can be used.

## 5.3 Mapping Study of Software Maintenance Techniques

In this section we present the methodology, analysis, and results of a mapping study [93] aimed at identifying a set of techniques from particular areas of SM, which could be

[14] www.mono-project.com/

implemented as TraceLab experiments in order to constitute an initial practical body of knowledge that would benefit the SM research community. Moreover, these identified techniques were reverse engineered into basic modules that we implemented as TraceLab components, in order to generate a *Component Development Kit* (see Section 0) and a *Component Library* (see Section 5.4.2) that serves as a starting point for any interested researcher to implement new techniques or build upon existing ones.

For our study, we used the systematic mapping process described by Petersen *et al.* [136]. The process consists of five stages: 1) defining the research questions of the study, 2) searching for papers in different venues, 3) screening the papers based on inclusion and exclusion criteria in order to find relevant ones, 4) classifying the papers, and 5) extracting data and then generating the systematic map.

A mapping study is different from a systematic literature review in that literature reviews aim to answer a specific research question by extracting and analyzing the results of primary studies [93], for example, a review of studies analyzing development effort estimation techniques to see which ones work the best [86]. In contrast, mapping studies attempt to address more abstract research topics by classifying the methodologies and findings into general categories. Mapping studies are useful to the research community in that they provide an overview of trends within the search space [136]. Furthermore, they may be used as a starting point by researchers looking to improve the field by describing common methodologies and perhaps discovering untapped areas that others have missed.

### 5.3.1 Defining the Research Question

Our goal is to identify a set of representative techniques from specific areas of SM, and then use them to generate TraceLab components and experiments to accelerate and support

research in SM. Thus, we defined the following research questions (RQs) for the mapping study:

$RQ_1$: What types of techniques are common to experiments in software evolution and maintenance research?

$RQ_2$. What individual techniques are used across many SM experiments?

$RQ_3$. How do experiments in SM research differ across different sub-domains?

$RQ_1$ attempts to identify high-level categories containing groups of techniques designed to perform similar research tasks. $RQ_2$ focuses on individual techniques and aims to identify the most common techniques used in experiments in the mapping study. $RQ_3$ is intended to compare and contrast how techniques are used in different high-level research tasks, such as traceability link recovery or feature location.

These three RQs can be reformulated into a single main research question as follows:

*RQ: Which SM techniques are suitable to form an initial actionable body of knowledge that other researchers could benefit from?*

In particular, we focused on a subset of SM areas in which Bogdan Dit, Evan Moritz, Mario Linares-Vásquez and Denys Poshyvanyk (see Section 1.2) have expertise. This allowed us to generate an initial and extensible body of knowledge that could support the research community. The SM research community, can contribute to the body of knowledge by continually adding new techniques and components.

### 5.3.2 Conducting the Search

In order to find these techniques, we narrowed the search space to the publications from the last ten years of a subset of journals and software engineering conferences. In addition, in our search we incorporated the "snowballing" discovery technique (*i.e.*, following references in the related work) discussed by Kitchenham *et al.* [93]. The list of journals and conferences

from which we selected at least one paper in our mapping study is presented in Table 5-2. Additional information can be found in our online appendix.

### 5.3.3 Screening Criteria

The primary *inclusion* criterion consisted of identifying whether the research paper described a technique that addressed one of the following maintenance tasks: traceability link recovery, feature location, program comprehension and duplicate bug report identification. In most cases, this information was determined by reading the title, abstract, keywords, and if necessary the introduction and the conclusion of the investigated paper.

The *exclusion* criteria were as follows. First, we discarded techniques that could not have been implemented effectively in TraceLab due to various reasons, such as (i) lack of sufficient implementation details, (ii) lack of tool availability or (iii) the technique was not fully automated, and would require interaction with the user. Second, we did not implement complex techniques that would have required a lengthy development time, or techniques that are outside the expertise of the authors. Third, we discarded techniques with numerous dependencies to deprecated libraries or other techniques, as our goal was to implement the most popular techniques that can be incorporated or built upon.

### 5.3.4 Classification

In our mapping study we used two independent levels of classification. The first one consisted of categorizing the papers based on the SM task (*e.g.*, traceability link recovery, feature location, program comprehension and detecting duplicate bug reports) they presented (see Section 5.3.3). This classification was targeted at answering $RQ_3$.

The second level of classification was identifying common functionality between the basic building blocks used in an approach (*e.g.*, all the functionality related to identifier

splitting, stemming, stopword removal and others, were grouped under "preprocessing"). This level of classification is necessary for answering $RQ_1$ and $RQ_2$.

### 5.3.5 Data extraction

The list of papers that we identified in our study is presented in Table 5-3 in the first column along with the Google Scholar citation count as of December 1, 2013 (second column). The papers are grouped by the primary SM tasks they address, and are sorted chronologically.

The remaining columns constitute the individual building blocks and components we identified in each approach, grouped by their common functionality. A checkmark (✓) denotes that we implemented the component in the *CL*. An *X* denotes that the code related to the components appears in the approach, but is not implemented in the *CL* at this time (see Section 5.4.2 and Section 5.7).

Table 5-3 shows only a subset of the information. For the complete information, we refer the interested reader to our online appendix.

## 5.4 Component Library and Development Kit

From the 27 papers identified in the mapping study, we reverse engineered their techniques in order to create a comprehensive library of components and techniques with the aim of providing the necessary functionality that SM researchers would need to reproduce experiments and create new techniques.

This process resulted in generating (i) a ***C**omponent **D**evelopment **K**it* (*CDK*) that contains the implementation of all the SM techniques from the study, (ii) a ***C**omponent **L**ibrary* (*CL*) that adapts the *CDK* components to be used in TraceLab and (iii) the associated documentation and usage examples for each.

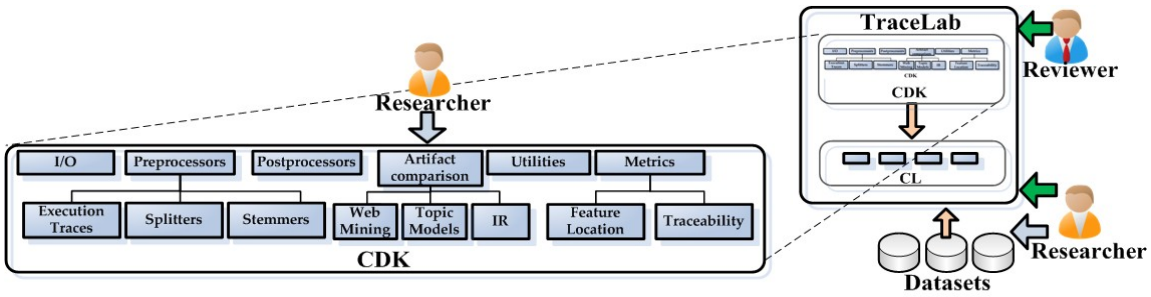



Figure 5-5 Diagram of the hierarchy of the *CDK* in the context of TraceLab. *CDK* and *CL* are part of TraceLab. Researchers can contribute to the *CDK* and the datasets (gray arrow), and reviewers and researchers (green arrow) can use TraceLab to verify details of existing experiments

### 5.4.1 Component Development Kit

The *Component Development Kit* (*CDK*) is a multi-tiered library of common tools and techniques used in SM research. These tools are organized in a well-defined hierarchical structure and exposed through a public API. The intent of this compilation is to aid researchers in reproducing existing approaches and creating new techniques for *software maintenance and evolution*. Therefore, the appropriate name for this CDK would be *Component Development Kit-Software Maintenance and Evolution (CDK-SME)*, in order to distinguish it from other CDKs (e.g., related to *requirements engineering*) that will be developed in the near future by other groups in the research community surrounding TraceLab. However, for brevity, and because in this chapter we are only discussing the CDK for software maintenance and evolution, throughout this chapter we will refer to *CDK-SME* as *CDK*.

By providing access to tools and techniques related to software maintenance and evolution tasks, the *Component Development Kit* facilitates the research evaluation process, and researchers no longer have to start from scratch or spend time adapting their pre-existing tools to a new project. Furthermore, researchers can use combinations of these tools to create new techniques and drive new research.

At the top level, the *CDK* is separated into categories of high-level tasks, such as I/O, preprocessing techniques, artifact comparison techniques, and metrics calculations (see Figure

5-5). Those categories are then further broken down as needed into more specific tasks. This design aids technique developers in locating relevant functionality quickly and easily, as well as providing base points for integrating new functionality in the future. The high-level categories are described as follows:

*Data preprocessing* techniques primarily convert the raw data into a different form that will be used in other steps of the approach. For text-based approaches, preprocessing typically involves extracting comments and identifiers from source code, removing stop-words, stemming, as well as other methods for text manipulation. For structural approaches, preprocessing could involve parsing an execution trace or calculating a static dependency graph. The preprocessing techniques are usually the first to run, before any other steps of the approach.

*Artifacts comparison* techniques encapsulate all the techniques that implement any kind of comparison between software artifacts to determine relationships between them. These techniques usually take in a set of software artifacts (such as source code or requirements documents) as input and produce a set of suggested relationships between documents. These suggestions may include a confidence score, textual similarity scores, etc., which are useful for ranking the set of input artifacts based on various criteria.

The *metrics* category encapsulates the measures by which an approach is evaluated. These are used to determine the accuracy (or performance) of a technique and to compare it with other techniques. Examples of such metrics include precision, recall, effectiveness, etc.

Guided by findings from the mapping study, we evaluated each technique based on coverage, usefulness, and perceived difficulty and effort in implementation. In addition to our design goals of providing a clean and easy to use API, another goal was to minimize the number of external dependencies necessary to implement the technique. As such, some techniques that have numerous external dependencies were left out.

### 5.4.2 Component Library

The *Component Library* (*CL*) is comprised of metadata and wrapper classes registering certain functionality as components in TraceLab. It acts as a layer in between TraceLab and the *CDK*, adapting the functionality of the *CDK* to be used within TraceLab. A typical component will import data from TraceLab's data sharing interface (the *Workspace*), call various functions on the data using the *CDK*, and then store the results back to the *Workspace*.

To register a component in TraceLab, a class must inherit from the `BaseComponent` abstract class defined in the TraceLab SDK. All components must override the `Compute()` method which contains the desired functionality of the component within the context of a TraceLab experiment. Component classes may also override `PreCompute()` and `PostCompute()` to pre-allocate and dispose of resources (these methods are called immediately before and after the `Compute()` method).

Furthermore, all components have a component declaration attribute (or annotation in Java terminology) that describes information about the component. For example, the `[Component]` attribute specifies information about the component's name, description, author, version, and optional configuration object. The configuration object is responsible for all the settings associated with the component (e.g., weighting schema for Vector Space Model, input path for a component loading a corpus, etc.). Any inputs and outputs from and to the workspace must be declared with individual `[IOSpec]` attributes describing the input or output name and data type. Lastly, components may optionally declare `[Tag]` attributes for automatic categorization in the component library. The declaration of all these attributes serve two purposes, namely to (i) allow the class to be registered in TraceLab as a component, and (ii) to ensure that a component can only be connected with a compatible component. Figure 5-4 shows an example of a component that used three `[IOSpec]` attributes (two for input and one for output) to define the name (e.g., *TargetArtifacts*, *SourceArtifacts* and *Similarities*) and

data type (e.g., `TLArtifactsCollection` and `TLSimilarityMatrix`) of inputs and outputs. In addition, the metadata specific to the component, which was defined using the `[Component]` attribute, is presented in the lower right corner of the figure.

After compiling, libraries containing components should be placed in a registered component directory, in order to allow TraceLab to recognize them and make them available in the Component Library (see Figure 5-2 (1)). These directories are defined in TraceLab's settings menu and user-defined directories can be added or removed as needed.

In addition to custom components, the TraceLab SDK allows users to define custom data types. These user-defined data types must declare a `[WorkspaceType]` attribute in order for TraceLab to recognize them as workspace types that can be used in the workspace. These types must also declare a `[Serializable]` attribute to allow data to be transferred between the workspace, components, and disk. It is important to note that any custom data types that do not need to be used in the workspace (e.g., intermediate data used within a component) do not need to be registered with TraceLab. Libraries containing types must also be placed in a registered directory containing types, which is usually separate from the components library. Workspace types may also provide custom visualizations for inspecting the data after an experiment has run.

The *Component Library* uses the same structure as the *CDK* (see Figure 5-5), providing a mapping from TraceLab to the *CDK*. Components can be organized in TraceLab through the use of hierarchically organized developer and user *Tags*, another feature of the TraceLab SDK. Components are grouped via Tags into the same high-level tasks as the *CDK*.

From the building blocks of the *CDK* identified in the mapping study, we implemented 25 out of 51 as TraceLab components. In many cases, this was done as a one-to-one mapping from the *CDK* to the *CL*. However, some techniques could be broken down into more general

Table 5-3 Mapping study results (first column) and implementation of these techniques in the *CDK* (✓ means that the component from the first row is implemented in *CDK* and *X* means is not yet implemented in *CDK*)

| Technique<br>Year / Venue / Name / Ref | Google Scholar Citation Count | Preprocessing | | | | | | | | | | | Artifact Comparison | | | | | | Metrics | | | Postprocessing | | | Other |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Bag-of-words tokenizer | Stopwords Remover | Porter stemmer | CamelCase splitter | Execution trace logger | Dependency Graph Generator | Samurai splitter | smoothing filter | Snowball Stemmer | Part-of-speech tagger | | Latent Semantic Indexing | Vector Space Model | Latent Dirichlet Allocation | Jensen-Shannon divergence | Relational Topic Model | HITS and PageRank | Precision / Recall metrics | Effectiveness Measure | Principal Component Analysis | Execution trace extractor | Affine transformation | O-CSTI and UD-CSTI | Genetic Algorithm |
| **Traceability Link Recovery** | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2008.ICPC.Abadi [2] | 58 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | | ✓ | ✓ | . | ✓ | . | . | ✓ | . | . | . | . | . | . |
| 2009.ICPC.Capobianco [24] | 26 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | ✓ | | ✓ | . | . | ✓ | . | . | ✓ | . | . | . | . | . | . |
| 2010.ICPC.Oliveto [132] | 71 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | | ✓ | ✓ | ✓ | ✓ | . | . | ✓ | . | ✓ | . | . | . | . |
| 2010.ICSE.Asuncion [9] | 87 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | | ✓ | . | ✓ | . | . | . | ✓ | . | . | . | . | . | . |
| 2011.ICPC.DeLucia [38] | 15 | ✓ | ✓ | ✓ | ✓ | . | . | . | ✓ | . | . | | ✓ | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . |
| 2011.ICSE.Chen [27] | 5 | ✓ | . | . | . | . | . | . | . | . | . | | . | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . |
| 2011.ICSM.Gethers [65] | 30 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | | . | ✓ | . | ✓ | ✓ | . | ✓ | . | ✓ | . | ✓ | . | . |
| 2013.CSMR.Panichella [135] | 5 | ✓ | ✓ | . | . | . | ✓ | . | . | . | . | | . | ✓ | . | ✓ | . | . | ✓ | . | . | . | . | ✓ | . |
| 2013.ICSE.Panichella [134] | 10 | ✓ | . | . | . | . | . | . | . | . | . | | . | . | ✓ | . | . | . | ✓ | ✓ | . | . | . | . | ✓ |
| 2013.TEFSE.Dit [50] | 3 | ✓ | . | . | . | . | . | . | . | . | . | | . | . | ✓ | . | . | . | ✓ | . | . | . | . | . | ✓ |
| **Feature Location** | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2004.WCRE.Marcus [115] | 277 | ✓ | . | . | ✓ | . | . | . | . | . | . | | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . | . |
| 2007.ASE.Liu [105] | 104 | ✓ | ✓ | . | ✓ | X | . | . | . | . | . | | ✓ | . | . | . | . | . | . | ✓ | . | ✓ | ✓ | . | . |
| 2007.TSE.Poshyvanyk [143] | 212 | ✓ | ✓ | . | ✓ | . | . | . | . | . | . | | ✓ | . | . | . | . | . | . | ✓ | . | . | ✓ | . | . |
| 2009.ICPC.Revelle [157] | 37 | ✓ | . | . | . | X | ✓ | . | . | . | . | | ✓ | . | . | . | . | . | . | . | . | ✓ | . | . | . |
| 2009.ICSM.Gay [63] | 48 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | | . | ✓ | . | . | . | . | . | . | . | . | . | . | . |
| 2011.ICPC.Dit [45] | 29 | ✓ | ✓ | ✓ | ✓ | X | . | X | . | . | . | | ✓ | . | . | . | . | . | . | ✓ | . | ✓ | . | . | . |
| 2011.ICPC.Scanniello [164] | 13 | ✓ | ✓ | ✓ | ✓ | . | ✓ | . | . | . | . | | . | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . |
| 2011.ICSM.Wiese [184] | 6 | ✓ | . | ✓ | . | . | . | . | . | ✓ | . | | . | ✓ | . | . | . | . | . | . | . | . | . | . | . |
| 2012.ICPC.Dit [49] | 11 | ✓ | ✓ | ✓ | ✓ | X | . | . | . | . | . | | ✓ | ✓ | . | . | . | . | . | ✓ | . | ✓ | . | . | . |
| 2013.EMSE.Dit [53] | 9 | ✓ | ✓ | ✓ | ✓ | X | ✓ | . | . | . | . | | ✓ | . | . | . | . | ✓ | . | ✓ | . | ✓ | . | . | . |
| **Program Comprehension** | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2009.MSR.Enslen [60] | 65 | ✓ | . | . | ✓ | . | . | X | . | . | . | | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 2009.MSR.Tian [178] | 38 | ✓ | ✓ | . | ✓ | . | . | . | . | . | . | | . | . | ✓ | . | . | . | ✓ | . | . | . | . | . | . |
| 2010.ICSE.Haiduc [76] | 32 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | | ✓ | . | . | . | . | . | . | . | . | . | . | . | . |
| 2012.ICPC.DeLucia [39] | 7 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | . | . | . |
| **Identify Duplicate Bug Rep.** | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2007.ICSE.Runeson [159] | 185 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | | . | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . |
| 2008.ICSE.Wang [182] | 188 | ✓ | ✓ | ✓ | . | X | . | . | . | . | . | | . | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . |
| 2012.CSMR.Kaushik [90] | 9 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | | ✓ | ✓ | ✓ | . | . | . | ✓ | . | . | . | . | . | . |
| Total: | 1,580 | 27 | 20 | 17 | 14 | 6 | 4 | 2 | 1 | 1 | 1 | | 14 | 14 | 7 | 5 | 1 | 1 | 14 | 7 | 2 | 5 | 3 | 1 | 2 |

ones, which were desirable for component re-use. For example, the Vector Space Model is a straightforward technique, but there can be many variations on its implementation (see Section 0). We implemented a few weighting schemes (*e.g.*, binary term frequency, tf-idf, and log-idf) and similarity functions (*e.g.*, cosine, Jaccard), so that a component developer could pick and choose from the desired schemes.

Another example is the precision and recall metrics in traceability link recovery. Although this component consists of only one column in the mapping study, the *CDK* covers many of the commonly used metrics in the literature (*e.g.*, precision, recall, average precision, mean average precision, F-measure, and precision-recall curves). Component developers could choose from any of these measures in their experiments.

### 5.4.3 Documentation

Documentation of the *CDK* and *CL* plays a key role in assisting researchers and component developers new to TraceLab. In addition to code examples and API references, documentation provides vital information about a program's functionality, design, and intended use. This adds a wealth of knowledge to someone who wants to use TraceLab and start designing new experiments from components. We provide this information in a wiki format on our website[15,16], which includes a developer guide, the CDK API reference, release notes, and code examples.

### 5.4.4 Extending the CDK and CL

The *CL* and *CDK* themselves are not the definitive collection of all the SM tools that researchers will ever need. However, their design and implementation in conjunction with TraceLab's framework provide a foundation for extending SM research in the future.

Both the *CL* and *CDK* are released under an Open Source license (GPL) in order to facilitate collaboration and community contribution. As new techniques are invented, they can be added to the existing hierarchy and thus into TraceLab.

[15] http://coest.org/coest-projects/projects/semeru/wiki
[16] https://github.com/CoEST

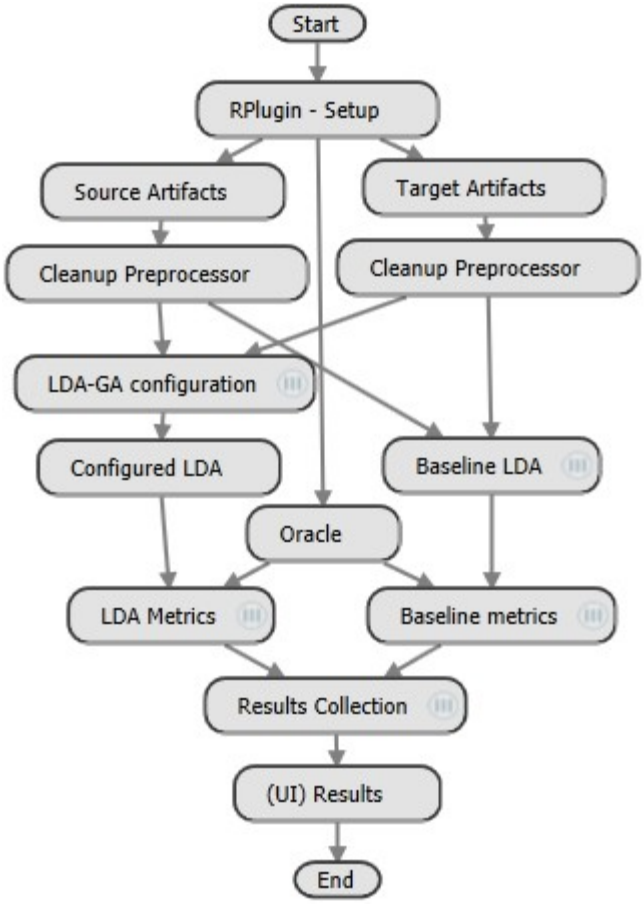



Figure 5-6 TraceLab experiment described in Section 3.3.1, which evaluates the performance of LDA-GA on the EasyClinic dataset and compares it with the baseline [132]

In creating the *CL* and *CDK*, we leveraged TraceLab's ability to modify existing components or to create custom (i.e., user made) components that will fit the need of a researcher, through the TraceLab SDK. Researchers can also create adapt or modify existing datatypes or create new ones if needed. It is important to know that multiple versions of the same datatypes can exist, but once a particular version of a datatype or component is referenced in a particular experiment, that version of the datatype or component will be exported and shared with the community (using TraceLab's packaging feature – see Section 0), to ensure the reproducibility of an experiment even in the case of having multiple versions of the same datatype or component. As the body of SM techniques grows, researchers can utilize our components and extend them to new ones via the same process. Part of our future work will be

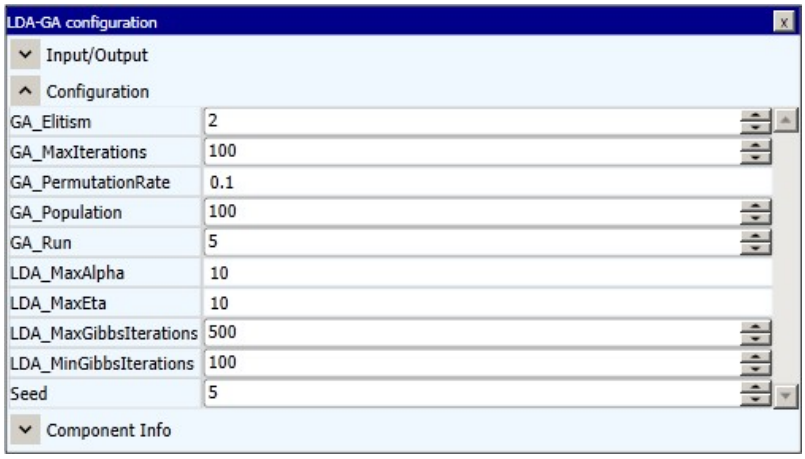



Figure 5-7 Setup window for the *LDA-GA Configuration* component for the experiment in Figure 5-6

dedicated to ensuring that existing components would be easier to discover and understand by other researchers (e.g., through proper documentation), to encourage reuse and reduce the number of overlapping components. Moreover, we will focus on establishing a process of incorporating user-made components into the *CL* and *CDK*, by establishing a standard of quality that each proposed component must satisfy.

## 5.5 Reproducing LDA-GA and IR-GA Experiments

This section presents the details of reproducing an experiment using the *CDK* and the *CL* proposed in this chapter. The reproduced experiment is the one described in Section 3.3.1, which evaluates the performance of LDA-GA on the EasyClinic dataset and compares it with the baseline [132]. The original results of this experiment are presented in Section 3.4.1.

The reproduced experiment (see Figure 5-6) is instantiated using a set of components included in the *CL*. Since part of this experiment is implemented as an R script, the *RPlugin - Setup* component allows configuring the R environment. The data is loaded using the *Source Artifacts* and *Target Artifacts* components and preprocessed using the *Cleanup Preprocessor* component. The *LDA-GA configuration* component allows configuring specific parameters for LDA-GA, as illustrated in Figure 5-7. For example, we can setup the search space for the LDA

parameters $\alpha$, $\beta$, number of iterations and number of topics, and we can also setup specific GA parameters (e.g., population size, mutation rate, elitism, etc.). The results of LDA-GA will be stored in the *Configured LDA* component and they will be compared against the *Baseline LDA*. Using the *Oracle*, the precision and recall values for LDA-GA and the baseline will be computed using the *LDA Metrics* and *Baseline metrics* components, which in turn will be compared using the *Results Collection* component and presented as a graph (see Figure 5-8) using the *(UI) Results* component.

Figure 5-8 shows the precision and recall curve for the Experiment results (i.e., LDA-GA) and the baseline [132]. These results are the same as the ones presented in Figure 3-3 (a), which were originally computed without TraceLab.

This section showed one example of using the *CL* to reproduce an LDA-GA experiment in TraceLab. The existing components from the *CDK* and *CL* (see Table 5-3) can be used to reproduce LDA-GA experiments for other datasets or to reproduce IR-GA experiments.

## 5.6 TraceLab: Alternative Uses

The community surrounding *TraceLab*, the *Component Development Kit* and the *Component Library* enables the replication of experiments, helps new researchers to become productive more quickly, and encourages innovation through equipping researchers with the means to synthesize techniques and to rigorously explore and evaluate new ideas across different domain. In addition, we argue that the curriculum for software engineering classes could be positively impacted as well.

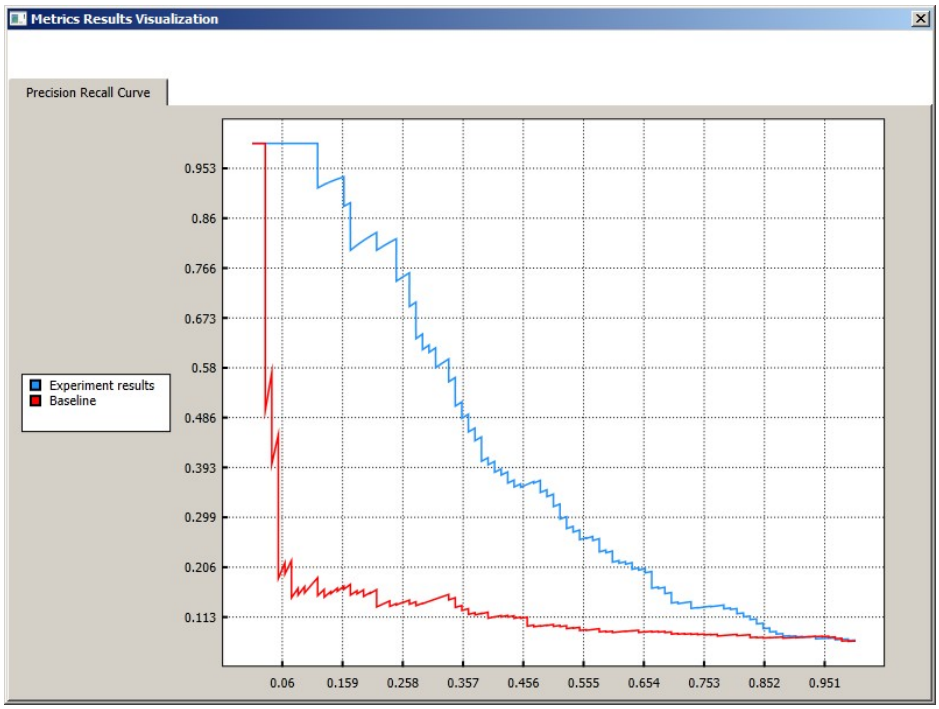



Figure 5-8 Precision and Recall curves for Experiment Results (LDA-GA) and Baseline [132] that was generated by executing the TraceLab experiment presented in Figure 5-6. These results are the same as the ones presented in Figure 3-3 (a), which were originally computed without TraceLab

First, class assignments and projects could be designed to support their submission using TraceLab. Students can be given a partial TraceLab experiment (e.g., loading data, saving data), and could be asked to implement one or more components that compute the results for a technique that supports a maintenance task. Alternatively, students can be given a complete experiment (which implements a technique for traceability link recovery, feature location, or any other maintenance tasks) and asked to improve on the existing techniques by writing new components, and compare the results to the original technique (which can be used as a baseline). The benefit of using TraceLab is that (i) it provides students the flexibility to implement code in some of the most popular languages (Java, C++, C#) on any OS (e.g., Windows, Linux, Mac), and it provides the instructor with a uniform method for evaluating and grading the assignments or projects (i.e., the instructor has to run an experiment that will be used as a ground truth alongside the experiment submitted by the student).

Second, class assignments or projects can be assigned to students as part of a customized *Challenge*, using *TraceLab's challenge* feature. Instructors can define the challenge (i.e., the problem), provide the necessary datasets, metrics, an experimental harness (in the form of a TraceLab experiment) in which student's solutions can be "plugged", and ask students to implement their solutions and submit them to an online portal supporting leaderboards, where challenge results will be compared and ranked. In the research community, one of these TraceLab Challenges made its debut during the 7th Traceability of Emerging Forms of Software Engineering (TEFSE) workshop in 2013, and will continue for the TEFSE workshops in 2015 and 2017.

## 5.7 Limitations

This section discusses some potential limitations for conducting research using TraceLab, the *Component Development Kit* and the *Component Library*.

For example, the current infrastructure does not support collecting metrics (e.g., LOC, cyclomatic complexity, depth of inheritance tree, coupling between objects, etc.) from different software systems. However, these metrics could be computed with external tools and imported into TraceLab and used in experiments.

Running experiments in TraceLab from code hosted in a .NET process is in general slower than running the code associated with experiments natively. For example, for typical experiments the code would require a few seconds more to run on TraceLab, but for computationally expensive experiments (which could take hours, or days on native code) TraceLab experiments could take from minutes to hours longer to run than experiments ran in native code. Therefore, the time or speed factors in evaluating an approach would need to be considered.

We attempted to identify papers that covered a number of topics in SM, which we were familiar with or had expertise with. Within the papers we covered, in some cases we were unable to obtain exact implementations due to lack of specific details or availability of tools. Additionally, many experiments cannot be reproduced directly because the datasets under study were undisclosed or unavailable.

The *CL* and *CDK* do not implement every technique and building block found in the mapping study. The amount of time, manpower, and testing required to do so would be far beyond the resources available. That being said, we tried to implement as many of the techniques that we could in order to show the efficacy and usefulness of TraceLab as a research tool in the domain of software evolution and maintenance. We are continuously working on driving new research projects with TraceLab [3, 29, 44, 50, 78, 101, 155, 183] and encourage others to do so as well.

A major issue that prevented us from using or implementing certain tools was their copyright licensing. In some cases they do not use permissive licenses, and even if the source code was available its license did not permit distribution. TraceLab is released under the open source license GPL, which we follow as well with the *CL* and *CDK*. Developers may release their own components under any license they wish, but if they wish to extend or modify the *CL* or *CDK*, they must also release under GPL.

With any new technology or framework, as is the case with TraceLab and the *CL*/*CDK*, there is an inherent learning curve that needs to be overcome before researchers can take advantage of this infrastructure in order to support their research and contribute to the community as well. To facilitate this learning process and to make it easier for new users to create new experiments and components, we provide numerous online tutorials (e.g., wikis and videos) to help new users get started. In fact, most of our early adopters have viewed a selection of tutorials prior to adoption.

## 5.8 Conclusions

This chapter is an extension of our previous work [47] and addressed the *reproducibility* problem associated with experiments in SM research. Our goal was to support and accelerate research in SE by providing a body of actionable knowledge in the form of reproduced experiments and a *Component Library* and *Component Development Kit* that can be used as the basis to generate novel, and most importantly reproducible techniques.

After conducting a mapping study of SM techniques in the areas of traceability link recovery, feature location, program comprehension and duplicate bug report detection, we identified 27 papers and techniques that we used to generate a library of TraceLab components. We implemented a subset of these techniques as TraceLab experiments to illustrate TraceLab's potential as a research framework and to provide a basis for implementing new techniques.

It is obvious that our effort does not cover the entire range of SM papers or techniques. Therefore, in the future, we are determined to continually expand the TraceLab *Component Library* and *Development Kit* by including more techniques and expanding it to other areas of SM (*e.g.*, impact analysis, developer recommendation, software categorization, etc.). In addition, we are expanding our online tutorials to make it easier for newcomers to get started with TraceLab, and we encourage other researchers to contribute to this body of knowledge for the benefit of conducting reproducible research, which in turn, will benefit the entire research community.

# 6 Conclusions

The dissertation proposes an approach to automatically configure and assemble IR-based solutions in order to support various SE tasks, such as traceability link recovery, feature location, software artifact labeling and detecting duplicate bug reports. Our empirical findings for this work showed that various factors (e.g., preprocessing steps, preprocessing options, IR technique, configuration of IR technique, etc.) that form an IR-based solution have a significant impact on the results when the IR-based solution is applied in the context of a SE task. Moreover, we also observed that the impact of these factors is often considered in isolation in the existing literature, and it is not approached as a whole. Our proposed approach is considering all the factors that can influence an IR-based solution (i.e., all the factors ranging from choosing the preprocessing steps, to choosing and configuring the IR technique with its (near) optimal parameters for the given dataset), and recommends the ones that would produce near optimal results.

The main contributions of this dissertation include:

- We presented the results of an empirical study of two feature location techniques utilizing three different strategies during preprocessing, and more specifically during the process of splitting identifiers. Our main research question was formulated as follows: "*does splitting identifiers have an impact on the accuracy of feature location techniques"*. In order to answer this research question we investigate two feature location techniques (one based on IR and the other one based on the combination of IR and dynamic analysis) on two open-source systems, namely Rhino and jEdit. The results of an extensive empirical evaluation revealed that feature location techniques using IR can

benefit from better splitting algorithms in some circumstances, and that various splitting algorithms can produce results that are statistically significant.

- We present a novel approach that automatically configures the parameters of LDA, a topic modeling IR technique, to support SE tasks. Topic modeling IR techniques have been used to support SE tasks, but these techniques have been used on software artifacts in a similar manner as they were used on natural language documents (i.e., using the same settings and parameters) because the underlying assumption was that source code and natural language documents are similar. However, applying topic models on software data using the same settings as for natural language text produced sub optimal results. Our proposed solution, LDA-GA, automatically configures LDA to achieve improved performance across various SE tasks. LDA-GA uses Genetic Algorithms to determine a near-optimal configuration of LDA's parameters by taking into account the unique characteristics of the input corpus. We evaluated LDA-GA in the context of three different SE tasks (e.g., traceability link recovery, feature location, and software artifact labeling) and the results demonstrate that LDA-GA is able to identify robust LDA configurations, which lead to a higher accuracy and better results on all the datasets for these SE tasks as compared to previously published results (which used "ad-hoc" or "default" values for configuring the LDA parameters).
- We have developed IR-GA, a novel approach that automatically determines and assembles the (near) optimal solution for each stage of assembling and instantiating an IR process that will be used to support a SE task. More specifically, IR-GA considers the task specific components and data sources,

as well as the internal properties of the IR model built from the underlying dataset to determine the specific configuration or parameter to be used for each stage of assembling the IR process. In an extensive empirical study, we applied IR-GA on three different software engineering tasks, namely traceability link recovery, feature location, and identification of duplicate bug reports. The results of the study indicate that IR-GA outperforms approaches previously used in the literature, and that it does not significantly differ from an ideal upper bound that could be achieved by a supervised approach (i.e., one that knows the results a-priori) and combinatorial approach (i.e., one that considers a large number of configurations and parameter combinations).

- We addressed the issue of research reproducibility by providing the details of a framework aimed at supporting it. Research studies in software maintenance are notoriously hard to reproduce due to lack of datasets, tools, implementation details and other factors (e.g., when applying an IR technique to address a SE tasks, details about the exact preprocessing steps and parameters used are missing or are not specified properly). The progress in the field is hindered by the challenge of comparing new techniques against existing ones, as researchers have to devote a large portion of their resources to the tedious and error-prone process of reproducing previously introduced approaches. We addressed the issue of experiment reproducibility in software maintenance and provided a long-term solution towards ensuring that future experiments will be reproducible and extensible. We conducted an initial mapping study of a number of representative maintenance techniques and approaches and implemented them as a set of experiments and a library of components that we

make publicly available with TraceLab, called the Component Library. The goal of these experiments and components is to create a body of actionable knowledge that would facilitate future research and allow the research community to contribute to it as well. Moreover, we have provided all the components required to reproduce the LDA-GA and IR-GA techniques presented in this dissertation.

In summary, our proposed approach considers the interaction of all the factors that can potentially influence the results, and determines which combination of those factors can produce (near) optimal results, as opposed to focusing on improving only one of these factors in isolation. Moreover, our findings indicate that our proposed approaches (also reproducible in TraceLab) can be used to outperform approaches previously used in the literature.

# Appendix A

# Generating Benchmarks for Feature Location

This appendix describes a set of benchmarks from Java applications, which were used in the evaluation of the feature location tasks described in Chapter 2, Chapter 3 and Chapter 4. In addition, these dataset can be used for evaluating other software engineering tasks such as impact analysis, developer recommendation and traceability link recovery. These datasets consist of textual description of change requests and locations in the source code where the change requests were addressed. These datasets are designed for evaluating techniques based on Information Retrieval. In addition, we provide execution traces that were collected based on the description of the change requests. These traces could be used in techniques that combine IR and dynamic information [105]. Some of these datasets were already evaluated in a number of research papers related to feature location [17, 18, 36, 45, 53, 134], impact analysis [64], developer recommendations [102] and traceability link recovery [4]. In addition, we describe in detail the methodology used for generating these datasets from the historical data of the software systems, as well as any limitations associated with the process of generating this data. We also provide a suite of Java tools that instantiate some steps of the methodology, which were used to generate these datasets, and can be used to generate datasets for new software systems. More information can be found in our online appendix [1].

## A.1 Datasets

These datasets contain static, textual, and dynamic information about the software systems, which were generated by analyzing two primary sources of information: (i) issue tracking systems (ITSs) and (ii) source code repositories.

### A.1.1 Glossary of Software Artifacts

**Dataset (or benchmark)**: is a collection of artifacts derived from the ITS and source code repositories and is referred by the name and version of the system (e.g., jEdit 4.3).

**Issue**: is the generic term given to change requests, such as bug reports, feature requests, or any other type of tasks submitted to an ITS (e.g., Bugzilla, Trac, etc.)

**IssueID**: is the ID (i.e., numerical value, such as 123) of an issue, which is automatically assigned by the ITS.

**GoldSet$_{\text{IssueID}}$**: is the set of unique method names that were modified when the issue IssueID was implemented in the system. In other words, it contains the names of the methods that were changed when a bug was fixed, or when a feature was added to the system. The method names in the gold set are fully qualified (i.e., they contain the package name, class name, method name and signature).

**Trace$_{\text{IssueID}}$** (or Execution Trace for IssueID): represents an execution trace that was collected by exercising the scenario presented in the description of the issue IssueID. The execution trace is characterized by a list of methods that were executed when the user attempted to (i) reenact the steps that lead to the buggy behavior described in IssueID or (ii) exercise a feature described in IssueID.

**Marked Trace**: is a trace where the user has control over the beginning and the end of the trace recording process.

**Full Trace**: is an execution trace that records executed methods from the start of the application until the application is closed. Full traces usually capture more information than marked traces.

**Query$_{\text{IssueID}}$**: represents the textual description of the issue IssueID, and consists of the title and description of the IssueID.

Table A-1 Description of the datasets. The columns represent the dataset name (system and version number), the major releases corresponding to the interval for analyzing the SVN data, the number of issues, the type of execution traces (marked or full), the total number of gold set methods in the entire dataset, the number of lines of code, files and methods for the system used to build the corpus

| Dataset | Period | Issues | Trace Type | # Gold Set Methods | KLOC | Files | Methods |
|---|---|---|---|---|---|---|---|
| ArgoUML 0.22 | 0.20-0.22 | 91 | Full | 701 | 149 | 1,439 | 11,000 |
| ArgoUML 0.24 | 0.22-0.24 | 52 | Full | 357 | 155 | 1,480 | 11,464 |
| ArgoUML 0.26.2 | 0.24-0.26.2 | 209 | Full | 1,560 | 186 | 1,752 | 14,597 |
| JabRef 2.6 | 2.0-2.6 | 39 | Full | 280 | 74 | 579 | 4,607 |
| jEdit 4.3 | 4.2-4.3 | 150 | Marked | 748 | 104 | 503 | 6,413 |
| muCommander 0.8.5 | 0.8.0-0.8.5 | 92 | Full | 717 | 77 | 1,069 | 8,187 |

**Corpus**: is a collection of textual documents (e.g., contents of files, classes or methods). For our datasets, we refer to a corpus as the collection of all the method contents for a particular version of the software system.

### A.1.2 Description of the Datasets

The datasets contain in total 633 issues, 633 execution traces and 4,363 gold set methods and are summarized in Table A-1.

The first three datasets (ArgoUML0.22, 0.24 and 0.26.2) are generated from ArgoUML [8], a popular UML editor. The other three datasets (JabRef 2.6 [84], jEdit 4.3 [85] and muCommander 0.8.5 [128]) were generated from JabRef, a manager for BibTeX references, jEdit, a popular text-editor for programmers, and muCommander, a cross-platform file manager. jEdit 4.3 was used in the evaluations from Chapter 2, Chapter 3 and Chapter 4, ArgoUML 0.22 was used in the evaluation from Chapter 3, and JabRef 2.6 was used in the evaluation from Chapter 4.

The columns from Table A-1 are enumerated and described next, and exemplified on the first dataset. The first column represents the name and version of the dataset (e.g., ArgoUML 0.22), which was generated by analyzing the SVN commits of ArgoUML submitted between version 0.20 and 0.22 (see column 2). For this dataset, there were 91 issues identified

(see column 3), which contain a total of 701 gold set methods (see column 5). The type of execution traces collected is full traces (see column 4). Version 0.22 of ArgoUML has 149 KLOC (lines of code) spreading across 1,439 files and 11,000 methods (see columns 6, 7 and 8 respectively).

## A.2 Methodology for generating the datasets

This section describes the methodology used for generating the datasets. The steps are as follows:

### A.2.1 Choose the Software System

The first step consists of choosing a Java software system (e.g., jEdit) with the following characteristics: (i) uses SVN as the source code repository, (ii) has an ITS that keeps track of the change requests, (iii) a subset of SVN log messages are referencing IssueIDs, and optionally, (iv) the system allows collecting execution traces (i.e., the system is not a library that would make it difficult for a user to interact with it in order to collect execution traces). The last requirement is optional, and is only needed for generating datasets that contain dynamic information in the form of execution traces.

Note that the choice of Java systems was restricted by the fact that our tools for (i) generating gold sets, (ii) generating the corpus, and (iii) collecting traces work only with Java systems.

### A.2.2 Choosing the SVN Commits

Choose the period of time between two major releases for the system (e.g., jEdit v4.2 and jEdit v4.3). In the following, we will refer to the earlier version of the system as the *previous release* (e.g., jEdit version 4.2), and to the older version as the *current release* (e.g., jEdit version 4.3).

For each SVN commit submitted between the *previous* and *current release*, we analyzed its log message (see Section 0) and its change set (see Section 0).

### A.2.3 Choosing the Issues

For each SVN Commit, its SVN log message was parsed in order to identify the *IssueID*s. The subset of SVN commits that contained *IssueID*s in their SVN log message (called *SVNCommitsMapped*) were mapped to the issue *IssueID* from the ITS. For example, if SVN commit #123 contained the log message "fix for bug #45678", the issue #45678 (from the ITS) was mapped to the SVN commit #123. We manually verified each mapping to ensure the correctness of the data and to discard SVN commits that contain numbers that do not represent *IssueID*s (e.g., "Eliminated a small code duplication found in r10817", "[...] viewtopic.php?f=4&t=413"). In addition, we also included the cases where an *IssueID* was mapped to multiple SVN commits (i.e., the change request represented by the *IssueID* was implemented across multiple SVN commits).

### A.2.4 Generating the Gold Sets

For each SVN commit from *SVNCommitsMapped* (e.g., #123), we analyzed its associated source code files. More specifically, the version of each modified file (e.g., #123) was compared against the previous version of the file (e.g., #122 or earlier) in order to identify the methods that were modified during the SVN commit. These methods are part of the gold set associated with the IssueID (e.g., #45678) the SVN commit is mapped to (i.e., #123). The details of the tool used for generating these gold sets are presented in Section 0.

### A.2.5 Generating the Corpus

The corpus of the *current release* was generated using the *CorpusGenerator* tool (see Section 0), which parses all the Java files associated with that release and extracts as documents

all the contents associated with a method (i.e., javadoc comments, modifies, type, name, signature and body).

### A.2.6 Generating the Execution Traces

For each issue generated in Section 0, we identified the candidates suitable for generating execution traces on the *current release*. We generated the execution traces by reproducing the scenario presented in the description of the issue. In some cases, the steps to reproduce the bug or feature are enumerated in a straightforward way, whereas in other cases these steps had to be inferred from the description (because they are not explicitly stated). Issues for which we could not collect an execution trace (i.e., the symptoms to reproduce the buggy behavior are not described or cannot be inferred) were discarded. The execution traces were collected using either the Java Platform Debugger Architecture (JPDA) [133] or the Eclipse Test & Performance Tools Platform (TPTP). The traces collected with JPDA (e.g., for jEdit) did not contain any method signatures and they are marked traces. The traces collected using TPTP contained the method signatures and they are full traces. Section 0 discusses the decision of choosing the *current release* for generating the execution traces.

### A.2.7 Cleanup

Not all the issues and gold sets generated in the previous steps became part of the final dataset. Some of the artifacts that did not adhere to a set of standards were discarded. For example, we only kept issues for which their gold sets had at least one method in the corpus of methods, and at least one method in the execution trace.

Methods that appear in the gold set may not necessarily appear in the corpus, due to the inherent process of refactoring that a software system undergoes between two consecutive releases. For example, a method *foo.A.a()* that was modified in an SVN commit (e.g., #123), and appears in the gold set of issue #45678, may not necessarily appear in the corpus, if the

system experienced refactorings, such as the method name was renamed, its signature was changed, the class name was renamed, the class was moved in other packages, or the method was deleted or merged with other methods. Our tools do not automatically keep track of all the changes to the fully qualified name of methods and this is left for future work.

In an initial attempt to address these limitations, we used a simple process, where we manually modified the fully qualified name from the gold set to reflect the name from the corpus. For example, if a large number of methods from the gold set (e.g., *foo.A.a()*, *foo.A.b()*, *foo.A.c()*, *foo.A.d()*, etc.) did not appear in the corpus because the class *foo.A* was renamed to *foo.ARenamed*, we manually renamed the methods in the gold set to *foo.ARenamed.a()*, *foo.ARenamed.b()*, and so on. This manual process was applied only on a handful of gold sets that were identified during quality control of ensuing that at least one gold set method appears in the corpus. We acknowledge that this anecdotal manual process should have been replaced with a more thorough automatic approach, one which keeps track of all the refactorings during two software releases, but this endeavor is left for future work.

## A.3 Tools for Generating the Datasets

This section describes a suite of Java tools that were used for generated these datasets, and these tools could be used by other researchers to generate new datasets, by following the methodology described in Section 0. In addition, we provide Matlab implementations for two IR techniques, namely VSM and LSI.

**DownloadSVNCommits** is a tool based on the SVNKit library, which extracts all the pertinent information related to the SVN commits between the specified *previous* and *current releases*: (i) the SVN log message (which will be parsed for issues) and (ii) the content of the files at SVN revision N and N-1 (these files will be analyzed for extracting the gold set).

**ConvertJPDATraces** and **ConvertTPTPTraces** are two tools that extract the list of methods that were executed for each type of execution trace.

**GoldSetGeneratorFromSVNCommits** uses the Eclipse Abstract Syntax Tree (from Eclipse's Java Development Tools) to automatically generate a list of methods that were changed between two versions of a java file (i.e., the version associated with the current SVN commit and its previous version). The tool only takes into account semantic changes to the code, and does not add to the gold sets methods that experienced formatting changes (e.g., indentation, adding blank lines, formatting comments).

**CorpusGenerator** uses the same underlying technology as *GoldSetGeneratorFromSVNCommits* to generate a corpus consisting of all the methods of a software system. In addition, this tool can also generate corpora for software systems at class or file-level granularity.

**CorpusPreprocessor** preprocesses a corpus produced by *CorpusGenerator*, by eliminating non-literals, splitting identifiers, stop word removal and stemming.

**CorpusConverter** converts a preprocessed corpus generated by *CorpusPreprocessor* to a term by document matrix that can be used as input for IR techniques, such as VSM and LSI.

**VSM** and **LSI** are two Matlab scripts that use VSM and LSI to compute the similarities between a query and the methods of a system (i.e., the corpus).

## A.4 Description of schema of the datasets

This section describes the format of the data. Each dataset contains the following files and folders:

**GoldSets:** a folder with files named *GoldSet[IssueID].txt*. Each file contains the gold set methods, one per line. A gold set method is the fully qualified name of a method (e.g., *foo.A.a(int)*).

**Traces:** a folder with files named *trace[IssueID].trcxml* (TPTP format) or *Trace[IssueID].log* (JPDA format). Each file represents an execution trace collected for issue *[IssueID]*. The online appendix contains more details about the trace format.

**Queries:** a folder where each issue *[IssueID]* has two files named *ShortDescription[IssueID].txt* (i.e., title) and *LongDescription[IssueID].txt* (i.e., the description).

**listOf[IssueType]IssueIDs.txt:** is a file containing the list of IssueIDs for the dataset, one per line. The [IssueType] represents the type (e.g., *bug*, *feature*, *patch*) that was assigned to the issue in the ITS. The IssueIDs correspond to the *[IssueIDs]* from file names from the *GoldSets*, *Traces* and *Queries* folders.

**CorpusMethods-<dataset>.corpusRaw** and **CorpusMethods-<dataset>-AfterSplitStopStem.txt**: are two files containing the un-preprocessed and preprocessed corpora respectively. Each line of these files is a document representing the content of a method.

**CorpusMethods-< dataset >.mapping**: is a file containing the fully qualified names of the methods that have a correspondence in the preprocessed corpus file (i.e., the method name from line *i* corresponds to the method on line *i* from the file *CorpusMethods-<dataset>-AfterSplitStopStem.txt*).

**IssuesToSVNCommitMapping.txt**: is a file containing the *IssueID* and the list of SVN commits that map to it.

## A.5 Discussion

Some of the methods from the gold sets do not have a correspondence in the corpus. This is due to the methodology for generating the data and the refactoring process between two consecutive software releases (see Section 0). In addition, the SVN commits that do not explicitly include in their log messages the IssueIDs they addressed (i.e., the log messages lack the link to the ITS), are not included in the dataset.

In our datasets, we do not exclude from the gold sets the methods that were modified at one point between two releases, which due to subsequent refactorings did not appear in the these releases. This information might be useful for evaluating some tasks that would not require a corpus for the evaluation. Moreover, the solution that requires minimum effort to bypass this discrepancy between the gold set methods and the methods corpus requires filtering the gold set methods from the results, as was done in previous evaluations (see Chapter 2, Chapter 3 and Chapter 4).

Due to the refactorings between two consecutive software releases, some methods may not appear in the *previous release* (e.g., if they were added or renamed) or the *current release* as well (e.g., if they were renamed). We chose the *current release* for generating the corpus and the execution traces because even though the methods that were changed in order to fix the bugs submitted between these releases have similar chances of being present in the *previous release* or *current release* (i.e., due to refactorings), the methods that were added in order to implement the features introduced in the *current release* have zero chance of being present in the *previous release* but have a very high chance of being present in the *current release*. Thus we used one release to capture both the added features and the locations of the methods responsible for the buggy behavior as described in the bug description. If other researchers would require the use of the *previous release* in their evaluation, they could generate the corpus

for the *previous release* using the *CorpusGenerator* tool, and filter from the gold sets the methods that do not appear in that corpus.

The quality of the execution traces might have been impacted by the quality of the steps to reproduce. For some issues, the steps to reproduce the bug or feature are described in an unambiguous way, whereas in other cases the description is open to interpretation. Due to the stochastic nature of the process of manually collecting execution traces, other researchers could generate different traces.

Some of these datasets were used in the evaluations from Chapter 2, Chapter 3 and Chapter 4 and were used to support various software maintenance tasks, such as feature location [17, 18, 36, 45, 53, 134], impact analysis [64], developer recommendations [102] and traceability link recovery [4].

# Bibliography


[1] (2015). *Online Appendix for Bogdan Dit's Dissertation*. Available: http://www.cs.wm.edu/semeru/data/DissertationDit/

[2] A. Abadi, M. Nisenson, and Y. Simionovici, "A Traceability Technique for Specifications," in *16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 103-112.

[3] N. Alhindawi, O. Meqdadi, B. Bartman, and J. I. Maletic, "A tracelab-based solution for identifying traceability links using LSI," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 79-82.

[4] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links," *IEEE Transactions on Software Engineering (TSE),* p. to appear doi: 10.1109/TSE.2012.71, October 2012.

[5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Transactions on Software Engineering (TSE),* vol. 28, pp. 970 - 983, October 2002.

[6] G. Antoniol, Y.-G. Gueheneuc, E. Merlo, and P. Tonella, "Mining the Lexicon Used by Programmers during Software Evolution," in *23rd IEEE International Conference on Software Maintenance (ICSM'07)*, Paris, France, 2007, pp. 14-23.

[7] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," in *Search Based Software Engineering*, ed: Springer, 2011, pp. 33-47.

[8] ArgoUML. (2011, 7/19/2011). *ArgoUML - UML Modeling Tool*. Available: http://argouml.tigris.org/

[9] H. Asuncion, A. Asuncion, and R. Taylor, "Software Traceability with Topic Modeling," in *32nd International Conference on Software Engineering (ICSE'10)*, 2010.

[10] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*: Addison-Wesley, 1999.

[11] P. Baldi, E. Linstead, C. Lopes, and S. Bajracharya, "A Theory of Aspects as Latent Topics," in *23rd ACM SIGPLAN Conference on Object-Oriented Programming,*

*Systems, Languages and Applications (OOPSLA'08)*, Nashville, Tennessee, USA, 2008, pp. 543-562.

[12] E. Barr, C. Bird, E. Hyatt, T. Menzies, and G. Robles, "On the Shoulders of Giants," in *FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10)*, Santa Fe, New Mexico, USA, 2010, pp. 23-28.

[13] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An Empirical Study on the Developers' Perception of Software Coupling," in *35th IEEE/ACM International Conference on Software Engineering (ICSE'13)*, San Francisco, CA, 2013, pp. 692-701.

[14] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 23, p. to appear, 2014.

[15] G. Bavota, R. Oliveto, G. M., D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *IEEE Transactions on Software Engineering (TSE),* p. to appear, 2014.

[16] F. Beck, B. Dit, J. Velasco-Madden, D. Weiskopf, and D. Poshyvanyk, "Rethinking User Interfaces for Feature Location," presented at the 23rd IEEE International Conference on Program Comprehension (ICPC'15), 2015.

[17] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring Latent Dirichlet Allocation based Feature Location," *Empirical Software Engineering (ESE),* pp. to appear, doi: 10.1007/s10664-012-9224-x, August 2012.

[18] L. R. Biggers and N. A. Kraft, "A Comparison of Stemming Algorithms for Text Retrieval Based Feature Location," http://software.eng.ua.edu/reports/SERG-2012-03, 2012.

[19] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To CamelCase or Under_score," in *17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, British Columbia, Canada, 2009, pp. 158-167.

[20] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Development," in *Encyclopedia of Software Engineering*, P. Laplante, Ed., ed: Taylor & Francis LLC, 2010.

[21] D. Binkley and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution," in *Encyclopedia of Software Engineering*, P. Laplante, Ed., ed: Taylor & Francis LLC, 2010.

[22] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research,* vol. 3, pp. 993-1022, 2003.

[23] M. Borg, P. Runeson, and A. Ardö, "Recovering from a Decade: a Systematic Mapping of Information Retrieval Approaches to Software Traceability," *Empirical Software Engineering (EMSE),* pp. 1-52, 2013/05/09 2013.

[24] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in IR-based traceability recovery," in *17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, British Columbia, Canada, 2009, pp. 148 - 157.

[25] B. Caprile and P. Tonella, "Restructuring Program Identifier Names," in *16th IEEE Internation Conference on Software Maintenance (ICSM'00)*, San Jose, California, USA, 2000, pp. 97-107.

[26] C. Caprile and P. Tonella, "Nomen Est Omen: Analyzing the Language of Function Identifiers," in *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, Atlanta, Georgia, USA, 1999, pp. 112-122.

[27] X. Chen, J. Hosking, and J. Grundy, "A Combination Approach for Enhancing Automated Traceability " in *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), NIER Track*, Honolulu, Hawaii, USA, 2011, pp. 912-915.

[28] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, G. O., J. Huffman Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder, "Grand Challenges, Benchmarks, and TraceLab: Developing Infrastructure for the Software Traceability Research Community," in *6th ICSE2011 International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE2011)*, Honolulu, HI, USA, 2011.

[29] J. Cleland-Huang, M. Mirakhorli, A. Czauderna, and M. Wieloch, "Decision-Centric Traceability of architectural concerns," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 5-11.

[30] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czauderna, G. Leach, E. Moritz, M. Gethers, D. Poshyvanyk, J. H. Hayes, and W. Li, "Toward Actionable, Broadly Accessible Contests in Software Engineering," in *34th IEEE/ACM International Conference on Software Engineering (ICSE'12), New Ideas and Emerging Results Track*, Zurich, Switzerland, 2012, pp. 1329-1332.

[31] W. J. Conover, *Practical Nonparametric Statistics*: Third Edition, Wiley, 1998.

[32] T. M. Cover and J. A. Thomas, *Elements of Information Theory*: Wiley-Interscience, 1991.

[33] R. Cummins, "The Evolution and Analysis of Term-Weighting Schemes in Information Retrieval," PhD Thesis, National University of Ireland, 2008.

[34] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison," *Empirical Software Engineering (ESE),* vol. 17, pp. 531-577, 2012.

[35] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk, "Enhancing Software Traceability By Automatically Expanding Corpora With Relevant Documentation," in *29th IEEE International Conference on Software Maintenance (ICSM'13)*, Eindhoven, the Netherlands, 2013, pp. 320-329.

[36] S. Davies, M. Roper, and M. Wood, "Using Bug Report Similarity to Enhance Bug Localisation," in *19th Working Conference on Reverse Engineering (WCRE'12)*, Kingston, Ontario, Canada, 2012, pp. 125-134.

[37] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Applying a Smoothing Filter to Improve IR-based Traceability Recovery Processes: An Empirical Investigation," *Information and Software Technology,* p. to appear, 2012.

[38] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based Traceability Recovery Using Smoothing Filters," in *19th IEEE International Conference on Program Comprehension (ICPC'11)*, Kingston, Ontario, Canada, 2011, pp. 21-30.

[39] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR Methods for Labeling Source Code Artifacts: Is it Worthwhile?," in *20th IEEE International Conference on Program Comprehension (ICPC'12)*, Passau, Germany, 2012, pp. 193-202.

[40] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 16, September 2007.

[41] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, "Information Retrieval Methods for Automated Traceability Recovery," in *Software and Systems Traceability*, A. Zisman, J. Cleland-Huang, and O. Gotel, Eds., ed: Springer Verlag, 2012, pp. 71-98.

[42] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science,* vol. 41, pp. 391-407, 1990.

[43] F. Deissenboeck and M. Pizka "Concise and Consistent Naming," *Software Quality Journal,* vol. 14, pp. 261-282 2006.

[44] A. Dekhtyar and M. Hilton, "Human recoverability index: A TraceLab experiment," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 37-43.

[45] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can Better Identifier Splitting Techniques Help Feature Location?," in *19th IEEE International Conference on Program Comprehension (ICPC'11)*, Kingston, Ontario, Canada, 2011, pp. 11-20.

[46] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, "Generating Benchmarks from Change History Data to Support Evaluation of Software Maintenance Tasks," in *10th Working Conference on Mining Software Repositories (MSR'13), Data Track*, San Francisco, CA, 2013, pp. 131-134.

[47] B. Dit, E. Moritz, M. Linares-Vásquez, and D. Poshyvanyk, "Supporting and Accelerating Reproducible Research in Software Maintenance using TraceLab Component Library," in *29th IEEE International Conference on Software Maintenance (ICSM'13)*, Eindhoven, the Netherlands, 2013, pp. 330-339.

[48] B. Dit, E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, and J. Cleland-Huang, "Supporting and Accelerating Reproducible Empirical Research in Software Evolution and Maintenance using TraceLab Component Library," *Empirical Software Engineering (EMSE),* p. to appear, 2015.

[49] B. Dit, E. Moritz, and D. Poshyvanyk, "A TraceLab-based Solution for Creating, Conducting, and Sharing Feature Location Experiments," in *20th IEEE International Conference on Program Comprehension (ICPC'12)*, Passau, Germany, 2012, pp. 203-208.

[50] B. Dit, A. Panichella, E. Moritz, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Configuring Topic Models for Software Engineering Tasks in TraceLab," in *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, San Francisco, California, 2013, pp. 105-109.

[51] B. Dit, D. Poshyvanyk, and A. Marcus, "Measuring the Semantic Similarity of Comments in Bug Reports," in *1st International Workshop on Semantic Technologies in System Maintenance (STSM'08)*, 2008.

[52] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process (JSEP),* vol. 25, pp. 53–95, January 2013.

[53] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software," *Empirical Software Engineering,* vol. 18, pp. 277-309, April 2013.

[54] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi, "ImpactMiner: A Tool for Change Impact Analysis," in *36th ACM/IEEE International Conference on Software Engineering (ICSE'14) - Formal Demonstration Track*, Hyderabad, India, 2014, pp. 540-543.

[55] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering,* vol. 10, pp. 405-435, 2005

[56] S. T. Dumais, "Improving the retrieval of information from external sources," *Behavior Research Methods, Instruments, and Computers,* vol. 23, pp. 229 - 236, 1991.

[57] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Guéhéneuc, "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis," in *16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 53-62.

[58] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, and A. V. Aho, "Do Crosscutting Concerns Cause Defects?," *IEEE Transaction on Software Engineering,* vol. 34, pp. 497-515, July-August 2008.

[59] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Transactions on Software Engineering (TSE),* vol. 29, pp. 210 - 224, March 2003.

[60] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining Source Code to Automatically Split Identifiers for Software Analysis," in *6th IEEE Working Conference on Mining Software Repositories (MSR'09)*, Vancouver, BC, Canada 2009, pp. 71-80.

[61] D. Falessi, G. Cantone, and G. Canfora, "Empirical Principles and an Industrial Case Study in Retrieving Equivalent Requirements via Natural Language Processing Techniques," *IEEE Transactions on Software Engineering (TSE),* vol. 39, pp. 18-44, 2013.

[62] FETCH. (April 15). *FETCH (Fact Extraction Tool CHain), University of Antwerp, http://lore.ua.ac.be/fetchWiki/*.

[63] G. Gay, S. Haiduc, M. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-Based Concept Location," in *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Canada, 2009, pp. 351-360.

[64] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated Impact Analysis for Managing Software Changes," in *34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012, pp. 430-440.

[65] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Link Recovery," in *27th IEEE International Conference on Software Maintenance (ICSM'11)*, Williamsburg, Virginia, USA, 2011, pp. 133-142.

[66] M. Gethers and D. Poshyvanyk, "Using Relational Topic Models to Capture Coupling among Classes in Object-Oriented Software Systems," in *26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timişoara, Romania, 2010, pp. 1-10.

[67] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "CodeTopics: Which Topic Am I Coding Now?," in *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), Formal Research Tool Demonstration*, Honolulu, Hawaii, USA, 2011, pp. 1034-1036.

[68] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1989.

[69] J. M. González-Barahona and G. Robles, "On the Reproducibility of Empirical Software Engineering Studies based on Data Retrieved from Development Repositories," *Empirical Software Engineering (ESE),* vol. 17, pp. 75-89, February 2012.

[70] S. Grant and J. R. Cordy, "Estimating the Optimal Number of Latent Concepts in Source Code Analysis," in *10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, Timişoara, Romania, 2010, pp. 65-74.

[71] S. Grant, J. R. Cordy, and D. B. Skillicorn, "Automated Concept Location Using Independent Component Analysis " in *15th IEEE Working Conference on Reverse Engineering (WCRE'08)*, Antwerp, Belgium, 2008, pp. 138-142.

[72] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "Exemplar: EXEcutable exaMPLes ARchive," in *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 2010, pp. 259-262.

[73] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine For Finding Highly Relevant Applications," in *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 2010, pp. 475-484.

[74] T. Griffiths and M. Steyvers, "Finding scientific topics," *Proc. of the National Academy of Sciences* 2004.

[75] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "TIDIER: An Identifier Splitting Approach using Speech Recognition Techniques," *Journal of Software Maintenance and Evolution: Research and Practice (JSME),* vol. to appear, 2011.

[76] S. Haiduc, J. Aponte, and A. Marcus, "Supporting Program Comprehension with Source Code Summarization," in *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa 2010, pp. 223-226.

[77] S. Haiduc and A. Marcus, "On the Use of Domain Terms in Source Code," in *16th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 113-122.

[78] M. Hays, J. H. Hayes, A. J. Stromberg, and A. C. Bathke, " Statistical analysis for traceability experiments: Software verification and validation research laboratory (SVVRL) of the University of Kentucky," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 90-94.

[79] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse," in *31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, Vancouver, British Columbia, Canada, 2009.

[80] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the Naturalness of Software," in *34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012, pp. 837-847.

[81] A. J. Hindle, M. W. Godfrey, and R. C. Holt, "What's Hot and What's Not: Windowing Developer Topic Analysis," in *IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, 2009.

[82] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*: U Michigan Press, 1975.

[83] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics,* vol. 6, pp. 65-70, 1979.

[84] JabRef. (2011, 7/19/2011). *JabRef - Reference Manager*. Available: http://jabref.sourceforge.net/

[85] jEdit. (2011, 7/19/2011). *jEdit - Programmer's Text Editor*. Available: http://www.jedit.org/

[86] M. Jørgensen, "A Review of Studies on Expert Estimation of Software Development Effort," *Journal of Systems and Software (JSS),* vol. 70, pp. 37-60, 2004.

[87] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software," *Empirical Software Engineering (EMSE),* vol. 18, pp. 933-969, October 2013.

[88] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code," in *17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, Beverly, Massachusetts, USA, 2010, pp. 119-128.

[89] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning Change Requests to Software Developers," *Journal of Software: Evolution and Process (JSEP),* vol. 24, pp. 3–33, January 2012.

[90] N. Kaushik and L. Tahvildari, "A Comparative Study of the Performance of IR Models on Duplicate Bug Detection," in *16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, 2012, pp. 159-168.

[91] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hussein, and D. Hearn, "TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions," in *34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012, pp. 1375-1378.

[92] Kepler. (April 24). *The Kepler Project - University of California*.

[93] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using Mapping Studies as the Basis for Further Research - A Participant-Observer Case Study," *Information and Software Technology,* vol. 53, pp. 638-651, 2011.

[94] S. Klock, M. Gethers, B. Dit, and D. Poshyvanyk, "Traceclipse: An Eclipse Plug-in for Traceability Link Recovery and Management," in *6th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'11)*, Honolulu, Hawaii, HI, 2011.

[95] J. Kogan, *Introduction to Clustering Large and High-Dimensional Data*: Cambridge University Press, 2006.

[96] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing Source Code Vocabulary," in *17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, Beverly, Massachusetts, USA, 2010, pp. 3-12.

[97] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective Identifier Names for Comprehension and Memory," *Innovations in Systems and Software Engineering,* vol. 3, pp. 303-318, 2007.

[98] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a Name? A Study of Identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, 2006, pp. 3-12.

[99] T. D. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk, "RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information," presented at the 23rd IEEE International Conference on Program Comprehension (ICPC'15), Florence, Italy, 2015.

[100] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," *Cybernetics and Control Theory,* vol. 10, pp. 707-710, 1966.

[101] W. Li and J. H. Hayes, "Query+ enhancement for semantic tracing (QuEST): Software verification and validation research laboratory (SVVRL) of the University of Kentucky," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 95-99.

[102] M. Linares-Vasquez, H. Dang, K. Hossen, K. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging Incoming Change Requests: Bug or Commit History, or Code Authorship?," in *28th IEEE International Conference on Software Maintenance (ICSM'12)*, Riva del Garda, Italy, 2012, pp. 451-460.

[103] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk, "An Exploratory Analysis of Mobile Development Issues Using Stack Overflow," in *10th Working Conference on Mining Software Repositories (MSR'13), Challenge Track*, San Francisco, CA, 2013, pp. 93-96.

[104] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining Eclipse Developer Contributions via Author-Topic Models," in *4th IEEE International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, MN, 2007, pp. 30-33.

[105] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, Atlanta, Georgia, 2007, pp. 234-243.

[106] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides, "Modeling Class Cohesion as Mixtures of Latent Topics," in *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, 2009, pp. 233-242.

[107] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving Trace Accuracy through Data-Driven Configuration and Composition of Tracing Features," in *9th Joint Meeting of the European Software Engineering Conference and the 21st*

*ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Saint Petersburg, Russia, 2013, pp. 378-388.

[108] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using Latent Dirichlet Allocation," *Information and Software Technology,* vol. 52, pp. 972-990, 2010.

[109] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering (TSE),* vol. 17, pp. 800-813, 1991.

[110] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," in *23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Ontario, Canada, 2001, pp. 103-112.

[111] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," in *25th IEEE/ACM International Conference on Software Engineering (ICSE'03)*, Portland, OR, 2003, pp. 125-137.

[112] A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of Traceability Links Between Software Documentation and Source Code," *International Journal of Software Engineering and Knowledge Engineering,* vol. 15, pp. 811-836, October 2005.

[113] A. Marcus and D. Poshyvanyk, "The Conceptual Cohesion of Classes," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, 2005, pp. 133-142.

[114] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems," *IEEE Transactions on Software Engineering (TSE),* vol. 34, pp. 287-300, 2008.

[115] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, Delft, The Netherlands, 2004, pp. 214-223.

[116] Mathworks. *Matlab*. Available: http://www.mathworks.com/products/matlab/

[117] Mathworks. (April 24). *Simulink*. Available: http://www.mathworks.com/products/simulink/

[118] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting Similar Software Applications," in *34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012, pp. 364-374.

[119] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A Source Code Search Engine For Finding Highly Relevant Applications," *IEEE Transactions on Software Engineering (TSE),* vol. 38, pp. 1069-1087, 2012.

[120] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: A Search Engine for Finding Functions and Their Usages," in *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), Formal Research Tool Demonstration*, Honolulu, Hawaii, USA, 2011, pp. 1043-1045.

[121] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding Relevant Functions And Their Usages," in *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11)*, Honolulu, Hawaii, USA, 2011, pp. 111-120.

[122] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Searching for Relevant Functions and Their Usages in Millions of Lines of Code," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 22, p. to appear, October 2013.

[123] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'09)*, Vancouver, Canada, 2009, pp. 41-48.

[124] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012 ). *The PROMISE Repository of Empirical Software Engineering Data*. Available: http://promisedata.googlecode.com

[125] A. Michail and D. Notkin, "Assessing software libraries by browsing similar classes, functions and relationships," in *IEEE International Conference on Software Engineering (ICSE'99)*, 1999, pp. 463-472.

[126] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "ExPort: Detecting and Visualizing API Usages in Large Source Code Repositories," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, Palo Alto, CA, 2013, pp. 646-651.

[127] Mozilla. (2011, 7/19/2011). *Rhino: JavaScript for Java*. Available: http://www.mozilla.org/rhino/

[128] muCommander. (2011, 7/19/2011). *muCommander - Cross-Platform File Manager*. Available: http://www.mucommander.com/

[129] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "The Effect of Omitted-Variable Bias on the Evaluation of Compiler Optimizations," *IEEE Computer,* vol. 43, pp. 62-67, 2010.

[130] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, Lawrence, Kansas, USA, 2011, pp. 263-272.

[131] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying Method Friendships to Remove the Feature Envy Bad Smell," in *33rd IEEE/ACM International Conference on Software Engineering (ICSE'11)*, Honolulu, Hawaii, USA, 2011, pp. 820-823.

[132] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," in *18th IEEE International Conference on Program Comprehension (ICPC'10)*, Braga, Portugal, 2010, pp. 68-71.

[133] Oracle. (2011, 7/19/2011). *JPDA (Java Platform Debugger Architecture)*. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jpda.html

[134] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to Effectively Use Topic Models for Software Engineering Tasks? An Approach based on Genetic Algorithms," in *35th IEEE/ACM International Conference on Software Engineering (ICSE'13)*, San Francisco, CA, 2013, pp. 522-531.

[135] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "When and How Using Structural Information to Improve IR-based Traceability Recovery," in *17th European Conference on Software Mainenance and Reengineering (CSMR'13)*, Genova, Italy, 2013, pp. 199-208.

[136] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering," presented at the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), Italy, 2008.

[137] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling, "Fast Collapsed Gibbs Sampling For Latent Dirichlet Allocation," in *14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008, pp. 569-577.

[138] M. Porter, "An Algorithm for Suffix Stripping," *Program,* vol. 14, pp. 130-137, July 1980.

[139] PorterStemmer. *Porter Stemmer*. Available: http://tartarus.org/~martin/PorterStemmer

[140] D. Poshyvanyk, "Using Information Retrieval to Support Software Maintenance Tasks," in *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, 2009, pp. 453-456.

[141] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept Location using Formal Concept Analysis and Information Retrieval," *ACM Transactions on Software Engineering and Methodology,* vol. 21, p. to appear, November 2012.

[142] D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, 2006, pp. 137-146.

[143] D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering (TSE),* vol. 33, pp. 420-432, June 2007.

[144] D. Poshyvanyk and A. Marcus, "The Conceptual Coupling Metrics for Object-Oriented Systems," in *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, USA, 2006, pp. 469 - 478.

[145] D. Poshyvanyk and A. Marcus, "Using Information Retrieval to Support Design of Incremental Change of Software," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Doctoral Symposium*, Atlanta, Georgia, 2007, pp. 563-566.

[146] D. Poshyvanyk, A. Marcus, and Y. Dong, "JIRiSS - an Eclipse plug-in for Source Code Exploration," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, 2006, pp. 252-255.

[147] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeyev, "IRiSS - A Source Code Exploration Tool," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, 2005, pp. 69-72.

[148] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using Information Retrieval based Coupling Measures for Impact Analysis," *Empirical Software Engineering,* vol. 14, pp. 5-32, 2009.

[149] D. Poshyvanyk and D. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, 2007, pp. 37-48.

[150] D. Poshyvanyk, M. Petrenko, and A. Marcus, "Integrating COTS Search Engines into Eclipse: Google Desktop Case Study," in *2nd International ICSE'07 Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques (IWICSS'07)*, Minneapolis, MN, 2007.

[151] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu, "Source Code Exploration with Google " in *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, 2006, pp. 334 - 338.

[152] R-Project. *R*. Available: http://www.r-project.org/

[153] Rapid-I. (April 24). *Rapid Miner*. Available: http://rapid-i.com/content/view/181/190/

[154] D. Ratiu and F. Deissenboeck, "From Reality to Programs and (Not Quite) Back Again," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, 2007, pp. 91-102.

[155] P. Rempel, P. Mader, and T. Kuschke, "Towards feature-aware retrieval of refinement traces," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 100-104.

[156] M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," in *18th IEEE International Conference on Program Comprehension (ICPC'10)*, Braga, Portugal, 2010, pp. 14-23.

[157] M. Revelle and D. Poshyvanyk, "An Exploratory Study on Assessing Feature Location Techniques," in *17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, British Columbia, Canada, 2009, pp. 218-222.

[158] G. Robles, "Replicating MSR: A Study of the Potential Replicability of Papers Published in the Mining Software Repositories Proceedings," in *7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, Cape Town, South Africa, 2010, pp. 171-180.

[159] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *29th IEEE/ACM International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, 2007, pp. 499-510.

[160] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM (CACM),* vol. 18, pp. 613-620, November 1975.

[161] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "TopicXP: Exploring Topics in Source Code using Latent Dirichlet Allocation," in *26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timişoara, Romania, 2010, pp. 1-6.

[162] T. Savage, M. Revelle, and D. Poshyvanyk, "FLAT^3: Feature Location and Textual Tracing Tool," in *32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 2010, pp. 255-258.

[163] S. J. Sayyad and T. J. Menzies. (2005 July 17). *The PROMISE Repository of Software Engineering Databases*. Available: http://promise.site.uottawa.ca/SERepository

[164] G. Scanniello and A. Marcus, "Clustering Support for Static Concept Location in Source Code," in *19th IEEE International Conference on Program Comprehension (ICPC'11)*, Kingston, Ontario, Canada, 2011, pp. 1-10.

[165] B. Sharif and J. I. Maletic, "An Eye Tracking Study on camelCase and under_score Identifier Styles," in *18th IEEE International Conference on Program Comprehension (ICPC'10)*, Braga, Portugal, 2010, pp. 196-205.

[166] T. U. o. Sheffield. (2011, April 24). *GATE: General Architecture for Text Engineering*. Available: http://gate.ac.uk/

[167] D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker, "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns," in *6th International Conference on Aspect Oriented Software Development (AOSD'07)*, 2007, pp. 212-224.

[168] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in Empirical Software Engineering," *Empirical Software Engineering,* vol. 13, pp. 211-218, 2008.

[169] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," *Communications of the ACM,* vol. 26, November 1983.

[170] S. Stemmer. (June 2012). *Snowball Stemmer*. Available: http://snowball.tartarus.org/

[171] R. Tairas and J. Gray, "An Information Retrieval Process to Aid in the Analysis of Code Clones," *Empirical Software Engineering,* vol. 14, pp. 33-56, February 2009.

[172] A. Takang, P. Grubb, and R. Macredie, "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation," *Journal of Programming Languages,* vol. 4, pp. 143-167, 1996.

[173] Taverna. (April 15). *Taverna, myGrid team, http://www.taverna.org.uk/*.

[174] Team, R Development Core, "R: A Language and Environment for Statistical Computing," ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria, 2013. url: http://www.R-project.org2005.

[175] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, "Hierarchical Dirichlet Processes," *Journal of the American Statistical Association,* vol. 101, pp. 1566-1581, 2006.

[176] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the Use of Topic Models for Software Evolution," in *10th IEEE International Working Conference on*

*Source Code Analysis and Manipulation (SCAM'10)*, Timişoara, Romania, 2010, pp. 55-64.

[177] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static Test Case Prioritization Using Topic Models," *Empirical Software Engineering (ESE),* p. to appear, 2012.

[178] K. Tian, M. Revelle, and D. Poshyvanyk, "Using Latent Dirichlet Allocation for Automatic Categorization of Software," in *6th IEEE Working Conference on Mining Software Repositories (MSR'09)*, Vancouver, British Columbia, Canada, 2009, pp. 163-166.

[179] A. Von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer,* vol. 28, pp. 44-55, 1995.

[180] T. U. o. Waikato. (April 24). *WEKA*. Available: http://www.cs.waikato.ac.nz/ml/weka/

[181] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for Better Configurations: a Rigorous Approach to Clone Evaluation," presented at the 9th Joint Meeting of the European Software Engineering Conference and the 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), Saint Petersburg, Russia, 2013.

[182] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information," in *30th IEEE/ACM International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 461-470.

[183] M. Wieloch, S. Amornborvornwong, and J. Cleland-Huang, "Trace-by-classification: A machine learning approach to generate trace links for frequently occurring software artifacts," in *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*, 2013, pp. 110-114.

[184] A. Wiese, V. Ho, and E. Hill, "A Comparison of Stemmers on Source Code Identifiers for Software Search," in *27th IEEE International Conference on Software Maintenance (ICSM'11)*, Williamsburg, Virginia, USA, 2011, pp. 496-499.

[185] D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Optimization," *IEEE Transactions on Evolutionary Computation,* vol. 1, pp. 67-82, April 1997.

[186] X. Xie, D. Poshyvanyk, and A. Marcus, "3D Visualization for Concept Location in Source Code," in *28th IEEE/ACM International Conference on Software Engineering (ICSE'06)*, Shanghai, China, 2006, pp. 839-842.

[187] Y. Ye and G. Fischer, "Reuse-Conducive Development Environments," *Journal Automated Software Engineering,* vol. 12, pp. 199-235, 2005.

[188] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *3rd International Workshop on Predictor Models in Software Engineering (PROMISE'07)*, Minneapolis, MN, USA, 2007, p. 9.