TraceLab: Reproducing Empirical Software Engineering Research

Evan Alexander Moritz

New Kent, Virginia

Bachelor of Arts, College of William and Mary, 2011

A Thesis presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Master of Science

Department of Computer Science

The College of William and Mary
May 2013

APPROVAL PAGE

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

Master of Science

Evan Alexander Moritz

Approved by the Committee, May, 2013

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

Professor Evengia Smirni, Computer Science
The College of William and Mary

Professor Jane Cleland-Huang, Computer Science
DePaul University

ABSTRACT

The ability to reproduce experiments in software engineering research is a hidden issue in validating and improving on new approaches. The lack of tool support, data availability, implementation-specific details, and even minute environment differences all contribute to problems for researchers attempting to investigate new ideas. In this thesis, I present examples of how unpublished details of an approach can drastically change the results. To address this issue, I promote the use of TraceLab, a research instrument designed to perform and share software engineering experiments in their entirety with accurate results. I leverage TraceLab's ability to extend its framework with new tools and functionality to create a new Component Library (CL) and Component Development Kit (CDK) designed to provide researchers with all of the tools necessary to evaluate and improve new techniques. To discover which tools to include, I perform a systematic mapping study of publications from a subset of top international software engineering conferences in the past 10 years. Based on these results, I implemented the most popular tools and techniques in the CL and CDK. I show that by using the CL and CDK in TraceLab, 37% of the approaches identified in the mapping study can be completely recreated, with an additional 37% of approaches missing only 1 technique. Lastly, I reproduce examples of existing software engineering approaches that provides a working body of knowledge in order to drive new research.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

First and foremost, I thank my advisor, Denys Poshyvanyk, without whom this thesis would not have been possible.

I thank current and former members of the SEMERU research group for their hard work, vibrant discussions, and constant feedback that aided in writing this thesis.

I thank the developers of TraceLab, which provided the foundational motivation for this thesis.

I thank my committee members for providing valuable support and criticism, and for reading this tome.

I thank all of my friends and family who have supported me throughout my graduate studies and encouraged me to pursue my passions.

This work was funded in by the National Science Foundation grant CNS-0959924.

I present this thesis in honor of my parents,
Robert and Janice.

LIST OF TABLES

LIST OF FIGURES

TRACELAB:

REPRODUCING EMPIRICAL SOFTWARE ENGINEERING RESEARCH

Chapter 1

Introduction

The field of software engineering in academic research is blossoming as modern technology becomes increasingly ingrained in our daily lives. This has resulted in a wealth of new ideas and interest, as evidenced by the large number of software engineering conferences and growing number of software engineering research groups at campuses across the globe. In the private sector, technology companies such as IBM[1], Microsoft[2], and Google[3] have their own branches dedicated to research. All of this has produced a large body of work which continues to grow every year.

As a science, software engineering strives to improve our lives through the use of technology. Unfortunately, one of the main tenets of the scientific method – reproducibility – remains rarely achievable. Each research group has its own customs and practices, data formats, homegrown tools, and projects. This makes sharing the specifics of a tool or technique difficult and time consuming, not only with external researchers but also with collaborators and even project members within the group. Sharing with the community at large through conferences and publications is limited

[1]`http://www.research.ibm.com/`
[2]`http://research.microsoft.com/`
[3]`http://research.google.com/`

to the paper or article itself. The tools, data, and even technical specifications of a technique are almost never provided. The internal settings of the experiment and even the environment in which the experiment is run remain undocumented [1]. The effects of this situation result in outside researchers wasting valuable time and resources reimplmenting established work from scratch, or to quote the old adage: "Reinventing the wheel."

Clearly, there exists a need for standardization in software engineering research. The Center of Excellence for Software Traceability (CoEST)[4] has taken this challenge head-on. Researchers at DePaul University[5] have been developing a research instrument in collaboration with Kent State University[6], University of Kentucky[7], and the College of William & Mary. This research instrument, called TraceLab, is funded by a grant from the the National Science Foundation (NSF) [2]. The main goals of TraceLab are to facilitate collaboration between researchers and jump-start the research process by providing a robust framework to perform experiments in software engineering. TraceLab provides many of the tools needed for software engineering research straight out of the box and comes with a software development kit (SDK) to create new ones [3, 4, 5].

TraceLab heralds a major shift in the way software engineering research is conducted. In this thesis, I provide motivations for transparency within the academic field of software engineering research. I describe and encourage the use of TraceLab as a foundational tool for performing academic research. I perform a survey of modern software engineering publications in the fields of traceability link recovery, program comprehension, feature location, and duplicate bug detection for the purposes of creating a collection of common tools for use in TraceLab. I describe the

[4]`http://www.coest.org/`
[5]`http://www.depaul.edu/`
[6]`http://www.kent.edu/`
[7]`http://www.uky.edu/`

structure and function of these tools, collectively known as the Component Library. Finally, I provide examples of reproducing previous research using the Component Library in TraceLab.

The rest of this thesis is organized as follows: Chapter 2 describes previous work in examining software engineering research and provides a comparison of tools similar to TraceLab. Chapter 3 provides a motivating example enumerating the issues in the current state of software engineering research. Chapter 4 describes in detail the different aspects of TraceLab and how it can be used for software engineering research. Chapter 5 details the process and results of a systematic mapping study of software engineering research techniques and approaches. Chapter 6 describes the structure and function of a new component library developed for TraceLab based on the results of the mapping study. Chapter 7 presents examples of reproducing previous software engineering research using TraceLab and the component library. Finally, the thesis concludes with Chapter 8.

Chapter 2

Related Work

This chapter discusses the existing work that examines the state of software engineering research, investigating the problems with reproducibility and evaluating commonly used tools in software engineering research in comparison to TraceLab.

2.1 Studies in Reproducibility

There have been several meta-studies in the research community investigating problems with the state of academic research and their effects on the ability to reproduce and drive new research.

In a survey of feature location techniques by Dit et al. [6], the authors found that only 38% of the papers they surveyed performed a comparison of the approach proposed in the paper to previously established approaches. Without this comparison, it is nearly impossible to determine whether a new approach is valid or whether it results in statistically significant improvement. Furthermore, the authors found that 5% of the surveyed papers performed a comparison using the same data as the previous approaches. Using different datasets could further cloud the validity of new approaches.

In a study by Robles [7], every paper accepted to the Working Conference on Mining Software Repositories (MSR) from 2004-2009 was investigated in search of reproducible approaches. Robles found that a majority of published submissions contained evaluations that were impossible to reproduce due to various factors, such as unavailable datasets, lack of tool support, and critical implementation details that were missing from the paper. Furthermore, only two papers from that time period made their data and tools publicly available.

D'Ambros et al. [8] evaluated a set of defect prediction approaches and found it difficult to compare the results among different approaches. The authors proposed that many of the approaches they studied were not evaluated correctly, either presenting the results of the approach by itself or in comparison to a small number of others.

Mytkowicz et al. [9] investigated the field of compiler optimization for the effects of ommiting seemingly unimportant aspects of the approach on the results of the experiment. Leaving out minute details, such as internal compiler settings or the order of objects processed by the linker, can greatly effect the outcome of an experiment in unexpected ways. Without this knowledge, it may be difficult or even impossible to accurately recreate an approach.

Barr et al. [10] discussed issues in the academic community with sharing research, such as the fear of being beaten to new findings by another research group. They compared the field of software engineering to other fields, such as medicine and physics, and discussed the benefits of sharing within those communities. The authors proposed different methods of facilitating collaboration and encourage the practice within software engineering research.

González-Barahona and Robles [11] investigated why certain approaches are reproducible and why others are difficult, attempting to determine which characteristics effect the reprodicibility of a study. The authors proposed a methodology for evaluating these characteristics and interpreting the results of the classification.

Borg et al. [12] performed a mapping study investigating publications that used information retrieval techniques in their approach. They found that most studies do not perform an evaluation on datasets with more than 500 artifacts and identified a need for industrial case studies. They encouraged researchers to publicly provide the datasets and tools used in their evaluations and provide a set of guidelines to raise the quality of publications in the field of software engineering research.

To address the problems presented in these papers, several benchmarking datasets have been made publicly available [6, 8, 13, 14, 15, 16, 17]. While these datasets provide a common data source to perform evaluations on, they do not solve all of the problems of reproducibility.

2.2 Research Tools

There are many tools commonly used in experimental research. The search for the "right" tool depends on the needs of the researcher for a particular task. While this may work perfectly for an individual experiment, the effort required dramatically increases once researchers begin attempting to reproduce or build off of the work of others. This is due to the variety and scope of tools that researchers use, in addition to the problems of specific settings, environments, and data formats. This section discusses some commonly used tools and compares them against TraceLab.

The R Project [18] 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 experimental design and can be difficult to reproduce when shared due to the number of libraries and different versions. Additionally, researchers must learn a new programming language when performing experiments in R.

Matlab [19] is similar to R Project but is geared more towards scientific computing tasks. It has more in-depth data analysis tools, such as 3-dimensional visualization. Experiments using Matlab run into the same issues as R, including forcing experimenters to learn an entirely new programming language.

WEKA [20] is a collection of machine learning algorithms and visualization tools in Java. WEKA features a graphical interface composed of tools to perform specific tasks. As an additional feature, WEKA provides a data-flow based interface called KnowledgeFlow. This view is a canvas of tools connected to perform a series of tasks, much in the same manner as TraceLab. However, WEKA lacks tools specific to software engineering research and does not have many of the useful sharing and extensibility features of TraceLab.

RapidMiner [21] builds on top of WEKA's machine learning library and offers an improved interface specifically for designing and executing experiments. It offers the data mining and classification techniques from WEKA combined with statistical computing from R Project. It also provides methods for creating new plugins for use in experiments. However, RapidMiner is not specialized for software engineering research and does not have the sharing features of TraceLab.

Simulink [22] is a tool for simulating embedded systems. It features a model-based design with pluggable components and runs on top of Matlab. However, it is designed for a different domain and does not contain the features that make TraceLab desirable.

GATE [23] is a natural-language processing tool for extracting information from text-based sources. Although this task is similar to many software engineering research tasks, it does not contain domain-specific knowledge and lacks the experimental design aspect of TraceLab.

Yahoo! Pipes [24] is a website data aggregation tool that allows users to collect and modify data from the web. Different components can be connected and configured to perform various tasks, such as collecting articles from a blog feed and providing suggestions to related sites. User-made pipes may be published and shared on the web. Yahoo! Pipes is entirely web-based and requires a Yahoo! account. Although it contains many similar features to TraceLab, it is designed for an entirely different purpose and does not contain functionality from the software engineering research domain.

Kepler [25] is a workflow-oriented tool that allows researchers to model experiments with pluggable components, much like TraceLab. Kepler is developed for many different operating systems and allows users to create and share their own components. The main difference between TraceLab and Kepler is that Kepler is oriented towards scientific computing in math and physics. For this reason, it is not suitable for software engineering research tasks.

Tool	**GUI**	**Plat.**	**License**	**Repos.**	**Lang.**	**OS**
GATE	N	D,API	OSS	Y	J	W,L,M
Kepler	Y	D,API	OSS	Y	J,R,C,Mat	W,L,M
Matlab	N	D	C	Y	Mat	W,L,M
R Project	N	D	OSS	Y	R	W,L,M
RapidMiner	Y	D,API	OSS	N	J	W,L,M
Simulink	Y	D	C	Y	C,Mat,F	W,L,M
TraceLab	**Y**	**D,API**	**OSS**	**Y**	**MS,J,R,Mat**	**W,L,M**
WEKA	Y	D,API	OSS	N	J	W,L,M
Yahoo! Pipes	Y	W	F	Y	-	-

Table 2.1: Comparison of research tools to TraceLab

Table 2.1 shows a comparison of features between other tools and TraceLab. The **GUI** column indicates whether or not the tool uses a graphical user interface to perform experiments (**Y**es, **N**o). The **Plat.** column describes the intended platform of the tool (**D**esktop, **API** use, **W**eb-based). The **License** column indicates what kind of license is required to use the tool (**C**ommercial, **O**pen **S**ource **S**oftware, **F**ree to use). The **Repos.** column indicates whether or not the tool provides a repository of user-made plugins (**Y**es, **N**o). The **Lang.** column lists the different programming languages that the tool supports (**C**/C++, **J**ava, **F**ortran, **Mat**lab, **M**icro**S**oft .NET, **R**). Finally, the **OS** column lists the different operating systems that the tool is available to run on (**W**indows, **L**inux, **M**ac).

Chapter 3

Motivating Example

To illustrate the problems of reproducing software engineering research, this chapter details a motivating example of one of the most commonly-used information retrieval techniques for traceability link recovery, the Vector Space Model (VSM) [26]. Many publications contain approaches that use VSM, but either do not specify which steps they used, or may mention that certain steps are "commonly used" in the field but do not implement those steps themselves. This example will investigate the effects on the results of different weighting schemes and preprocessing techniques which, if not specified, can greatly change the outcome of an experiment.

3.1 Overview of Vector Space Model

The Vector Space Model is the most common of information retrieval (IR) techniques used for document-to-document relationship recovery. In its simplest form, this technique creates a term-by-document matrix describing the corpus of documents (Figure 3.1). Each column in the matrix represents a single document, and its rows represent the terms found in the entire corpus. Thus each cell in the matrix is the frequency of that particular term in the document. The benefit is that documents can now be considered as *distributions of terms* and many different mathematical techniques can be applied to and inferred from the data [26].

	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
t_1	8	7	6	0	5	4	6	6
t_2	4	6	4	6	2	1	4	8
t_3	7	0	2	7	7	0	5	0
t_4	5	6	5	5	6	0	2	5
t_5	5	3	0	4	2	7	0	6
t_6	7	4	6	0	4	1	5	5

Figure 3.1: term-by-document matrix

A common practice in VSM is to weight the term-by-document matrix to emphasize important terms and trivialize terms that do not add to the meaning of the document. Weighting can be performed in a number of ways, which are investigated in Section 3.2.

To compute the similarity between a pair of artifacts, VSM calculates the cosine of the document vectors via the Euclidean dot product formula (Equation 3.1). Since none of the terms in the document are negative, the cosine of the angle is bound within $[0, 1]$, providing a concrete range for comparison between pairs of documents. Calculating the cosine similarity between pairs of documents and sorting the resulting list provides a ranked-list of candidate links between documents.

If the relationships between documents are already known, the effectiveness of a technique can be measured via precision (Equation 3.2) and recall (Equation 3.3). Precision measures the percentage of links returned that are correct and recall measures the percentage of all correct links that were returned. When recall is plotted against precision for different cutpoints in the ranked list, the resulting graph forms a precision-recall curve which can be used to visually determine the effectiveness of a technique.

$$\text{sim}(\vec{A}, \vec{B}) = \frac{\sum_{i=1}^{n} a_i \times b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \times \sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{3.1}$$

$$\text{P} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{retrieved}|} \tag{3.2}$$

$$\text{R} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{correct}|} \tag{3.3}$$

3.2 Comparison of Weighting Schemes

This section compares the effects of different weighting schemes on the results of computing document-to-document relationships.

3.2.1 Types of weighting schemes

No weighting Once the term-by-document matrix is created, it is perfectly valid to use the term counts within documents for comparison. This method does not gain any of the benefits of weighting techniques that attempt to promote or diminish the contributions of certain terms within the matrix.

tf-idf The standard weighting scheme in software engineering research is tf-idf [27], which emphasizes terms that appear frequently in a document but diminishes the contribution of terms common across all documents. In this scheme, documents in the matrix are normalized by setting the most common term to 1 and dividing all of the other terms in the document by its former value (Equation 3.4). This results in a document consisting of term frequencies (tf). Then the document frequencies (df) are computed by recording the total number of times a term is used throughout the whole corpus (Eq. 3.5). The df are used to calculate the inverse document frequencies (idf) (Eq. 3.6). Then each tf-weighted term in the document is multiplied by its idf, resulting in a tf-idf weight for each term in the document (Eq. 3.7).

$$\begin{aligned}
\mathrm{tf}(t,d) &= \frac{\mathrm{f}(t,\vec{d})}{\max\{\mathrm{f}(w,\vec{d}) \mid w \in \vec{d}\}} && (3.4)\\
\mathrm{df}(t) &= |\{t \in \vec{d}, \vec{d} \in \{D\} \mid \mathrm{tf}(t,\vec{d}) \neq 0\}| && (3.5)\\
\mathrm{idf}(t,\{D\}) &= \log_2 \frac{|D|}{\mathrm{df}(t)} && (3.6)\\
\mathrm{tfidf}(t,\vec{d},\{D\}) &= \mathrm{tf}(t,\vec{d}) \times \mathrm{idf}(t,\{D\}) && (3.7)
\end{aligned}$$

Boolean queries Another practice when using VSM is to treat one set of documents as a known entity and use another set of documents as *queries*, attempting to identify which of the known documents are related to the queries. The known documents are indexed with tf-idf weights. The queries, being unknown beforehand, are given boolean weights. The two matrices must be modeled carefully to ensure that their row indexes correspond to the same terms. Terms that appear in the query are assigned a 1, and terms that are missing are assigned a 0. Any additional terms that were not in the known documents are extended to the known matrix and assigned 0.

3.2.2 Results

Figure 3.2 shows the effects of different weighting schemes on computing relationships between requirements documents and source code in EasyClinic, which is a software system designed to manage medical offices. EasyClinic has been used in the 2009 TEFSE Challenge [28] for evaluating software traceability techniques.

This evaluation was run with the full preprocessing suite of techniques listed in Section 3.3 (cleanup, splitting, stemming, and stopwords removal). Although tf-idf is a clear winner in this example, the results of IR-based traceability are often dataset-dependent. Without disclosing which weighting scheme a researcher used in an approach, other researchers attempting to reproduce the approach may run into challenges in getting similar results.

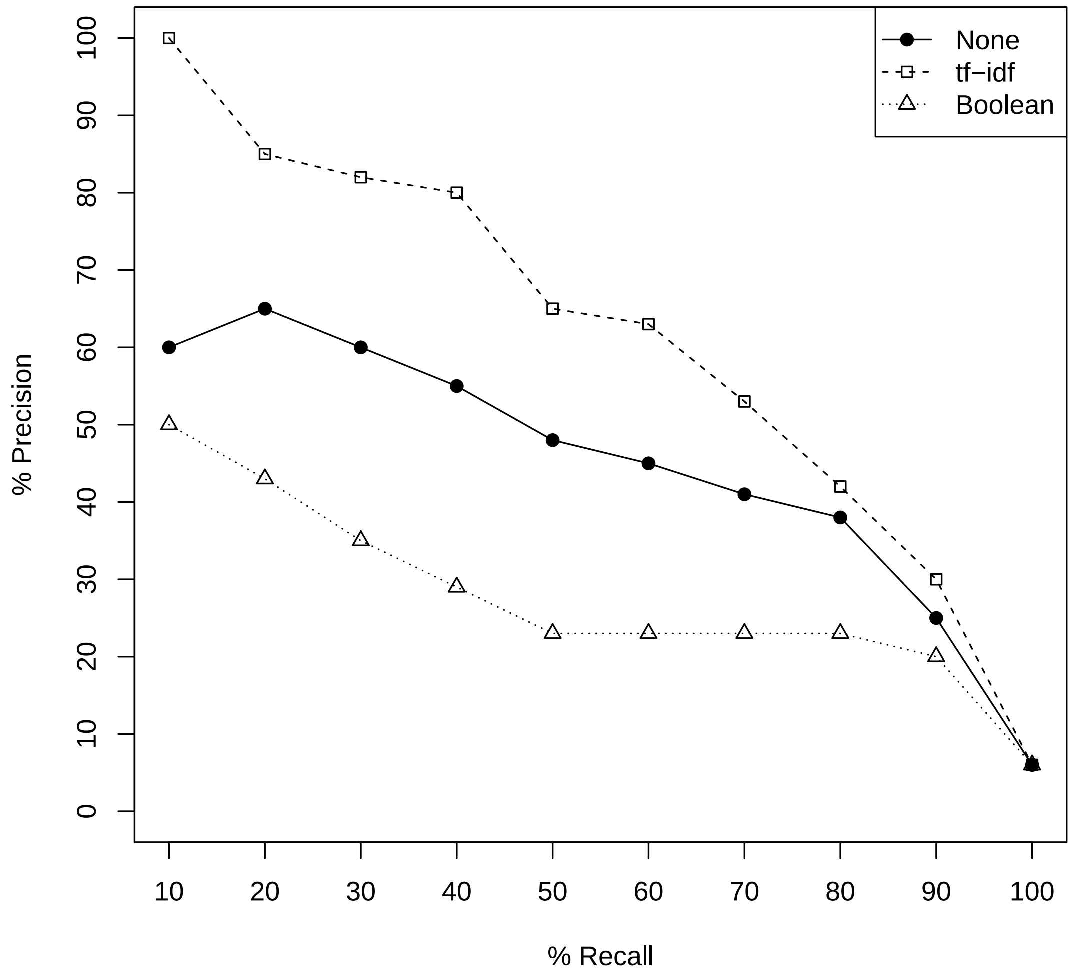


Figure 3.2: Results of different weighting schemes tracing from requirements to source code

3.3 Comparison of Preprocessing Techniques

This section compares the effects of different preprocessing techniques on the results of computing document-to-document relationships.

3.3.1 Preprocessing techniques

No preprocessing Although some sort of preprocessing is normally performed, it is possible to use VSM without any form of preprocessing. The term-by-document matrix is constructed out of the raw data, resulting in separate terms such as "found", "FOUND", "found.", "found,", and "founding".

Basic cleanup This technique turns raw documents into "bag-of-words" documents that have all symbols and punctuation removed. Unless identifier splitting is performed as well, terms are usually converted to lowercase in this stage. This results in combining the terms in the above example to "found" and "founding".

Identifier splitting A common practice in software development is to create variable or method names that describe their function. Different capitalization schemes are employed, such as "CamelCase," "pascalCase," and "CAPITALCase." Splitters can recognize these schemes and separate compound identifiers into their individual words. Studies have been performed investigating the effectiveness of different identifier splitting methods [29, 30].

Stopwords removal Some words are considered to not contribute to the meaning of the text, or they are so common that they increase the noise of the results. Predefined lists of these "stopwords" can be removed from the text in an attempt to increase the effectiveness of an approach, in addition to standalone numbers and terms less than a certain length. In natural language text, an example of these could be articles, prepositions, or pronouns. In source code, certain programming commands are common within the code, such as "for," "if," or "return." Removal of these terms results in a smaller search space, but are difficult to know beforehand.

Word stemming Words often change form when used in different parts of speech or tenses, without changing the basic meaning of the word. For example, the word "find" can also appear as "found," "finds," or "finding." Word stemmers attempt to find the common root of these words and reduce them to a singular form. Thus, all of the words in this example will be stemmed to the root word "find." An example of a popular stemmer is the Porter English stemming algorithm [31].

3.3.2 Results

Figure 3.3 shows the effects of different weighting schemes on computing relationships between requirements documents and source code in EasyClinic. Except for the Raw (no preprocessing) technique, every other technique uses the basic cleanup technique. These techniques may be used in different combinations, further changing the accuracy of the results. This evaluation uses the tf-idf weighting scheme from Section 3.2 and exemplifies the same issues with not disclosing preprocessing techniques for reproduction purposes.

These examples provide merely a small sampling of the issues that can effect the results of an approach. Tiny details such as different weighting schemes or even the order of events in an approach can drastically change the outcome. If these details are not given in the paper, researchers can become frustrated by spending a long time getting inconsistent results when trying to reproduce the approach or try new ideas.

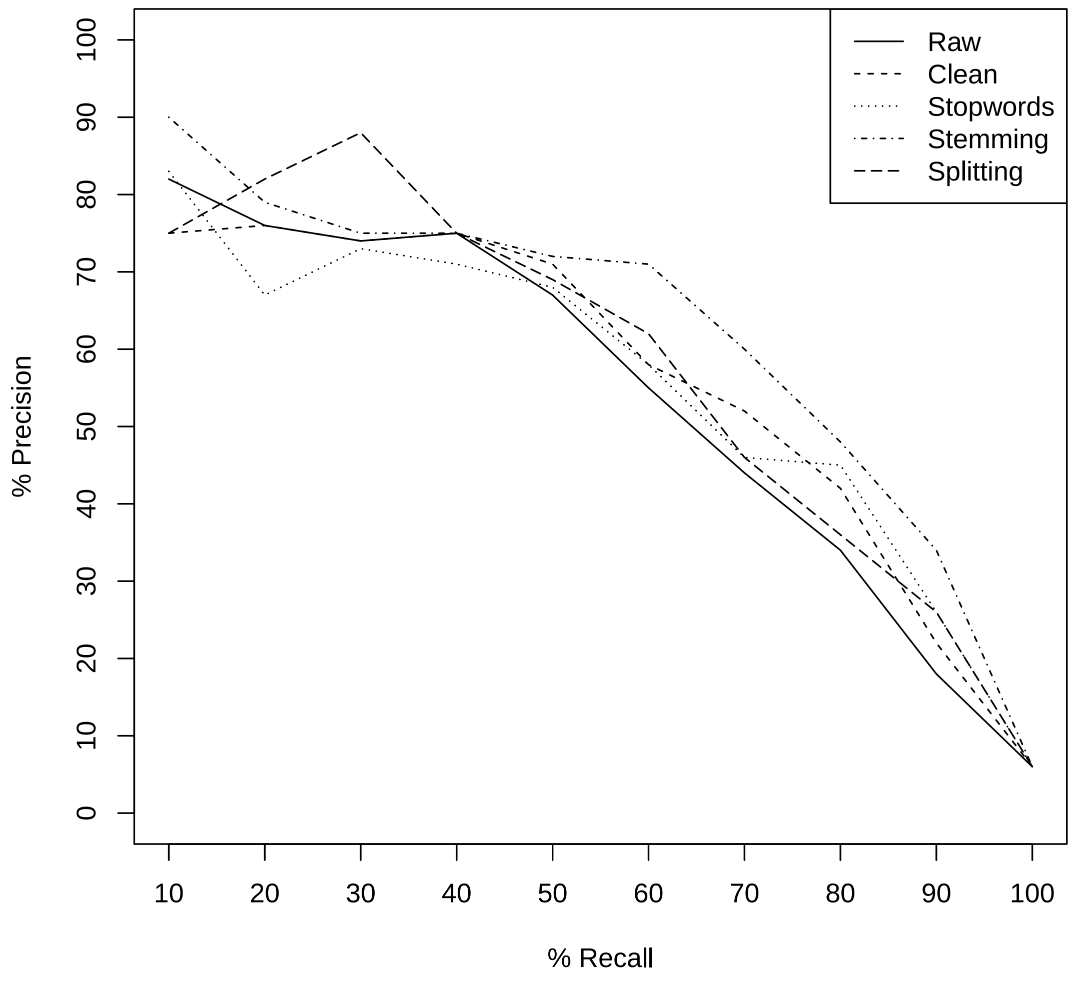


Figure 3.3: Results of different preprocessing techniques tracing from requirements to source code

Chapter 4

TraceLab

TraceLab [3, 4, 5] is a “Software Traceability Instrument to Facilitate and Empower Traceability Research and Technology Transfer” in development by researchers at DePaul University in collaboration with Kent State University, the University of Kentucky, and the College of William & Mary. TraceLab was developed to enable researchers to quickly design experiments in software engineering research by providing a set of tools and resources in an easy-to-use framework. Researchers can effortlessly add to or modify experiments, providing ways to build on existing experiments and investigate new ideas. Furthermore, TraceLab provides an accessible way to share entire experiments with others, providing the tools, data, and exact settings of the experiment for reproducibility. TraceLab is already used by researchers all over the world (Fig. 4.1).

This chapter details the features of TraceLab, including an in-depth description of how researchers can use the tools and develop their own experiments and components. Finally, I will describe my own contributions to the TraceLab project.

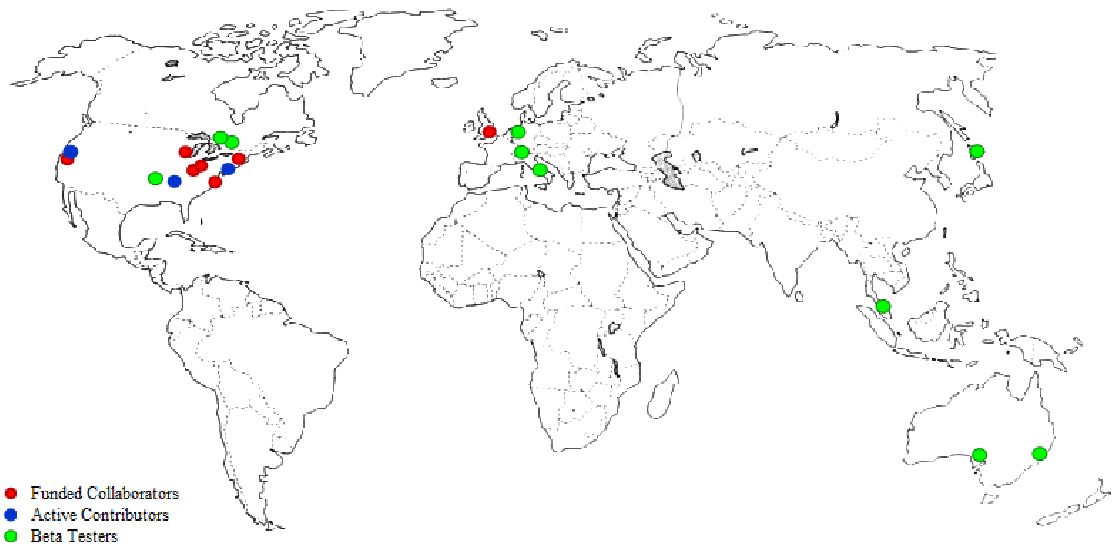


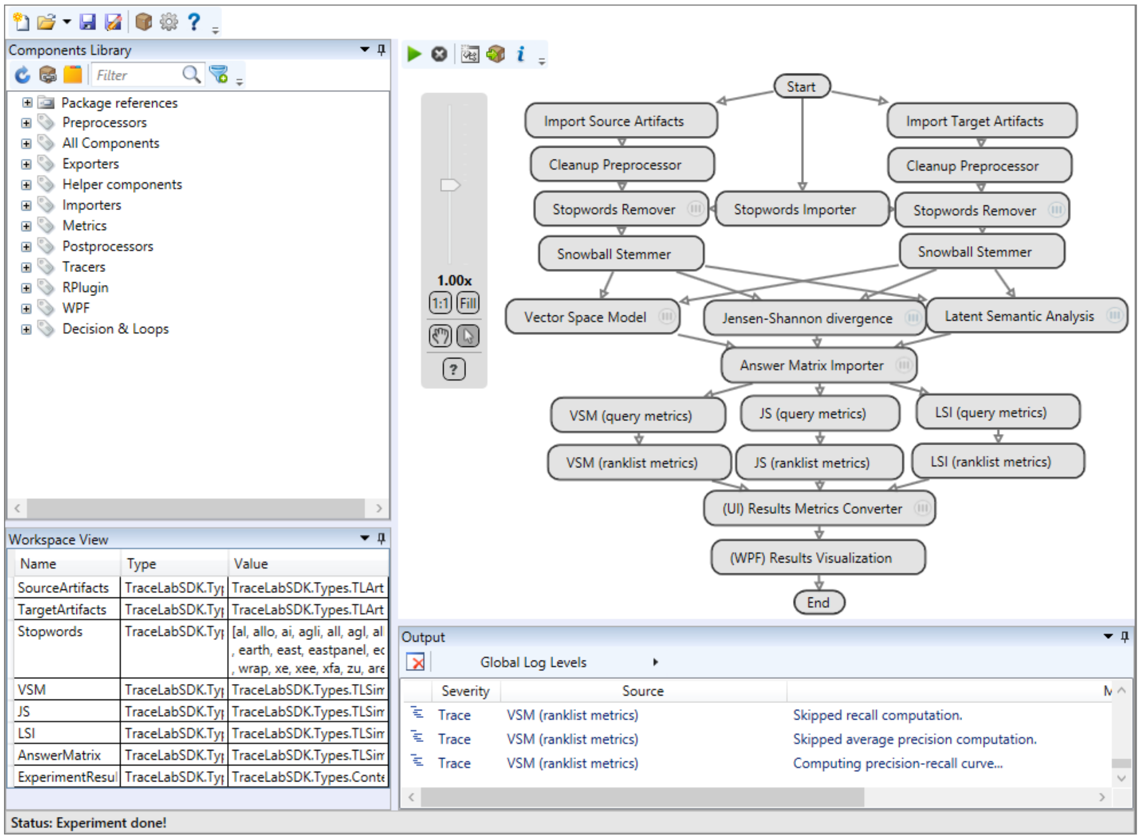
Figure 4.1: Distribution of TraceLab users worldwide as of April 2013

4.1 Overview

At its core, TraceLab is a visual workbench for running experiments in software engineering research. TraceLab presents experiments in the form of a graph composed of tools that share data throughout experiment execution. Experiments may be packaged and shared to ensure reproducibility. Figure 4.2 shows a screenshot of the layout of TraceLab.

4.1.1 Experimental Graph

The heart of a TraceLab experiment is in its workflow of tools. Independent tools and techniques are represented in TraceLab as *components*, shown as ovals in Figure 4.2 and Figure 4.3. An experiment is a directed precedence graph of components. Execution begins at the “Start” node and flows through every path to the “End” node, which completes the experiment. Since it is a precedence graph, each node must wait for all of the incoming edges to complete before executing. This ensures that the previous techniques have completed and the correct data is



Workspace View

Name	Type	Value
SourceArtifacts	TraceLabSDK.Ty	TraceLabSDK.Types.TLArt
TargetArtifacts	TraceLabSDK.Ty	TraceLabSDK.Types.TLArt
Stopwords	TraceLabSDK.Ty	[al, allo, ai, agli, all, agl, al , earth, east, eastpanel, ec , wrap, xe, xee, xfa, zu, are
VSM	TraceLabSDK.Ty	TraceLabSDK.Types.TLSim
JS	TraceLabSDK.Ty	TraceLabSDK.Types.TLSim
LSI	TraceLabSDK.Ty	TraceLabSDK.Types.TLSim
AnswerMatrix	TraceLabSDK.Ty	TraceLabSDK.Types.TLSim
ExperimentResul	TraceLabSDK.Ty	TraceLabSDK.Types.Conte

Output

```
Severity   Source                    M
Trace      VSM (ranklist metrics)    Skipped recall computation.
Trace      VSM (ranklist metrics)    Skipped average precision computation.
Trace      VSM (ranklist metrics)    Computing precision-recall curve...
```

Status: Experiment done!

Figure 4.2: Layout of TraceLab workbench

available.

Components in TraceLab are implicitly parallelizable. Each component is given its own copy of the data and is run in a separate thread. Therefore, when two components branch out from a parent component (such as components "Load data 1" and "Load data 2" in Figure 4.3) they each will run concurrently and independently. This is built into the TraceLab framework, so researchers and component developers do not need to take any special action to acheive this benefit.

TraceLab provides many control flow elements to allow for dynamic experiment flow. *Goto decisions* (Fig. 4.4) allow flow redirection to any of the outgoing nodes based on a given condition. *If statement decisions* (Fig. 4.5) go one step further by directing the flow to one of a number of subgraphs (called *scopes*) based on a given condition. Scopes provide independent experiment graphs that execute in their own

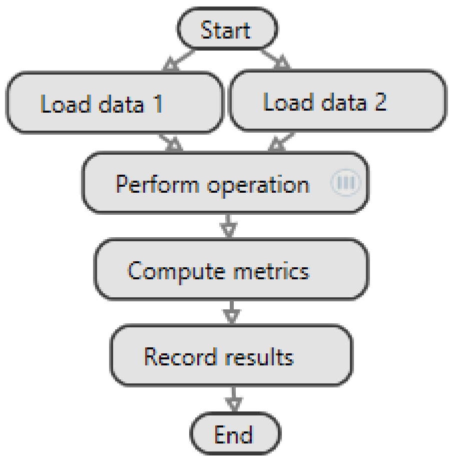


Figure 4.3: Sample experiment in TraceLab

namespace and once completed, return to the parent graph. Similarly, *While loops* (Fig. 4.6) repeatedly execute the scope as long as the given condition is true.

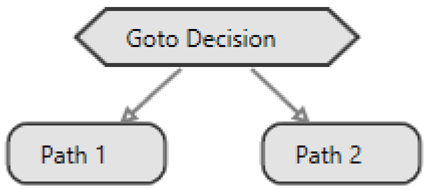


Figure 4.4: Goto decision in TraceLab

4.1.2 Component Library

The *component library* (shown in the top-left of Figure 4.2) lists all of the tools and techniques available to the researcher for use in an experiment. Components may be categorized by multiple *tags*, both by component developers and users. To use a component in an experiment, users need only to drag-and-drop the component from the component library and connect it into the workflow.

Each component has a set of metadata that 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, and versioning information.

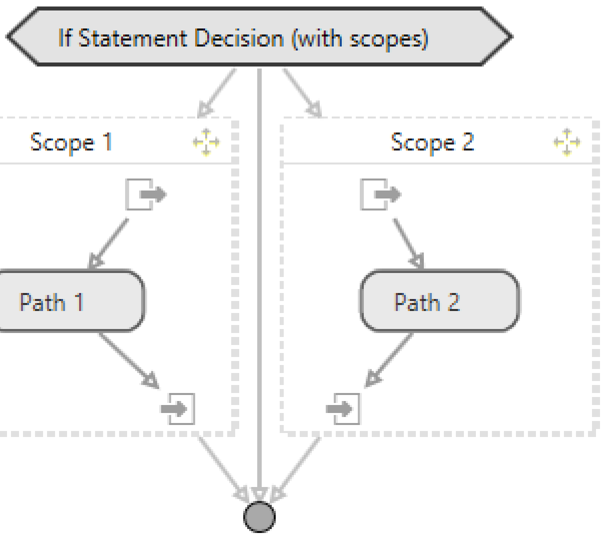


Figure 4.5: If statement decision in TraceLab

Each component must declare its 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 (perhaps named "SourceArtifacts" and "TargetArtifacts") and declare a `TLSimilarityMatrix` as output. This allows TraceLab to evaluate the experiment graph before running it, checking for valid inputs and flow errors. If a component declares an input that is not an output of any preceding components, TraceLab will catch the error before the experiment starts.

Additionally, components may declare a *configuration object* that describes additional settings when running the experiment. A common practice in TraceLab is to declare *data* as inputs and outputs to and from the Workspace (Section 4.1.3) and *settings*–such as technique-specific parameters–as configurations.

The component metadata, declared inputs, and configuration parameters can be viewed and edited in the information pane for each component (Figure 4.7).

More detail about the component library is given in Chapter 6. Information about building custom components is given in Section 4.2.1.

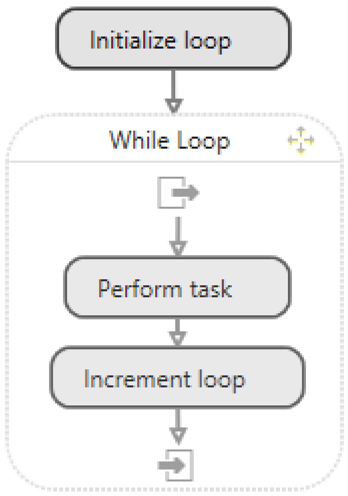


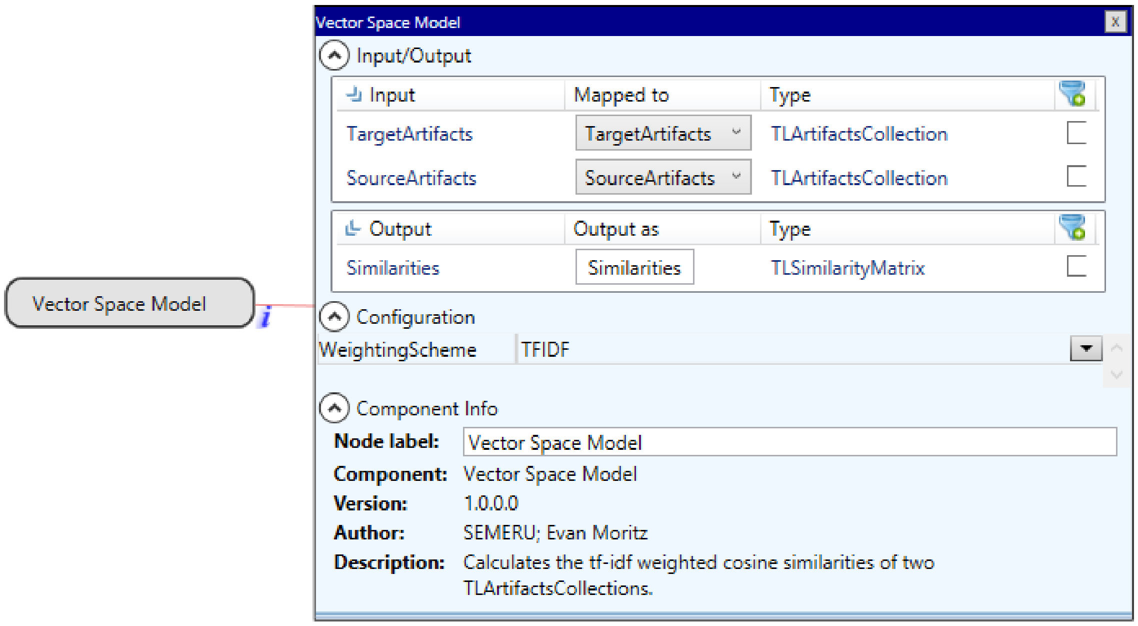
Figure 4.6: While loop in TraceLab

4.1.3 Workspace

The *workspace* (shown in the bottom left of Figure 4.2) is the data-sharing interface that allows components to communicate with one another during the course of experiment execution. Components can load and store data from and to 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 serialized to disk as an XML file for later use. Additionally, some data types can be viewed from the workspace by clicking on their workspace entry. Additional information about workspace data is given in Section 4.2.2.

4.1.4 Component Log

The *component log* (shown in the bottom right of Figure 4.2) is a convenient way to display messages to the user during experiment execution. There are different levels of severity that can be written to the log, such as *info, trace, debug, warning, and error*. Each log entry displays the component name, severity, message, and optionally an exception dialogue describing an uncaught exception and a stack trace.



Input	Mapped to	Type
TargetArtifacts	TargetArtifacts	TLArtifactsCollection
SourceArtifacts	SourceArtifacts	TLArtifactsCollection

Output	Output as	Type
Similarities	Similarities	TLSimilarityMatrix

WeightingScheme TFIDF

Node label: Vector Space Model
Component: Vector Space Model
Version: 1.0.0.0
Author: SEMERU; Evan Moritz
Description: Calculates the tf-idf weighted cosine similarities of two TLArtifactsCollections.

Figure 4.7: Info pane in TraceLab

4.1.5 Packaging Feature

In order to share a TraceLab experiment, all of the necessary information must be included. Therefore, the packaging feature of TraceLab allows a user to collect and specify the datasets and custom components used in the experiment. This information is included with the experiment in an all-in-one package that can be sent to other users and run exactly the same as the original researcher.

4.2 User-defined Components and Types

TraceLab ships with a software development kit (SDK) that allows users to define their own custom components and types in .NET languages[1], Java[2], and (via plugins) R [18] and Matlab [19]. This section will describe in detail how to create, register, and use these components in TraceLab. The descriptions here primarily deal with implementations in C#, with a section summarizing use of other languages.

[1]`http://www.microsoft.com/net`
[2]`http://www.java.com/`

4.2.1 Components

All components in TraceLab must inherit from the `BaseComponent` abstract class defined in the TraceLab SDK. Classes inheriting from `BaseComponent` must override the `Compute()` method, which should contain the main functionality of the component and is called from TraceLab during experiment execution. 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. The abstract class also gives the component class access to the workspace and informs TraceLab of any configuration settings.

In order for TraceLab to recognize a class as a component for use in an experiment, the class must declare a `[Component]` attribute which contains information about the component's name, description, author, version, and optional configuration object. 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.

Finally, TraceLab needs to know where to look for custom components. After compiling, libraries containing components should be placed in a registered component directory. These directories are defined in TraceLab's settings menu and user-defined directories can be added or removed.

Figure 4.8 shows an example component class definition for use in TraceLab. I leverage the ability to create user-made components and types by creating the TraceLab Component Library described in Chapter 6.

```
[Component(Name = "Vector␣Space␣Model",
  Description = "Calculates␣the␣tf-idf␣weighted␣cosine␣
     similarities␣of␣two␣TLArtifactsCollections.",
  Author = "SEMERU;␣Evan␣Moritz",
  Version = "1.0.0.0",
  ConfigurationType = typeof(VSMComponentConfig))]
[IOSpec(IOSpecType.Input, "SourceArtifacts",
  typeof(TLArtifactsCollection))]
[IOSpec(IOSpecType.Input, "TargetArtifacts",
  typeof(TLArtifactsCollection))]
[IOSpec(IOSpecType.Output, "Similarities",
  typeof(TLSimilarityMatrix))]
[Tag("Tracers.InformationRetrieval")]
public class VSMComponent : BaseComponent
{
  private VSMComponentConfig _config;

  public VSMComponent(ComponentLogger log)
    : base(log)
  {
    _config = new VSMComponentConfig();
    Configuration = _config;
  }

  public override void Compute()
  {
    TLArtifactsCollection sourceArtifacts =
       (TLArtifactsCollection)Workspace.Load("SourceArtifacts");
    TLArtifactsCollection targetArtifacts =
       (TLArtifactsCollection)Workspace.Load("TargetArtifacts");
    TLSimilarityMatrix sims = VSM.Compute(sourceArtifacts,
       targetArtifacts, _config.WeightingScheme);
    Workspace.Store("Similarities", sims);
  }
}
```

Figure 4.8: TraceLab component class

```
[Serializable]
[WorkspaceType]
public class BiGram
{
  public string Caller { get; private set; }
  public string Callee { get; private set; }

  private BiGram() { }

  public BiGram(string caller, string callee)
  {
    Caller = caller;
    Callee = callee;
  }
}
```

Figure 4.9: TraceLab types class

4.2.2 Types

Data types must be registered with TraceLab before they can be used in the workspace. These types must declare a `[WorkspaceType]` attribute so that TraceLab will recognize them as workspace types. Types must also declare a `[Serializable]` attribute so that data may easily be transferred between the workspace, components, and disk. Figure 4.9 shows an example of a user-defined type. It is important to note that any custom types that do not need to be used in the workspace (such as intermediate data used in an algorithm) do not need to be registered with TraceLab. Types libraries must also be placed in a registered types directory and are usually separate in libraries from the components.

Workspace types may also have a custom visualization for inspection after an experiment has run. TraceLab's built-in types all have visualizations that can be accessed by double clicking the entry in the workspace. Custom types may also have their own visualizations, which requires knowledge of the GUI framework of the platform the user is running on.

4.2.3 Languages

.NET languages

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#.

Java

TraceLab comes with the IKVM.NET[3] virtual machine so that developers can create components and types in Java. The main difference between the C# and Java (in terms of implementing components for TraceLab) is that Java uses annotations instead of attributes and does not support properties (*ie.* implicit getters and setters). This second feature is emulated in the Java version of the TraceLab SDK by using explicit getters and setters for those properties. After compiling the Java components, the JAR file is converted to a DLL through IKVM. When called in TraceLab, the Java code is actually run in the IKVM virtual machine.

R

Although tools like R.NET exist for running R code in .NET languages, they impose additional external dependencies on TraceLab and the development environment. In addition, TraceLab has no built-in mechanism for recognizing components writter in R. To address this issue, I have created a lightweight language plugin for R (named RPlugin) that allows R scripts to be run from TraceLab. Component classes are written as normal (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

[3]`http://www.ikvm.net/`

with the TraceLab Component Library described in Chapter 6.

Matlab

The developers of TraceLab have created a Matlab plugin similar to RPlugin that can run Matlab scripts from .NET. As of this writing, the Matlab plugin is not included in any TraceLab distribution nor the TraceLab Component Library described in Chapter 6, but is available from them by request.

4.3 Contributions

As a collaborator of the TraceLab project, I have had many opportunities to contribute directly. Through use and experimentation, I have provided valuable feedback that influenced the direction of development. To date, I have submitted 31 bug reports and feature requests, and was able to contribute directly to TraceLab's code by fixing 5 of them myself. The SEMERU research group and I have worked hard to promote TraceLab's use in top-tier international software engineering conferences. My influence with the project ultimately led TraceLab's developers to allow our group to entirely restructure the component library and include it with future TraceLab releases. My experiences with TraceLab have led to the compilation of this thesis.

Furthermore, many of the papers I have co-authored have implemented their approaches in TraceLab and shared online.

- *TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions* [5] presents tracelab as a tool for performing evaluations in traceability link recovery.

- *Toward actionable, broadly accessible contests in Software Engineering* [4] presents

TraceLab as a tool for organizing and performing contests in traceability link recovery in order to encourage the community to drive new solutions.

- *A TraceLab-Based Solution for Creating, Conducting, and Sharing Feature Location Experiments* [32] presents TraceLab as a tool for expanding TraceLab to new areas of software engineering research.

- *Using Structural Information to Improve IR-based Traceability Recovery* [33] includes an evaluation in TraceLab for analyzing the effects of including structural information in traceability link recovery.

- *Configuring Topic Models for Software Engineering Tasks in TraceLab* [34] presents the TraceLab implementation of a technique for configuring topic models using genetic algorithms.

- *Enhancing Software Traceability By Automatically Expanding Corpora With Relevant Documentation* (submitted to ICSM'13, under review) investigates the effects of expanding software artifacts with API documentation to increase the accuracy of traceability link recovery.

- *Supporting and Accelerating Reproducible Research in Software Maintenance using TraceLab Component Library* (submitted to ICSM'13, under review) is the companion paper to this thesis.

Chapter 5

Surveying the Needs of the Research Community

For TraceLab to be an effective research tool, it must come with a collection of the most popular tools and techniques used in state of the art software engineering research. To evaluate which tools are needed, a survey of publications in top-tier software engineering conferences from the past 10 years reveals the most commonly-used building blocks for experiments in software engineering.

In this chapter I perform a formal mapping study examining the use of common techniques in software engineering research. 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 [35], for example, a review of studies analyzing development effort estimation techniques to see which ones work the best [36]. 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 [37]. 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.

The primary motivation of this mapping study is to analyze the the current state of software engineering research – focusing on the areas that fall within the SEMERU research group's expertise in software evolution and maintenance (SEM) – in order to compile a comprehensive library of tools for use in TraceLab. The following sections of this chapter describe the methodology, primary studies, and results of a systematic mapping study covering representative papers in software engineering research.

5.1 Methodology

I use the systematic mapping process found in Peterson et al. [37] to drive the study. The process consists of five stages: (1) defining research questions, (2) search for papers, (3) screening criteria, (4) classification, and (5) data extraction.

5.1.1 Definition of Research Questions

This section enumerates the research questions I wish to answer with the mapping study. Since the goal is to discover the breadth and usefulness of different techniques in software evolution and maintenance, I formulate the following research questions (RQs):

RQ1. What types of techniques are common to experiments in software evolution and maintenance research?

RQ2. What individual techniques are used across many SEM experiments?

RQ3. How do experiments in SEM research differ across different sub-domains?

RQ1 attempts to identify high-level categories containing groups of techniques designed to perform similar research tasks. RQ2 instead focuses on individual techniques and aims to identify the most common techniques used in experiments in the mapping study. RQ3 is intended to compare and contrast how techniques are used in different high-level research tasks, such as traceability link recovery or feature location.

For the purposes of this thesis, a *technique* is defined as an individual action performed within an approach. An *approach*, therefore, is the collection of techniques that form the main contribution of a paper. Finally, an *evaluation* is composed of the metrics computations, statistical analyses, and comparison techniques used to analyze the performance of an approach.

5.1.2 Conducting the Search

The goal is to identify modern software engineering techniques shared across many experiments in SEM research. Therefore, I begin by searching the last 10 years of top-tier international software engineering conferences (see Table 5.1). As recommended by [35], I also include ”snowballing” discovery - following references to related work.

5.1.3 Screening Criteria

The formost method used for selecting papers in the study was determining whether or not the research in the paper fell under one of the following high-level tasks in software evolution and maintenance research: traceability link recovery, feature location, program comprehension, and duplicate bug report detection. Generally, this can be done by reading the title, abstract, keywords, and introduction. This is the primary criteria for papers to be *included* in the search.

Given the primary research task of constructing a suite of tools to aid in software engineering research, papers were also evaluated based on the difficulty of implementing the paper's techniques in TraceLab. This determination was based on a number of factors, including lack of implementation details, lack of tool avaiblility, or techniques that required user interaction. Furthermore, techniques that required a significant amount of time or resources to reproduce were not considered at this time. This is the primary reason for papers to be *excluded* from the search.

The complete list of papers used in the mapping study can be found in Section 5.2.

5.1.4 Classification

There are two independent levels of classification used in the mapping study. First is the classification of the papers themselves. The papers (and the approaches contained within) are categorized by the high-level SEM tasks they address. These categories form natural boundaries for evaluating RQ3 and show the usefulness of our contributions across different domains.

The second form of classification I wish to investigate is the categorization of similar techniques within software engineering research experiments. For example, a word stemmer, an identifier splitter, and a stopwords remover may all fall into the general category of textual preprocessors. This classification will help to answer RQ1 and RQ2.

5.1.5 Data Extraction

Each paper in the study is analyzed and the results are recorded in a series of tables. Section 5.2 records the papers surveyed, grouped by SEM task. Section 5.3 records the individual techniques found in each paper, grouped by categorization of technique.

5.2 Papers

Table 5.1 shows the distribution of papers from each conference. Tables 5.2, 5.3, 5.4, and 5.5 list the papers surveyed in the mapping study, grouped by software engineering task. Each entry shows the paper's reference number, citation count, and title. Papers within each table are ordered by publication date.

The high-level software engineering tasks under investigation are as follows. **Traceability link recovery** is the process of recovering lost or missing links between software requirements and source code artifacts. This kind of requirements traceability usually defined as "the ability to describe and follow the life of a requirement" [38]. **Program comprehension** involves the ability to understand what is happening in a program's source code. **Feature location** is the ability to identify relevent source code artifacts that implement a specified feature of the software [39]. **Duplicate bug report detection** refers to the practice of evaluating incoming program defect reports to determine whether a pre-existing report that addresses the same problem has already been filed [40]. These topics constitute the areas that I and the rest of the SEMERU research group have extensive knowledge and experience.

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

Table 5.1: Mapping study: conferences

Traceability Link Recovery		
Ref.	Cit.*	Title
[41]	45	A Traceability Technique for Specifications
[42]	21	On the Role of the Nouns in IR-based Traceability Recovery
[43]	57	On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery
[44]	57	Software Traceability with Topic Modeling
[45]	8	Improving IR-based Traceability Recovery Using Smoothing Filters
[46]	1	Combination Approach for Enhancing Automated Traceability
[47]	18	On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery
[33]	NA	Using Structural Information and User Feedback to Improve IR-based Traceability Recovery
[48]	NA	How to Effectively Use Topic Models for Software Engineering Tasks? An Approach Based on Genetic Algorithms
[34]	NA	Configuring Topic Models for Software Engineering Tasks in TraceLab

* Google Scholar, 4/23/2013

Table 5.2: Mapping study: traceability link recovery papers

Program Comprehension		
Ref.	Cit.*	Title
[29]	46	Mining Source Code to Automatically Split Identifiers for Software Analysis
[49]	31	Using Latent Dirichlet Allocation for Automatic Categorization of Software
[50]	22	Supporting Program Comprehension with Source Code Summarization
[51]	3	Using IR Methods for Labeling Source Code Artifacts: Is It Worthwhile?

* Google Scholar, 4/23/2013

Table 5.3: Mapping study: program comprehension papers

Feature Location		
Ref.	Cit.*	Title
[39]	246	An Information Retrieval Approach to Concept Location in Source Code
[52]	92	Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace
[53]	176	Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval
[54]	31	An Exploratory Study on Assessing Feature Location Techniques
[55]	33	On the Use of Relevance Feedback in IR-Based Concept Location
[30]	16	Can Better Identifier Splitting Techniques Help Feature Location?
[56]	4	Clustering Support for Static Concept Location in Source Code
[57]	2	A Comparison of Stemmers on Source Code Identifiers for Software Search
[32]	4	A TraceLab-Based Solution for Creating, Conducting, and Sharing Feature Location Experiments
[58]	4	Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software

* Google Scholar, 4/23/2013

Table 5.4: Mapping study: feature location papers

Duplicate Bug Detection		
Ref.	Cit.*	Title
[40]	144	Detection of Duplicate Defect Reports Using Natural Language Processing
[59]	150	An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information
[60]	4	A comparative study of the performance of IR models on duplicate bug detection

* Google Scholar, 4/23/2013

Table 5.5: Mapping study: duplicate bug detection papers

5.3 Results

Tables 5.6, 5.7, 5.8, 5.9, and 5.10 list the individual techniques found in each paper, grouped by the categorization of the technique. Each entry is comprised of the paper's reference number and marks indicating that the approach in the paper uses a technique. Papers are ordered by publication year within each software engineering task.

5.3.1 Technique categorization

The mapping study identified five different high-level categories of techniques: data preprocessors, artifacts comparison techniques, results postprocessors, metrics calculations, and a category simply known as "miscellaneous."

Preprocessors Data preprocessing techniques primarily convert the raw data into a different form that will be usable by other techniques in the approach. For text-based approaches, this could involve extracting comments and identifiers from source code, removing stopwords, and other methods of text manipulation. For structural approaches, this could involve parsing an execution trace or calculating a static dependency graph. These techniques usually run *before* the main bulk of the approach.

Artifacts comparison A majority of approaches involve some 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, which is useful for ordering the suggestions based on how strong the score is.

Postprocessors Some techniques build upon the results of a comparison technique and further modify the suggested links between artifacts. They may take into acccount additional information to promote certain links or perform some kind of link pruning to remove false positives.

Metrics Metrics are the measures by which an approach is evaluated. Without this, it would be impossible to determine whether a given approach was useful or not. In order to perform an informative evaluation, the same metrics must be run for different approaches, otherwise the evaluations are not comparing the same thing. Metrics are generally not part of an approach, but are used to perform evaluations between approches.

Miscellaneous These techniques do not fall into any clear category. From the techniques identified in this survey, these techniques are comprised of either (a) complex combinations of different techniques, or (b) techniques used for comparison purposes only (*ie.* not part of the approach).

5.3.2 Analysis

The categorization of techniques given in Section 5.3.1 addresses **RQ1**. To analyze **RQ2**, the Tables 5.6-5.10 of techniques include counts for how many approaches implement that technique. From these results, it can be seen that every single approach uses some kind of textual processing techniques. This makes sense, because software engineering approaches primarily operate on source code, documentation, and other text-based data. More than half of approaches implement stopwords removers, term stemmers, and identifier splitters. Seven approaches incorporate some kind of structural information in their approach, using dependency graphs from either execution traces or static analysis. 88% of approaches use Vector Space Model, Latent Semantic Indexing, or both.

In terms of evaluating an approach, there is a clear distinction between metrics used in different SEM domains. In the areas of traceability link recovery and duplicate bug report detection, every single approach uses some form of precision and recall metrics, although they may include additional metrics in their evaluation. 60% of feature location approaches implement the effectiveness measure metric given in Poshyvanyk et al [53]. Only 30% of papers present some form of statistical analysis in their evaluation. Note that "statistical analysis" is a general term covering a broad range of tests to determine the statistical improvement of an approach. I include them here as a single technique to investigate how many papers include statistical tests. These observations address **RQ3**.

Preprocessors																		
Ref.	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	Regular expressions	Key phrase extractor	Paice stemmer	Kstem	Mstem	SameCase splitter	program language parser	thesaurus matching
[41]	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[42]	✓	✓	✓	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.	
[43]	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[44]	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[45]	✓	✓	✓	✓	.	.	.	✓	.	.	.	.	.	.	.	.	.	
[46]	✓	.	.	.	.	.	.	.	.	.	✓	✓	.	.	.	.	.	.
[47]	✓	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[33]	✓	✓	.	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.
[48]	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[34]	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[29]	✓	.	.	✓	.	.	✓	.	.	.	.	.	.	.	.	✓	.	.
[49]	✓	✓	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[50]	✓	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	✓	.
[51]	✓	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[39]	✓	.	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[52]	✓	✓	.	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
[53]	✓	✓	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[54]	✓	.	.	.	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.
[55]	✓	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
[30]	✓	✓	✓	✓	✓	.	✓	.	.	.	.	.	.	.	.	.	.	.
[56]	✓	✓	✓	✓	.	✓	.	.	.	.	.	.	.	.	.	.	.	.
[57]	✓	.	✓	.	.	.	.	.	✓	.	.	.	✓	✓	✓	.	.	.
[32]	✓	✓	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
[58]	✓	✓	✓	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.
[40]	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	✓
[59]	✓	✓	✓	.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
[60]	✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	27	20	17	15	6	4	2	1	1	1	1	1	1	1	1	1	1	1

Table 5.6: Mapping study: preprocessing techniques

Artifact Comparisons											
Ref.	Latent Semantic Indexing	Vector Space Model	Latent Dirichlet Allocation	Jensen-Shannon divergence	Relational Topic Model	HITS	PageRank	Probablistic LSI	Sufficient Dimensionality Reduction	Scenario-based Probabilistic Ranking	Random Projection
[41]	✓	✓	.	✓	.	.	.	✓	✓	.	.
[42]	✓	.	.	✓	.	.	.	.	.	.	.
[43]	✓	✓	✓	✓	.	.	.	.	.	.	.
[44]	✓	.	✓	.	.	.	.	.	.	.	.
[45]	✓	✓	.	.	.	.	.	.	.	.	.
[46]	.	✓	.	.	.	.	.	.	.	.	.
[47]	.	✓	.	✓	✓	.	.	.	.	.	.
[33]	.	✓	.	✓	.	.	.	.	.	.	.
[48]	.	.	✓	.	.	.	.	.	.	.	.
[34]	.	.	✓	.	.	.	.	.	.	.	.
[29]	.	.	.	.	.	.	.	.	.	.	.
[49]	.	.	✓	.	.	.	.	.	.	.	.
[50]	✓	.	.	.	.	.	.	.	.	.	.
[51]	✓	✓	✓	.	.	.	.	.	.	.	.
[39]	✓	.	.	.	.	.	.	.	.	.	.
[52]	✓	.	.	.	.	.	.	.	.	✓	.
[53]	✓	.	.	.	.	.	.	.	.	✓	.
[54]	✓	.	.	.	.	.	.	.	.	.	.
[55]	.	✓	.	.	.	.	.	.	.	.	.
[30]	✓	.	.	.	.	.	.	.	.	.	.
[56]	.	✓	.	.	.	.	.	.	.	.	.
[57]	.	✓	.	.	.	.	.	.	.	.	.
[32]	✓	✓	.	.	.	.	.	.	.	.	.
[58]	✓	.	.	.	.	✓	✓	.	.	.	.
[40]	.	✓	.	.	.	.	.	.	.	.	.
[59]	.	✓	.	.	.	.	.	.	.	.	.
[60]	✓	✓	✓	.	.	.	.	.	.	.	✓
	15	14	7	5	1	1	1	1	1	2	1

Table 5.7: Mapping study: artifact comparison techniques

Postprocessors							
Ref.	Execution trace extractor	Affine transformation	O-CSTI	UD-CSTI	K-means clustering	combination hueristics	Rocchio Relevence Feedback
[41]	.	.	.	.	.	.	.
[42]	.	.	.	.	.	.	.
[43]	.	.	.	.	.	.	.
[44]	.	.	.	.	.	.	.
[45]	.	.	.	.	.	.	.
[46]	.	.	.	.	✓	.	.
[47]	.	✓	.	.	.	.	.
[33]	.	.	✓	✓	.	.	.
[48]	.	.	.	.	.	.	.
[34]	.	.	.	.	.	.	.
[29]	.	.	.	.	.	.	.
[49]	.	.	.	.	.	.	.
[50]	.	.	.	.	.	.	.
[51]	.	.	.	.	.	.	.
[39]	.	.	.	.	.	.	.
[52]	✓	✓	.	.	.	.	.
[53]	.	✓	.	.	.	.	.
[54]	✓	.	.	.	.	.	.
[55]	.	.	.	.	.	.	✓
[30]	✓	.	.	.	.	.	.
[56]	.	.	.	.	.	.	.
[57]	.	.	.	.	.	.	.
[32]	✓	.	.	.	.	.	.
[58]	✓	.	.	.	.	.	.
[40]	.	.	.	.	.	.	.
[59]	.	.	.	.	.	✓	.
[60]	.	.	.	.	.	.	.
	5	3	1	1	1	1	1

Table 5.8: Mapping study: postprocessing techniques

	Metrics								
Ref.	Precision / Recall metrics	statistical analysis	Effectiveness Measure	Principal Component Analysis	link overlap metrics	Jaccard overlap	Cliff's delta	ROC curve	Pyramid score
[41]	✓	.	.	.	.	.	.	.	.
[42]	✓	✓	.	.	.	.	.	.	.
[43]	✓	.	.	✓	✓	.	.	.	.
[44]	✓	.	.	.	.	.	.	.	.
[45]	✓	✓	.	.	.	.	✓	.	.
[46]	✓	.	.	.	.	.	.	.	.
[47]	✓	✓	.	✓	✓	.	.	.	.
[33]	✓	.	.	.	.	.	.	.	.
[48]	✓	✓	✓	.	.	✓	.	.	.
[34]	✓	.	.	.	.	.	.	.	.
[29]	.	.	.	.	.	.	.	.	.
[49]	✓	.	.	.	.	.	.	.	.
[50]	.	.	.	.	.	.	.	.	✓
[51]	.	.	.	.	.	✓	.	.	.
[39]	✓	.	.	.	.	.	.	.	.
[52]	.	.	✓	.	.	.	.	.	.
[53]	.	.	✓	.	.	.	.	.	.
[54]	.	.	.	.	.	.	.	.	.
[55]	.	.	.	.	.	.	.	.	.
[30]	.	✓	✓	.	.	.	.	.	.
[56]	.	✓	✓	.	.	.	.	.	.
[57]	.	.	.	.	.	.	.	✓	.
[32]	.	✓	✓	.	.	.	.	.	.
[58]	.	✓	✓	.	.	.	.	.	.
[40]	✓	.	.	.	.	.	.	.	.
[59]	✓	.	.	.	.	.	.	.	.
[60]	✓	.	.	.	.	.	.	.	.
	15	8	7	2	2	2	1	1	1

Table 5.9: Mapping study: metrics techniques

Miscellaneous						
Ref.	Genetic Algorithm	prospective approach	BorderFlow	RIPPLES	grep	MUDABlue
[41]	.	.	.	.	.	.
[42]	.	.	.	.	.	.
[43]	.	.	.	.	.	.
[44]	.	✓	.	.	.	.
[45]	.	.	.	.	.	.
[46]	.	.	.	.	.	.
[47]	.	.	.	.	.	.
[33]	.	.	.	.	.	.
[48]	✓	.	.	.	.	.
[34]	✓	.	.	.	.	.
[29]	.	.	.	.	.	.
[49]	.	.	.	.	.	✓
[50]	.	.	.	.	.	.
[51]	.	.	.	.	.	.
[39]	.	.	.	✓	✓	.
[52]	.	.	.	.	.	.
[53]	.	.	.	.	.	.
[54]	.	.	.	.	.	.
[55]	.	.	.	.	.	.
[30]	.	.	.	.	.	.
[56]	.	.	✓	.	.	.
[57]	.	.	.	.	.	.
[32]	.	.	.	.	.	.
[58]	.	.	.	.	.	.
[40]	.	.	.	.	.	.
[59]	.	.	.	.	.	.
[60]	.	.	.	.	.	.
	2	1	1	1	1	1

Table 5.10: Mapping study: miscellaneous techniques

Chapter 6

Component Library and Development Kit

TraceLab provides extensibility to users through a software development kit (SDK) that enables them to create new components for use in TraceLab experiments. I leverage this ability in order to extend TraceLab's component library with many common tools and techniques used in software engineering research that are not included in the base distribution. From the papers and techniques identified in the mapping study (Chapter 5), I implement a comprehensive library of components and techniques with the goal of assisting researchers and developers by providing them with the tools they need to jump start their research. Where possible, I incorporated existing TraceLab functionality, SEMERU research tools, and implementations of open source software. When this was not possible, I implemented the tool from scratch to the best of my ability based on the description in the paper. With the approval of the developers of TraceLab, the new component library has been incorporated into the base distribution of TraceLab.

In this chapter I provide (6.1) the details of a Component Development Kit

(CDK) that contains implementations of the techniques found in the study, as well as many other useful tools; (6.2) a Component Library (CL) that contains wrapper classes for the CDK techniques to be used in TraceLab as components; (6.3) links to online documentation and usage examples; (6.4) an invitation to other developers to extend the component library; and (6.5) an analysis of the coverage of the approaches in the mapping study when using the CL in TraceLab.

6.1 Component Development Kit

The Component Development Kit (CDK) is a library of commonly-used tools and techniques in software evolution and maintenance research. These tools are organized in a well-defined hierarchy and exposed through a public API.

The CDK is separated into high-level tasks. These tasks include data I/O, preprocessing techniques, artifact tracing techniques, postprocessing techniques, metrics calculations, and common utilities (see Figure 6.1). These namespaces are then further broken down into more specific granularity for the desired task. For example, Metrics computations are broken down by SEM domain, such as *traceability* or *feature location*, and tracing techniques are broken down into information retrieval (IR), topic models, and web mining algorithms. This design aids component developers in locating relevant functionality quickly and easily, as well as providing a framework for including new techniques in the future.

Each technique in the mapping study was evaluated based on paper coverage, utility, and perceived difficulty in implementation. Some of the techniques that appear in more than one paper were left out due to various reasons, such complexity, numerous dependencies, and lack of resources. For as many missing techniques as possible, I tried to make sure that the expected output of the technique could be easily imported into TraceLab. Component developers should find that the CDK contains many of the tools necessary to aid them in creating a missing technique. Any new techniques that developers create and wish to incorporate into the CDK can easily be added (see Section 6.4).

The following paragraphs detail the functionality and tasks of each namespace. Figure 6.1 shows an overview of the CDK structure in relation to the Component Library and TraceLab.

I/O The I/O namespace delegates the task of importing and exporting the datatypes used in TraceLab to and from storage locations. Data may be stored and retrieved in multiple formats.

Preprocessors The preprocessors namespace deals with transforming raw data into something usable for techniques further in the experiment. This level is further broken down into stemmers, identifier splitters, and execution trace analyzers.

Artifacts Comparison The artifacts comparison namespace (which is currently "Tracers" in the CDK) contains algorithms which compute relationships between different software artifacts. Generally, these techniques take in one of the standard TraceLab datatypes (`TLArtifactsCollection`) and produce a set of candidate links between artifacts in a `TLSimilarityMatrix`. This level is further broken down into information retrieval, topic modeling, and web mining techniques.

Postprocessors The postprocessors namespace is comprised of algorithms that modify the results of a `TLSimilarityMatrix` based on additional information. Link pruning algorithms are not included here; instead, they are present in the `TLSimilarityMatrixUtils` utility class in the utility namespace.

Metrics The metrics namespace has a special inheritance heirarchy in the CDK. Individual metrics computations (such as precision, recall, and F-measure) must inherit from the abstract class `MetricComputation`, which forces them to define a method of providing both fine-grained and summary results. This structure is beneficial for use in the Results Visualization component (see Section 6.2). This level is broken down into common software engineering task, such as traceability and feature location metrics.

Utilities This namespace contains utility classes that perform common programming tasks related to individual data types.

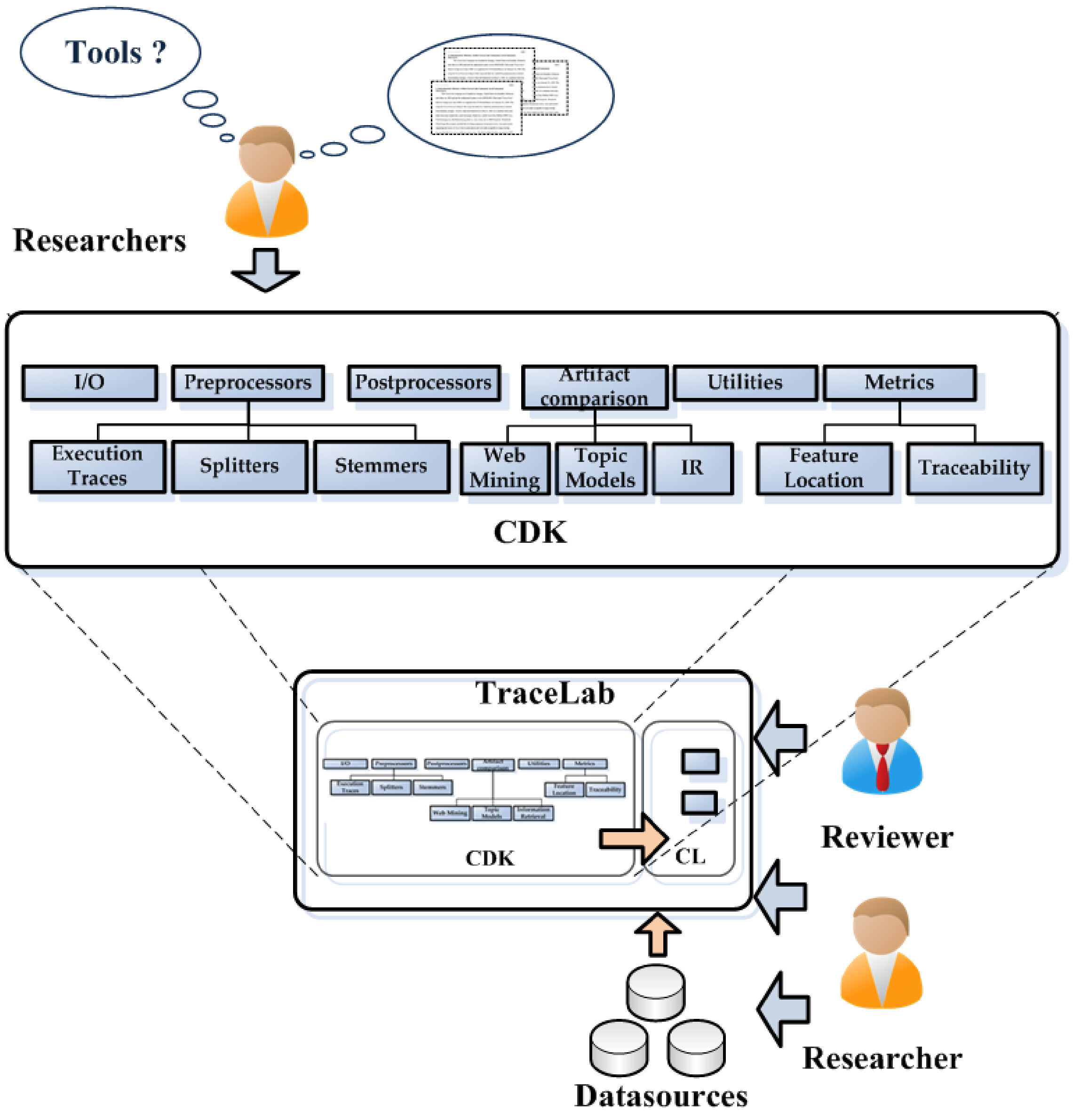


Figure 6.1: Visualization of component library hierarchy

6.2 Component Library

The Component Library (CL) is composed of the component classes and metadata described in Section 4.1.2. It acts a layer between TraceLab and the CDK, adapting the functionality of the CDK to be used within TraceLab. A typical component will import data from the workspace, make calls to the CDK, and then output the results. The structure of the Component Library mirrors the CDK hierarchy, providing a mapping from TraceLab to the CDK. Components appear in TraceLab's component library viewer grouped by tags into the same high-level tasks as the CDK.

The CL is not a 1-to-1 mapping from the CDK. For example, the I/O namespace in the CDK is split into individual importers and exporters in the component library. In addition to the major functionality of the CDK, the CL includes a number of helper components that assisst in programmatic operations within an experiment, such as incrementing a loop counter or retrieving a string from a list.

Another addition present in the CL is the metrics storage engine. Previously, the vast number of metrics computations was creating increasingly crowded experiments and workspace entries. The storage engine provides a single point of entry for storing the results of metrics computations, organized by technique and dataset. The engine contains functionality for retrieving fine-grained results and automatically generating summary statistics for use in the Results Visualization component.

6.3 Documentation

Documentation is often overlooked, both in source code and overall design. In addition to code examples and API references, documentation provides vital information about a program's functionality, design, and intended use. An important contribution to this project is thorough documentation about the CL and CDK in order to assist new users in learning about TraceLab and help them in developing their own components and tools. This adds a wealth of knowledge to someone who wants to use TraceLab and start designing new experiments from components. Documentation can be found online for the general TraceLab wiki[1] and the SEMERU wiki[2].

6.4 Extending the CL and CDK

The CL and CDK do not contain all of the tools that researchers will ever need. However, their design and implementation alongside TraceLab's development framework provide a firm foundation for supporting future research. The CL and CDK is released under an open source license[3] in order to facilitate collaboration and community contribution. As new techniques are created, they can be added to the existing framework and thus into TraceLab. TraceLab's developers and the SEMERU research group encourage all contributions to the project. See Section 4.2 for more information on extending the TraceLab component library.

6.5 Coverage

The CL and CDK was implemented on a subset of the techniques identified in the mapping study, based on their popularity and the amount of resources I had at the time. As such, the CL and CDK contains implementations of 23 of the 51 (45%) techniques identified in the mapping study. Looking at only techniques involved in

[1] `http://coest.org/coest-projects/projects/tracelab/wiki`
[2] `http://coest.org/coest-projects/projects/semeru/wiki`
[3] `http://www.gnu.org/licenses/gpl.txt`

an approach (*ie.* excluding metrics and comparison techniques), the CL and CDK contains implementations of 20 out of 39 (51%) techniques. Tables 6.1, 6.2, and 6.3 record the techniques in the mapping study that are implemented in the CL and CDK.

Using the CL and CDK, it is possible to completely reproduce 10 of the 27 (37%) approaches in the mapping study. Of the remaining approaches, 10 of them (58%, 37% overall) are missing only 1 technique. Table 6.4 shows the breakdown of implemented techniques of the approaches in the mapping study.

Name	#	In TraceLab?
Bag-of-words tokenizer	27	✓
Stopwords Remover	20	✓
Porter stemmer	17	✓
CamelCase splitter	15	✓
Latent Semantic Indexing	15	✓
Vector Space Model	14	✓
Latent Dirichlet Allocation	7	✓
Execution trace logger	6	.
Execution trace extractor	5	✓
Jensen-Shannon divergence	5	✓
Dependency Graph Generator	4	✓
Affine transformation	3	✓
Genetic Algorithm	2	✓
Samurai splitter	2	.
Scenario-based Probabilistic Ranking	2	.
BorderFlow	1	.
combination hueristics	1	.
HITS	1	✓
Key phrase extractor	1	.
K-means clustering	1	.
Kstem	1	.
Mstem	1	.
O-CSTI	1	✓
PageRank	1	✓
Paice stemmer	1	.
Part-of-speech tagger	1	✓
Probablistic LSI	1	.
program language parser	1	.
prospective approach	1	.
Random Projection	1	.
Regular expressions	1	.
Relational Topic Model	1	✓
Rocchio Relevence Feedback	1	.
SameCase splitter	1	.
smoothing filter	1	✓
Snowball Stemmer	1	✓
Sufficient Dimensionality Reduction	1	.
thesaurus matching	1	.
UD-CSTI	1	✓

Table 6.1: List of approach techniques

Name	**#**	**In TraceLab?**
Precision / Recall metrics	15	✓
statistical analysis*	8	.
Effectiveness Measure	7	✓
Jaccard overlap	2	.
link overlap metrics	2	.
Principal Component Analysis	2	✓
Cliff's delta	1	.
Pyramid score	1	.
ROC curve	1	.

* Due to the large number of different statistical tests available, I do not include them in the library but provide functionality for exporting the results to be analyzed offline.

Table 6.2: List of metrics techniques

Name	**#**	**In TraceLab?**
grep	1	.
MUDABlue	1	.
RIPPLES	1	.

Table 6.3: List of comparison techniques

Ref.	**Impl.**	**Missing**	**Total**	**%**
[41]	6	2	8	0.75
[42]	**7**	**0**	**7**	**1.00**
[43]	**7**	**0**	**7**	**1.00**
[44]	5	1	6	$0.8\overline{3}$
[45]	**7**	**0**	**7**	**1.00**
[46]	2	3	5	0.40
[47]	**8**	**0**	**8**	**1.00**
[33]	**7**	**0**	**7**	**1.00**
[48]	**3**	**0**	**3**	**1.00**
[34]	3	**0**	3	**1.00**
[29]	2	2	4	0.50
[49]	**4**	**0**	**4**	**1.00**
[50]	5	1	6	$0.8\overline{3}$
[51]	**7**	**0**	**7**	**1.00**
[39]	**3**	**0**	**3**	**1.00**
[52]	6	2	8	0.75
[53]	5	1	6	$0.8\overline{3}$
[54]	4	1	5	0.80
[55]	5	1	6	$0.8\overline{3}$
[30]	6	2	8	0.75
[56]	6	1	7	0.86
[57]	4	3	7	0.57
[32]	7	1	8	0.88
[58]	9	1	10	0.90
[40]	4	1	5	0.80
[59]	4	2	6	$0.6\overline{6}$
[60]	6	1	7	0.88

Table 6.4: Percentage of approaches that can be reproduced with CL & CDK

Chapter 7

Reproducing Software Engineering Research

This chapter presents examples of recreating existing software engineering approaches using the Component Library and TraceLab. Each approach is summarized and the results in TraceLab are compared to the original results, where applicable. Section 7.1 describes studies in traceability link recovery that compare information retrieval approaches. Section 7.2 describes the evolution of feature location approaches using Latent Semantic Indexing. Finally, Section 7.3 provides additional examples of approaches that can be reproduced in TraceLab. These approaches are made available online[1] as TraceLab experiments.

[1] `http://www.cs.wm.edu/semeru/TraceLab_CDK`

7.1 Investigating Information Retrieval techniques for software traceability

This section presents existing research involved with investigating different information retrieval (IR) approaches in the area of traceability link recovery. Section 7.1.1 presents background information regarding traceability link recovery. Section 7.1.2 presents a comparison of IR techniques from Abadi et al [41]. Section 7.1.3 presents a study in the equivelence of different IR methods by Oliveto et al [43]. Finally, Section 7.1.4 leverages the findings of Oliveto et al. to investigate combining complementary IR techniques from Gethers et al [47].

7.1.1 Background of software traceability

A quote by O.C.Z. Gotel is often used to define requirements traceability. In her paper [38] she states,

> Requirements traceability refers to the ability to describe and follow the life of a requirement, in both forwards and backwards direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases).

In practical terms, this involves tracking traceability links at all stages of development. The most common research involves recovering missing or broken links towards the end of a project's life cycle. Since this is extremely difficult to perform by hand – especially in large software projects with tens of thousands of requirements – automated solutions are desired.

When investigating traceability links, the artifacts under consideration are called *source* and *target artifacts.* Source artifacts (also referred to as *queries*) are usually the high-level documents that need to be linked to source code, such as requirements or use case documents. Target artifacts are source code documents split into the desired level of granularity. For example, if a developer wanted to trace requirements to specific code classes, he or she could direct the traceability technique to generate a corpus of artifacts by extracting each class and saving it as a separate document. The same could be done for method- or package-level artifacts.

The output of a traceability technique is a set of ordered links called a *ranked-list*. The list is sorted in decreasing order based on the technique's confidence level that the candidate link is a true link. The links at the top of the list have a strong confidence level and indicate to the developer that there may be a link between the two artifacts. The developer can then investigate the suggested target artifacts instead of searching the entire source code repository for possible links.

Software engineering researchers need to be able to evaluate and compare their techniques. Therefore, they perform studies on software projects that have well-documented links between requirements and code. These links are used as an *oracle* for evaluating the output of a new technique. The most common metrics for this evaluation are *precision* and *recall*, which are described in the Vector Space Model example in Chapter 3. Plotting a *precision-recall curve* at each level of recall is a common method of displaying the results of a technique.

7.1.2 A Traceability Technique for Specifications

Abadi et al. [41] performed an evaluation of 5 different information retrieval techniques for the purposes of comparing the results of each technique for traceability link recovery. These techniques consisted of the Vector Space Model (VSM), Latent Semantic Indexing (LSI), Probabilistic LSI (PLSI), Sufficient Dimensionality Reduction (SDR), and Jensen-Shannon similarity (JS). Each technique is experimented with different weighting schemes and parameters. The authors performed an evaluation on two datasets: SCARI-OPEN, from Communications Research Centre

Canada [61], and the GNU Classpath implementation of CORBA [62]. The authors presented the metrics precision, recall, and mean average precision. They concluded that VSM (tf-idf weighted) and JS (with information-gain) perform the best overall.

Only three of these techniques were implemented in the Component Library. As such, PLSI and SDR cannot be reproduced at this time. VSM is described in the motivating example in Chapter 3. LSI [63] is an extension of VSM that performs Singular Value Decomposition to decompose the original term-by-document matrix into three reduced matrices. Documents are compared in this reduced space by cosine similarity. JS similarity was introduced in this paper as a useful information retrieval technique for traceability link recovery. JS treats documents as probability distributions and measures the distance between them.

The two datasets used in this evaluation were not available to me at the time of this writing. Instead, I perform an evaluation of the three IR techniques on eTour, an electronic tourist guide used in the 2011 TEFSE Challenge [64]. Figure 7.2 shows the experiment as it appears in TraceLab. The top portion of the graph involves importing the dataset and performing various preprocessing techniques. The three components in the middle (Vector Space Model, Jensen-Shannon divergence, and Latent Semantic Analysis) represent the three IR techniques under study. The bottom half of the graph involves importing the oracle and computing the results of the techniques, ending in the results visualization GUI.

Figure 7.1 and Table 7.1 show the results of the evaluation. While VSM is the top performer, JS is outstripped by LSI. This may be due to a variety of factors, such as using different datasets and internal weighting schemes. This example shows some of the problems with reproducing research in software engineering.

Technique	MAP
LSI	0.379
JS	0.294
VSM	0.417

Table 7.1: Abadi et al. Mean average precision of evaluation in TraceLab

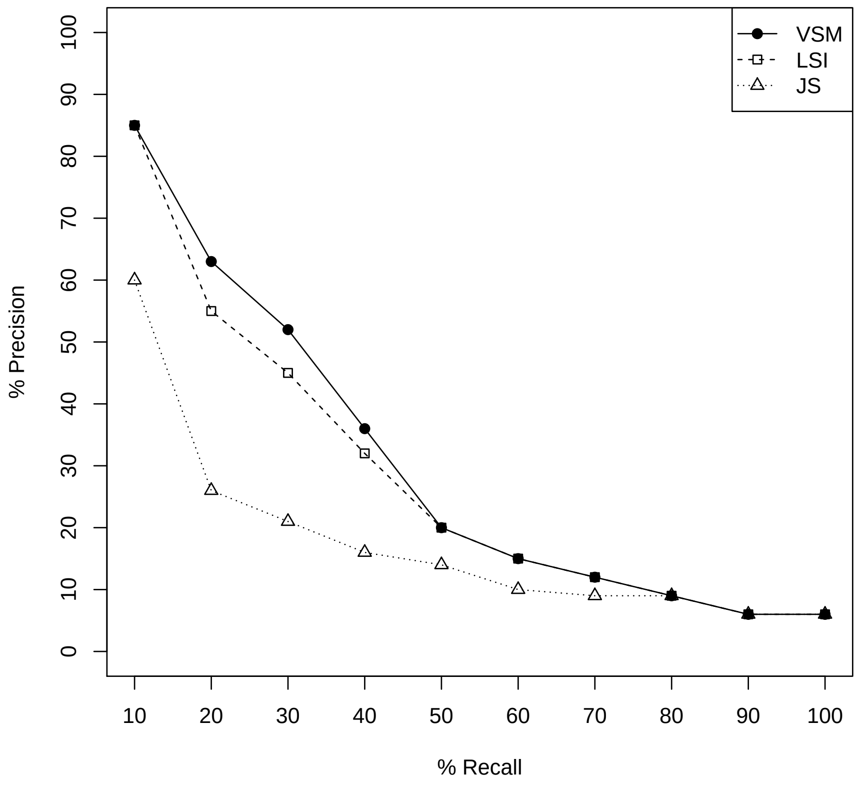


Figure 7.1: Abadi et al. Precision-recall curve of evaluation in TraceLab

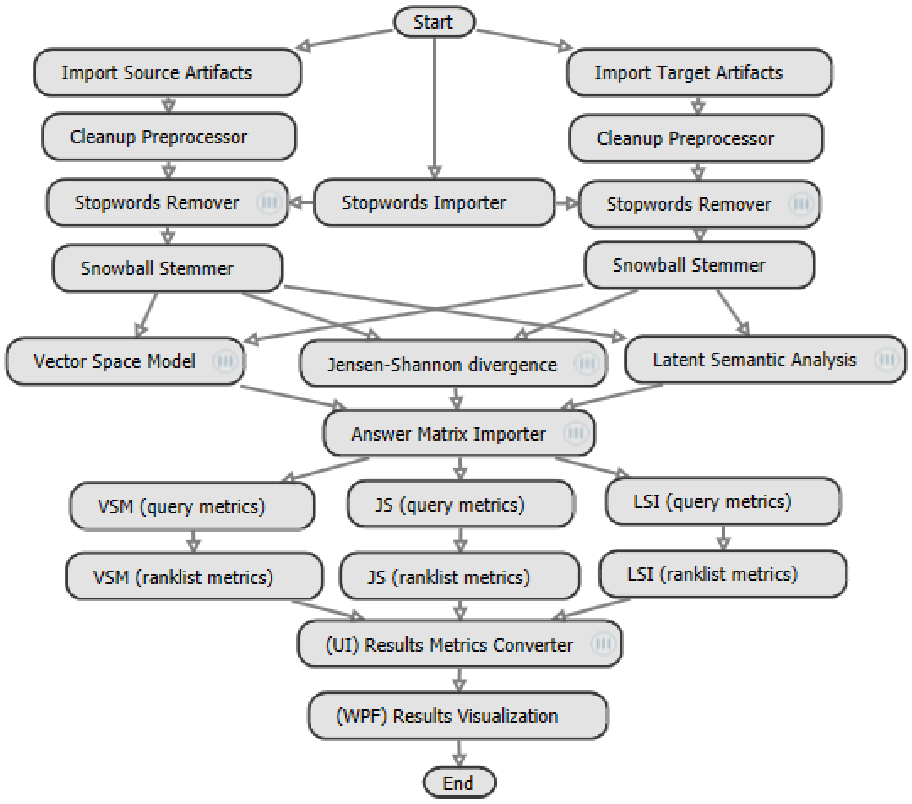


Figure 7.2: Abadi et al. TraceLab experiment

7.1.3 On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery

Oliveto et al. [43] investigated different IR methods to see if they produced equivalent results, namely, VSM, LSI, JS, and a topic modeling technique called Latent Dirichlet Allocation (LDA) [44]. They performed an evaluation on two datasets, EasyClinic and eTour. They reported precision and recall of the results in addition to link overlap metrics. They performed Principal Component Analysis (PCA) to determine which techniques are equivelent in terms of performance in traceability link recovery. The authors found that VSM, LSI, and JS are equivelent, while LDA provides orthogonal results.

This paper contains one of the approaches from the mapping study that can be entirely reproduced in TraceLab. In addition, the same datasets used in the evaluation were available to me. Figure 7.3 shows the experiment in TraceLab. The graph was modified from the experiment in the previous section by adding an LDA component, a PCA component, and an additional metrics calculation component for LDA.

Figure 7.4 and Table 7.2 show the results of the evaluation. The precision-recall curve shows that LDA as configured does not perform as well as the other three IR methods. Table 7.2a shows the results of PCA as I assumed it was configured - VSM with tf-idf weights, and 4 principal components. The paper states they use tf-idf weighting for VSM, and the script to calculate PCA was provided to me by one of the authors. As the table shows, each IR technique is correlated with its own PC. Then I surmised that the weighting scheme for VSM could be different, so I ran it again with no weight, resulting in Table 7.2c. It showed that VSM and JS were equivelent, but LSI was still correlated with its own PC. I realized that the paper only reports 2 PCs, so I modified the PCA computation to calculate only 2 PCs, resulting in Tables 7.2b and 7.2d. In both cases they show that VSM, JS, and LSI are highly correlated with PC1, and LDA is correlated with PC2. This is consistent with the results in the paper. The proportion of variance of each component is displayed at the bottom of each table.

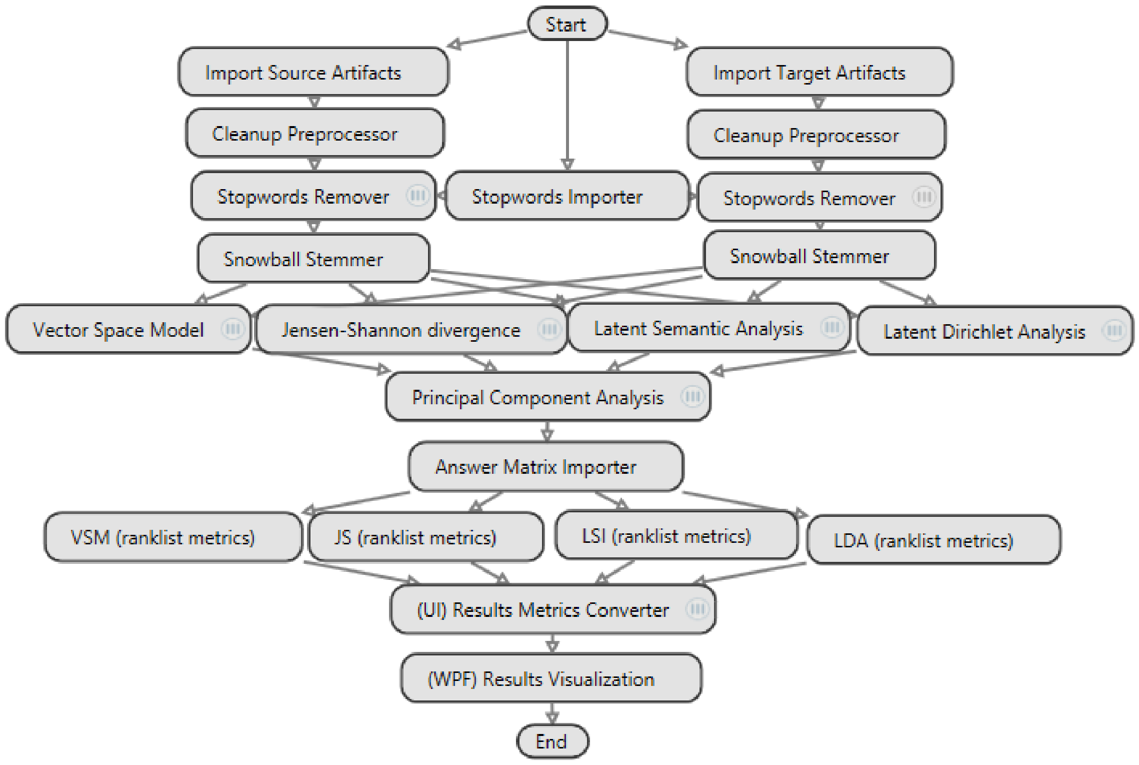


Figure 7.3: Oliveto et al. TraceLab experiment

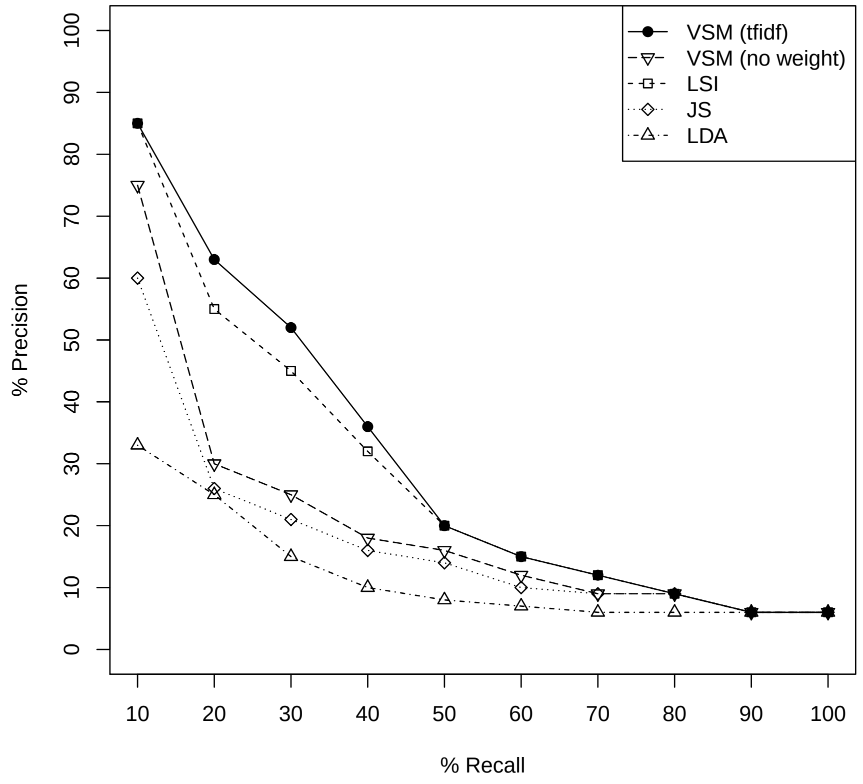


Figure 7.4: Oliveto et al. Precision-recall curve of evaluation in TraceLab

	PC1	PC2	PC3	PC4
VSM	0.55	0.23	0.33	**0.73**
JS	0.27	0.19	**0.92**	0.21
LSI	**0.88**	0.22	0.29	0.31
LDA	0.19	**0.96**	0.18	0.14
% Var.	0.29	0.26	0.27	0.17

(a) VSM (tf-idf), 4 PC

	PC1	PC2
VSM	**0.94**	-0.06
JS	**0.82**	-0.04
LSI	**0.92**	-0.06
LDA	0.56	**0.83**
% Var.	0.68	0.17

(b) VSM (tf-idf), 2 PC

	PC1	PC2	PC3	PC4
VSM	**0.86**	0.20	0.38	0.27
JS	**0.94**	0.20	0.25	-0.13
LSI	0.35	0.23	**0.91**	0.02
LDA	0.20	**0.96**	0.20	0.02
% Var.	0.45	0.26	0.27	0.02

(c) VSM (no weight), 4 PC

	PC1	PC2
VSM	**0.95**	0.22
JS	**0.93**	0.18
LSI	**0.73**	0.41
LDA	0.23	**0.96**
% Var.	0.58	0.29

(d) VSM (no weight), 2 PC

Table 7.2: Oliveto et al. Primary Component Analysis of IR techniques

7.1.4 On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery

Based on the findings of Oliveto et al. [43], Gethers et al. [47] investigated the effects of combining orthogonal IR techniques. In addition to using VSM and JS, the authors introduced Relational Topic Model (RTM) [65] as a traceability link recovery technique. Their approach implemented PCA to determine the level of contribution of each technique, which is then used as a lambda parameter for an affine transformation between pairs of techniques. The authors performed an evaluation on four datasets: EAnci, eTour, EasyClinic, and SMOS. They reported precision, recall, and average precision of the results as well as link overlap metrics. The authors confirmed that VSM and JS are equivelent and find that RTM is orthogonal to the two. The authors found that using the hybrid approach of VSM+RTM and JS+RTM significantly increases the accuracy of traceability link recovery.

This paper contains one of the approaches from the mapping study that can be entirely reproduced in TraceLab. In addition, the same datasets used in the evaluation were available to me. As in the previous sections, I use eTour for the evaluation. Figure 7.7 shows the experiment in TraceLab. The graph was modified from the experiment in the previous section by removing LSI and LDA and adding an RTM component. Additionally, the different combinations of techniques were added with affine transformation components. Finally, the metrics components for each technique were added at the end of the experiment.

Figures 7.6 and 7.5 and Tables 7.3 and 7.4 show the results of the evaluation. From the precision-recall curve, it can be seen that RTM performs about as well as JS and VSM with no weight. Differences begin to appear, however, when investigating the effects of VSM with tf-idf weight, which the paper claims to use. Table 7.3b shows that each IR method is correlated with its own PC when using tf-idf weight for

VSM, while Table 7.3a shows that VSM with no weight and JS are both correlated with PC1, which matches the results published in the paper.

Figures 7.5a and 7.5b show the effects of combining different IR techniques on mean average precision of traceability link recovery when using VSM with no weight and tf-idf weight, respectively. Tables 7.4a and 7.4b show an analysis of these results. When using VSM with no weight, combining VSM and RTM improves the MAP of traceability link recovery by 10% over standalone VSM and 11% over standalone RTM, which is consistent with the results shown in the paper. Similar results are found for combining JS and RTM.

Conversely, when using VSM with tf-idf weight, VSM is the best overall performer. Combining VSM with RTM actually reduces accuracy by 7% over standalone VSM. JS and RTM show great improvement when combined with VSM over their standalone technique because of tf-idf's greater accuracy, but are still not greater than standalone VSM. This can be interpreted as an averaging of two techniques, rather than an information gain from combining orthogonal techniques. What can be concluded from this evaluation is that it is very important to provide all the details and assumptions of an approach in order to ensure reproducibility.

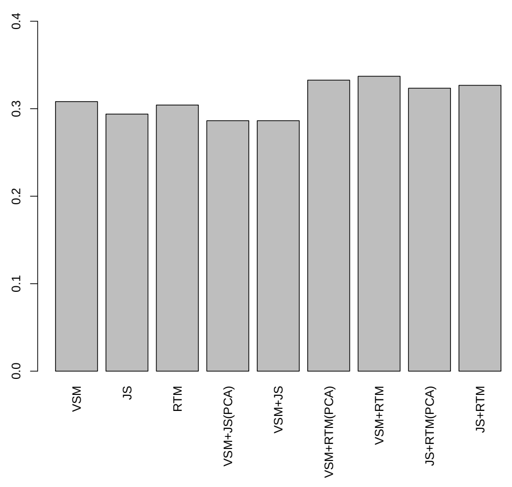
	PC1	PC2	PC3
VSM	**0.97**	0.13	-0.21
JS	**0.97**	0.13	0.21
RTM	0.30	**0.95**	0.00
% Var.	0.65	0.32	0.03

(a) VSM (no weight)

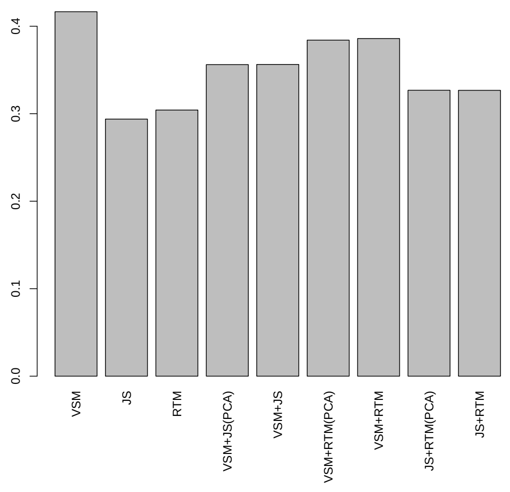
	PC1	PC2	PC3
VSM	**0.91**	0.23	0.34
JS	0.32	0.19	**0.93**
RTM	0.21	**0.96**	0.18
% Var.	0.33	0.34	0.33

(b) VSM (tf-idf)

Table 7.3: Gethers et al. Primary Component Analysis of IR techniques



(a) VSM (no weight)



(b) VSM (tf-idf)

Figure 7.5: Gethers et al. MAP of IR techniques

	VSM	JS	RTM
VSM	-	-7%	**+10%**
JS	-2%	-	**+11%**
RTM	**+11%**	**+7%**	-

(a) VSM (no weight)

	VSM	JS	RTM
VSM	-	-14%	-7%
JS	**+21%**	-	**+11%**
RTM	**+26%**	**+7%**	-

(b) VSM (tf-idf)

Table 7.4: Gethers et al. Effect on MAP of combining IR techniques

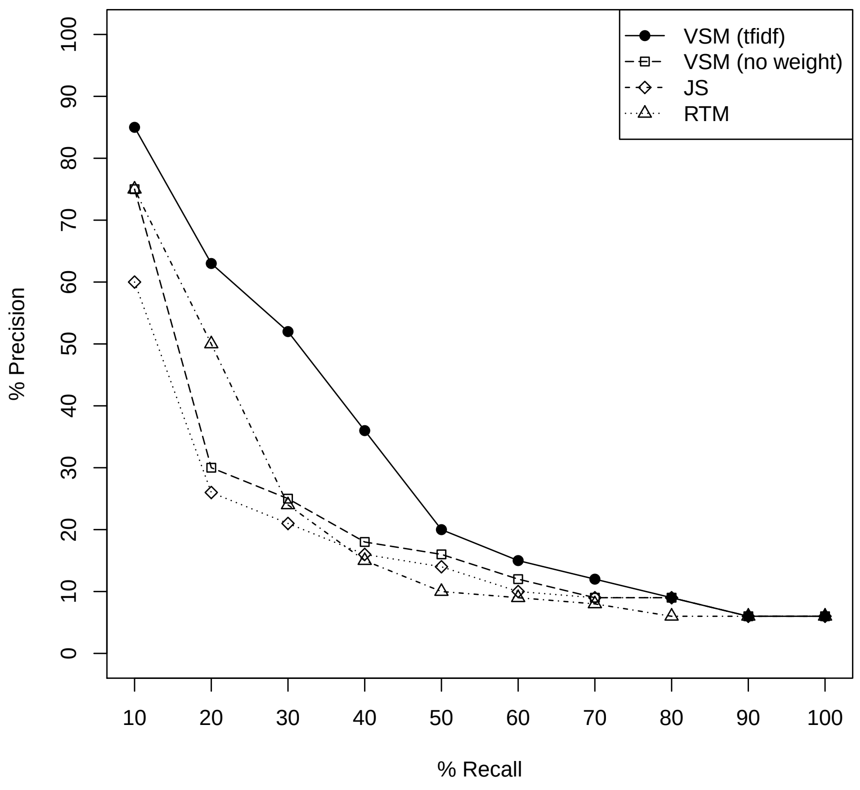


Figure 7.6: Gethers et al. Precision-recall curve of evaluation in TraceLab

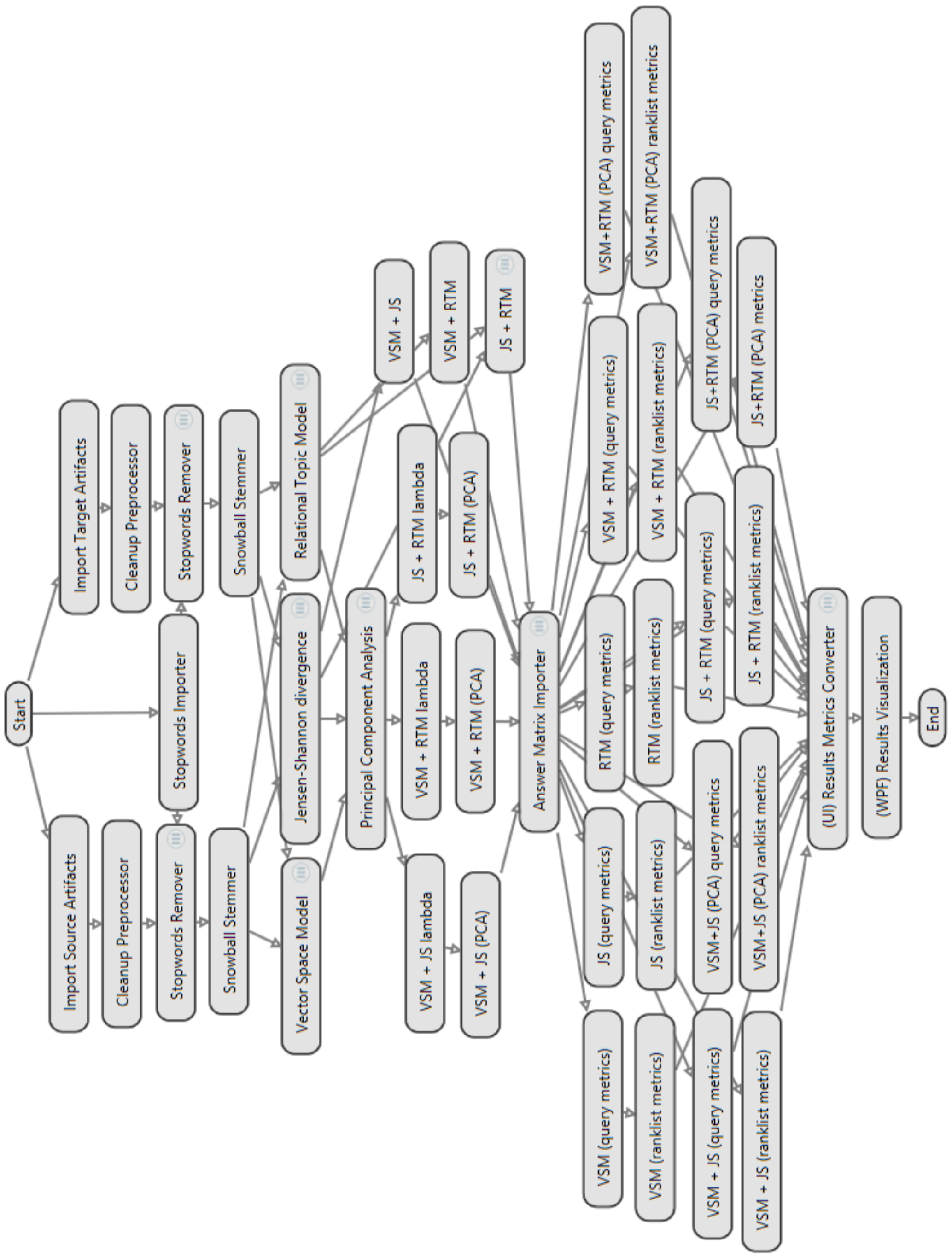


Figure 7.7: Gethers et al. TraceLab experiment

7.2 Applications of Latent Semantic Indexing for feature location in software

This section presents existing research involved with investigating different applications of LSI for feature location. Section 7.2.1 presents background information regarding feature location. Section 7.2.2 presents the use of LSI for feature location by Marcus et al [39]. Section 7.2.3 presents the improvement over standalone LSI by applying dynamic execution trace information by Liu et al [52]. Finally, Section 7.2.4 presents the data fusion technique of combining web mining algorithms with dynamic execution trace information by Dit et al [58].

7.2.1 Background of feature location

Feature location [39] (also known as concept location) involves locating source code elements that implement a certain high-level functionality. For example, if someone wanted to know where a textbox's autocomplete functionality was located in the code, he or she could inspect the code, starting with the textbox declaration and tracing back to a method that calls a database and produces a list of matches. In practice, the number of features and complexity of design in a program make this task extremely difficult and time consuming.

Automated solutions are similar to traceability link recovery, except in this case, the queries are natural language descriptions of a feature and the target documents are the classes or methods that implement the feature. Approaches produce a similar ranklist of similarities. However, a separate ranklist is created for each query and the metrics by which approaches are compared are different. Poshyvanyk et al. [53] introduce the *effectiveness measure*, which is defined as the number of methods that a user has to investigate before locating a relevant method. This is often reported as two separate metrics: *effectiveness best measure*, which is the location of the first relevant link in the results; and *effectiveness all measure*, which is the location of all of the relevant links in the results.

The purpose of this section is to demonstrate the evolution of LSI when used for feature location. In this approach, source code is transformed into a term-by-document matrix alongside queries similar to VSM. The matrix is then decomposed into a product of three other matrices via Singular Value Decomposition, then reconstructed using two of the submatrices to form a least-squares best fit [66]. Documents within this space are then compared via cosine similarity, producing a ranked-list of results.

7.2.2 An Information Retrieval Approach to Concept Location in Source Code

Marcus et al [39]. proposed using LSI as a method for identifying relevant methods for the task of feature location. They performed an evaluation on the Mosaic web browser [67] using user-made and automatically generated queries. They compared their approach to static analysis tools and `grep`, a Unix search utility. They reported the precision and recall of the results, noting the positions of correct links.

The TraceLab implementation of this approach is somewhat different than the original. Firstly, I run the evaluation on jEdit [68], which is a Java-based text editor. Queries were formulated by extracting bug reports, feature requests, and patch summaries from a version control repository. Furthermore, I report the effectiveness measure metric from Poshyvanyk et al[53], as it is used in all future papers. The approach taken by using LSI remains the same.

Figure 7.8 shows the experiment as it appears in TraceLab. The nodes in the top half of the experiment import the data and apply various preprocessing techniques. The original experiment in the paper only mentions basic cleanup and identifier splitting, but I added stopwords removal and a term stemmer to be consistent with the other experiments in this section. After computing LSI, an oracle containing the correct methods is imported and the effectiveness measures are computed. Finally, the results are displayed in a GUI.

Table 7.5 shows the effectiveness best measures and effectiveness all measures of the evaluation in TraceLab. The table shows that the median rank of the first relevant method is 39, while the median of all ranks is 136. Although the first relevant method was often at position 0 (*ie.* the first method returned), a user may have to investigate many incorrect methods before finding a relevant one.

Percentile	**Best**	**All**
Max	4140	5728
Q3	165.25	508.5
Median	39	136
Q1	4.75	37.25
Min	0	0

Table 7.5: Marcus et al. effectiveness measures of TraceLab evaluation

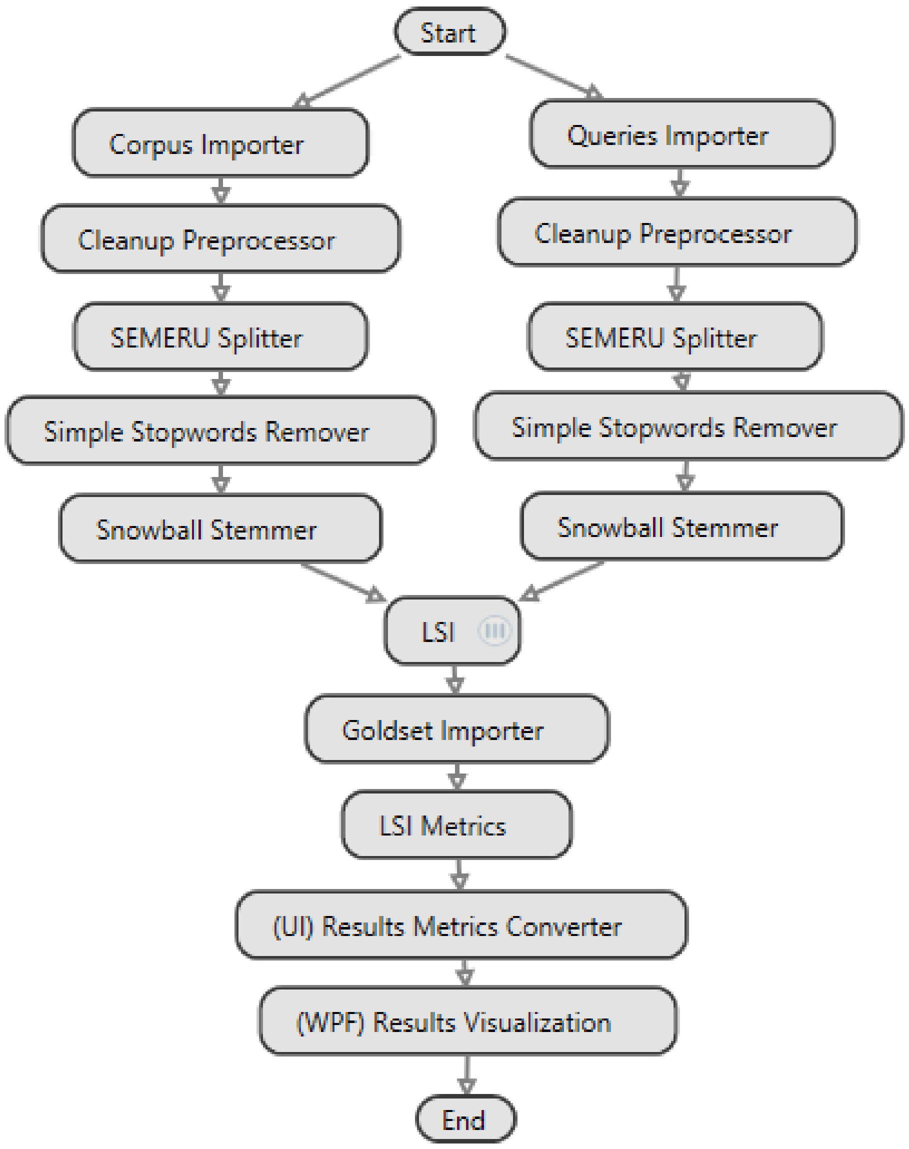


Figure 7.8: Marcus et al. TraceLab experiment

7.2.3 Feature location via information retrieval based filtering of a single scenario execution trace

Liu et al [52] improved on the approach of using LSI for feature location. They surmised that if a code artifact implements a certain feature, then that code will be run when the feature is exercised. Therefore, by recording an execution trace when using a feature, the trace will contain methods relevant to that feature. The resulting ranklist produced by LSI can then be pruned of all the methods that were not present in the trace, moving the relevant methods closer to the top of the list. The authors performed an evaluation on jEdit and Eclipse [69], a Java IDE, and compared their approach to standalone LSI. The authors reported the effectiveness measures of each and concluded that LSI with execution traces significantly improves the effectiveness of feature location.

The TraceLab implementation of this approach uses jEdit, but a different version than the one in the paper. I use the queries from the previous section. Figure 7.9 shows the experiment in TraceLab. This experiment was modified from the experiment in Section 7.2.2. The nodes to the left and right of while loop were in the initial TraceLab implementation. The experiment was modified to include a while loop that looped over the execution trace of each query and perform the LSI+Dyn approach. The component to compute metrics for this approach was inserted after the goldset importer. The results of LSI and LSI+Dyn are then shown side by side in the results visualization GUI.

Table 7.6 shows the effectiveness measures of LSI and LSI in combination with execution traces (referred to as "LSI+Dyn" from here on). As the table shows, LSI+Dyn greatly improves the effectiveness of feature location.

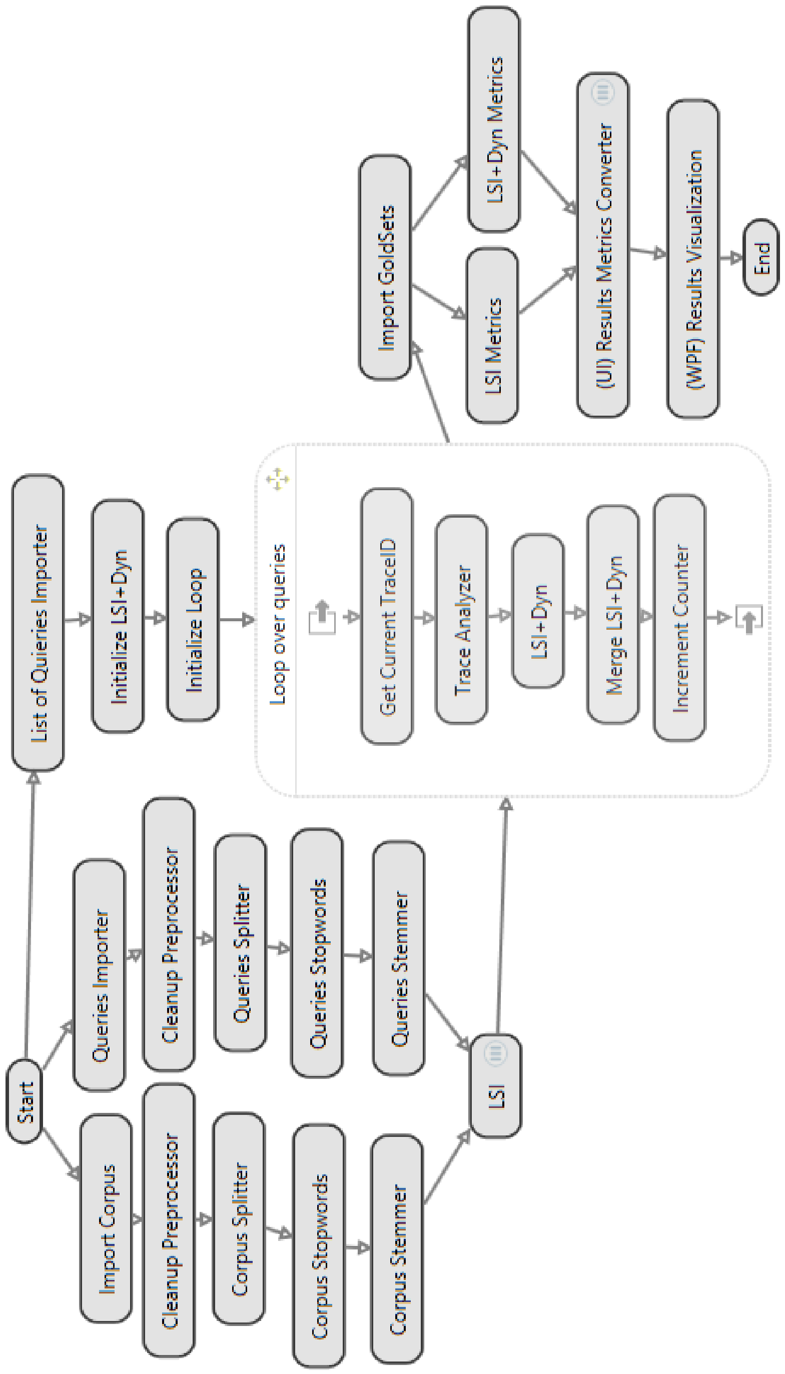


Figure 7.9: Liu et al. TraceLab experiment

Percentile	**Best**		**All**	
	LSI	LSI+Dyn	LSI	LSI+Dyn
Max	4140	2438	5728	3384
Q3	165.25	116	508.5	320.25
Median	39	23	136	94
Q1	4.75	3	37.25	22
Min	0	0	0	0

Table 7.6: Liu et al. effectiveness measures of TraceLab evaluation

7.2.4 Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software

Dit et al [58] took the idea of incorporating dynamic information one step further. They realized that by analyzing the execution traces, a program dependency graph (PDG) could be created by extracting method call information. Once a PDG was created, they could use link analysis algorithms borrowed from web mining to determine which methods in the trace were most important. The two algorithms they used were PageRank [70] and Hyperlinked-Induced Topic Search (HITS) [71]. PageRank imitates user behavior of navigating links on web pages and calculates a page's relative importance from the probability that a user on another page will follow a link to that page. HITS treats pages as hubs and authorities, where hubs are pages that have many links to other information and authorities are pages with information that are pointed to by many other pages. In both cases, methods are modeled as "pages" and method calls are "links" between pages.

From the results of the link analysis, they filtered the ranklist produced by LSI by including only a percentage of the most important methods. For example, if PageRank determined that methods A, B, and C were the most important in the PDG, then the LSI ranklist would be filtered of all other methods, leaving only A, B, and C. This further removes irrelevant methods and moves relevant ones closer to the top of the list. The results of the standalone technique (*ie.* the methods are ranked by their link analysis scores) are also computed.

The authors compared their approach to standalone LSI and LSI+Dyn by performing an evaluation on jEdit, Eclipse, and Rhino [72], a JavaScript engine written in Java. They reported the effectiveness measures of the standalone techniques as well as filtering the LSI ranklist with the top and bottom methods of PageRank and HITS[2], varying the amount in 10% steps. They also reported the effect of using *binary* weights (*ie.* assigning a 0 or 1 to a link, depending on if it is called) and *frequency* weights (*ie.* the actual number of times the method is called). They concluded that using link analysis algorithms to complement the dynamic approach significantly improves the effectiveness of feature location.

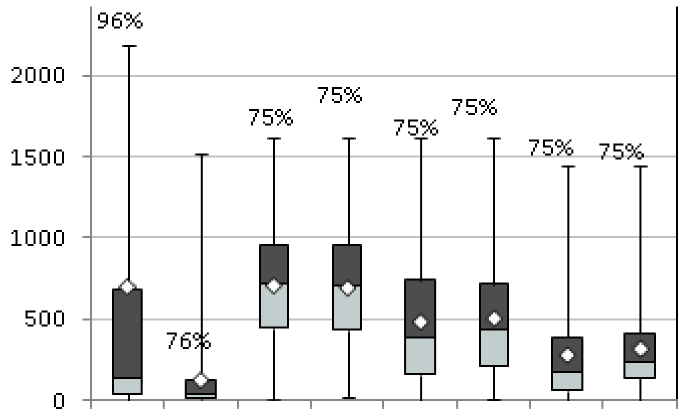
I was able to obtain from the authors the exact same tools and jEdit data used in the paper. Figure 7.11 shows the experiment in TraceLab. This was built upon the experiment in the previous section by adding PageRank and HITS components to the while loop, then creating composite nodes to loop over the top and bottom filtering techniques. Finally, the metrics calculations were merged into a single composite component due to the large number of calculations needed (to preserve space).

Figure 7.10 shows a comparison of some of the original results in the paper and the results computed in TraceLab. From left to right, the boxplots in Figures 7.10a and Figure 7.10b represent the effectiveness all measures for the standalone methods used in the experiment: LSI, LSI+Dyn, Pagerank (frequency), PageRank (binary), HITS (authorities, frequency), HITS (authorities, binary), HITS (hubs, frequency), and HITS (hubs, binary). Small variations in the percentiles are due to differences

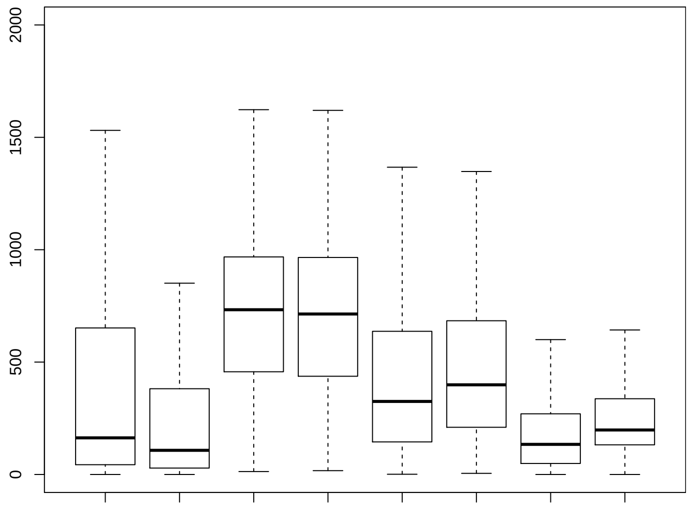
[2]The motivation for filtering the top methods returned by link analysis comes from the intuition that in HITS, methods with high hub scores will in very general classes that perform a variety of tasks and not relevant to the feature. They apply this to every technique for the sake of comparison.

in the way outliers were calculated for the graphs, but the results are the same.

Due to the large amount of results, the rest of the results graphs are not shown here. However, because I had access to the same data and tools from the paper, the TraceLab evaluation produces the same exact results. The evaluation shows that by combining web mining algorithms with execution traces and LSI, the effectiveness of feature location is greatly improved.



(a) Figure 8(c) in Dit et al.



(b) TraceLab results

Figure 7.10: Dit et al. Comparison of results from paper and TraceLab

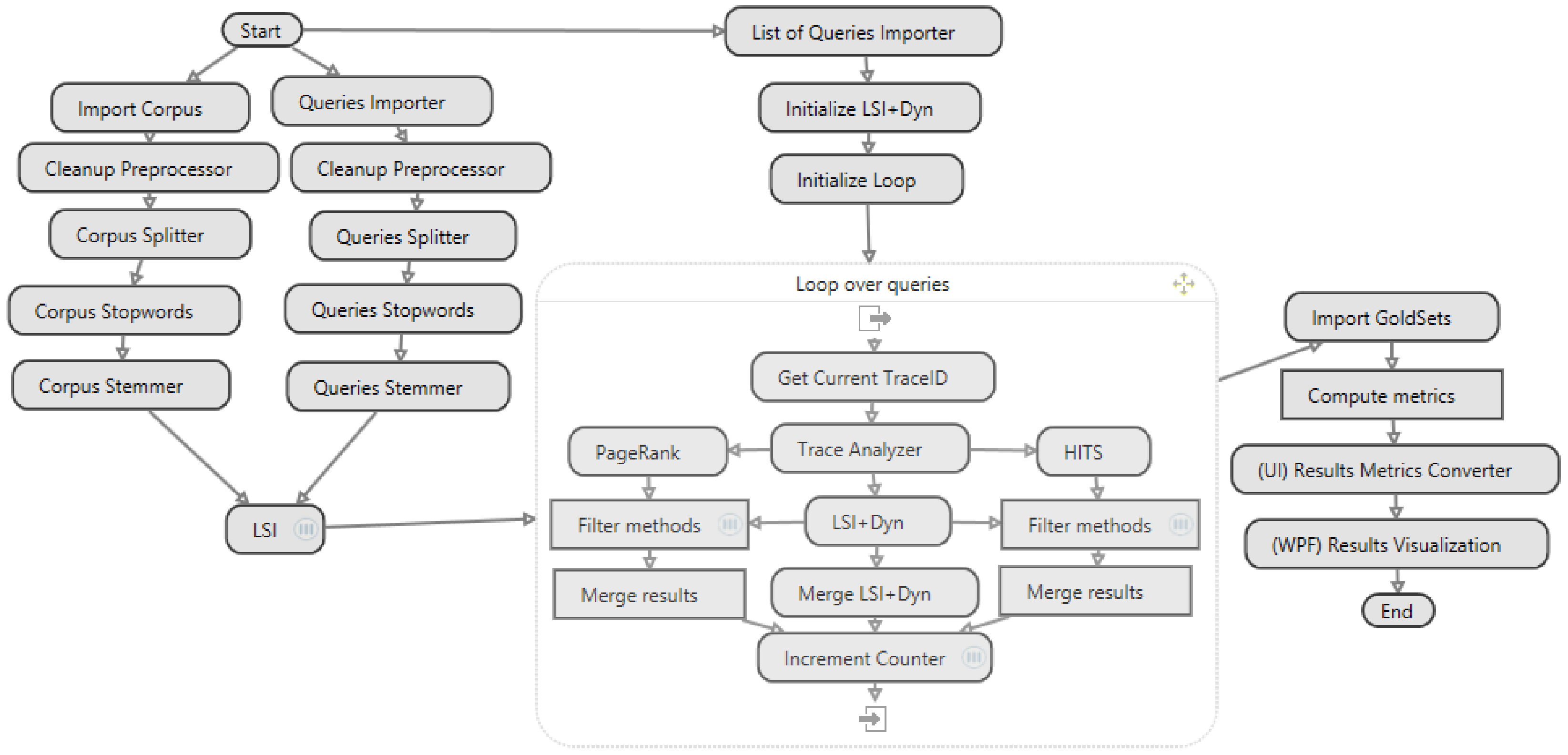


Figure 7.11: Dit et al. TraceLab experiment (link analysis)

7.3 Additional Examples

This section provides additional examples of reproducing research in software engineering. Section 7.3.1 shows how smoothing filters in the term-by-document matrix can improve the results of traceability link recovery. Section 7.3.2 shows the effects of combining structural information about source code with information retrieval to improve traceability link recovery. Finally, Section 7.3.3 shows how genetic algorithms can be applied to configure topic models to improve traceability link recovery.

7.3.1 Improving IR-based Traceability Recovery Using Smoothing Filters

De Lucia et al [45] proposed a new method of altering the term-by-document matrix in order to promote and minimize certain terms in the matrix. Their approach consisted of constructing a smoothing filter to reduce the "noise" within documents. To do this, they calculated a vector consisting of the average weight of each term in the matrix, then subtracted that vector from each document in the matrix. In essence, terms that appear frequently in a corpus are removed, while the important terms remain. This is done on each set of artifacts independently. They then computed the cosine similarities between documents.

The authors performed an evaluation of their approach on two datasets, EasyClinic and Pine. They compared their approach to basic VSM and LSI and reported the precision and recall of the results. They concluded that their approach significantly improved the results of traceability link recovery.

This paper contains one of the approaches that can be completely implemented in TraceLab. Furthermore, I had access to one of the datasets used in the evaluation, EasyClinic. Figure 7.12 shows the experiment as it appears in TraceLab. After importing each of the artifacts types (use cases, interaction diagrams, test cases, and code classes) and performing various preprocessing techniques, basic VSM and VSM with the smoothing filter is computed between the source artifacts and code classes. Finally, the precision-recall curves are computed and displayed in a GUI.

The precision-recall curves of the evaluation are shown in Figure 7.13. Figure 7.13a shows the effects of the smoothing filter in tracing use cases to code classes. Figure 7.13b shows the effects of the smoothing filter in tracing interaction diagrams to code classes. Figure 7.13c shows the effects of the smoothing filter in tracing test cases to code classes. In the case of tracing use cases and interaction diagrams to code classes, the smoothing filter significantly improves the results over basic VSM. The smoothing filter provides no significant improvement over basic VSM in tracing test cases to code classes. These results are consistent with the results in the paper.

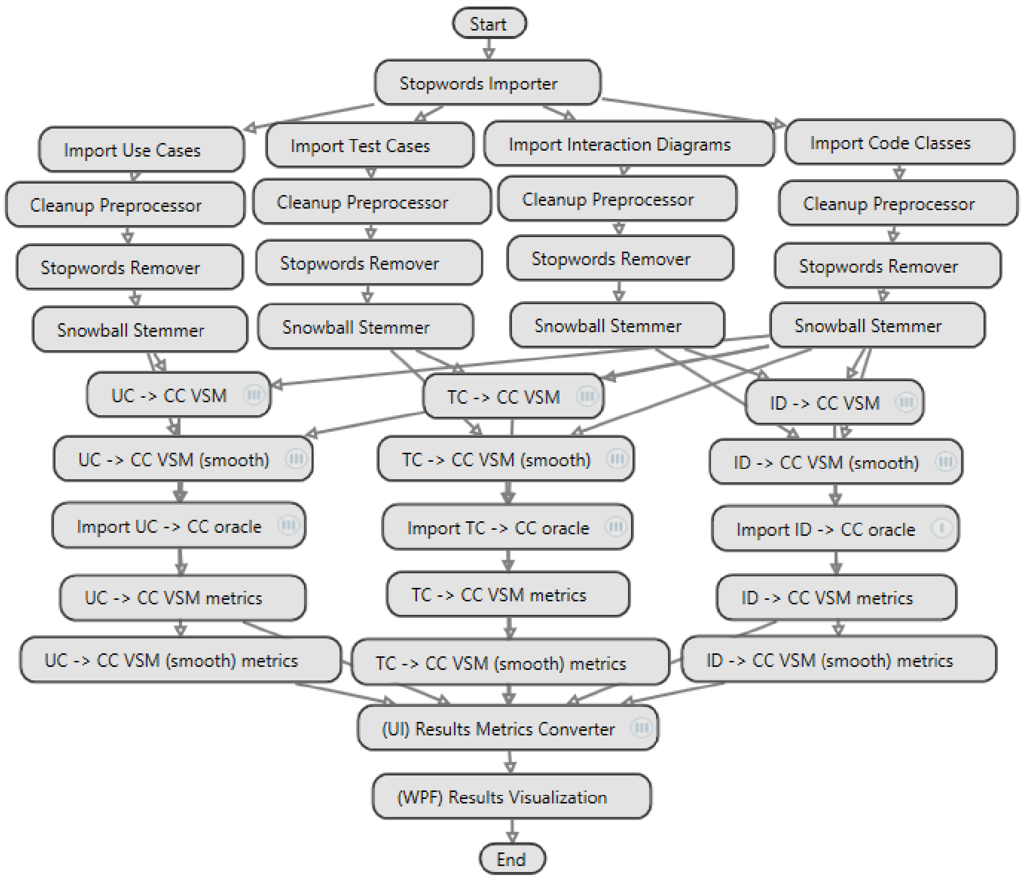
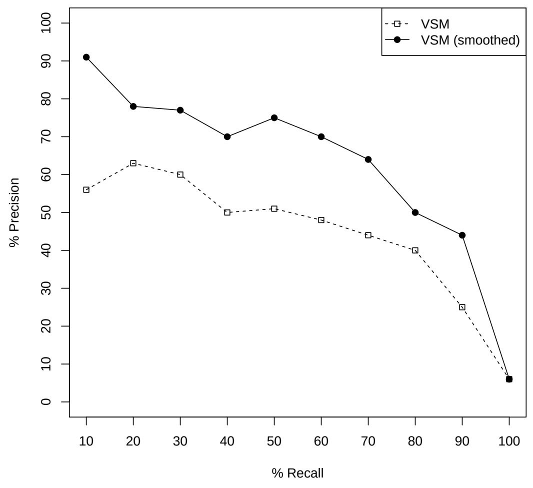
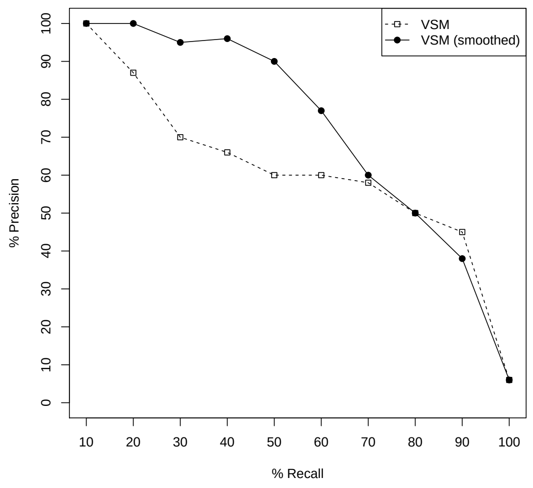


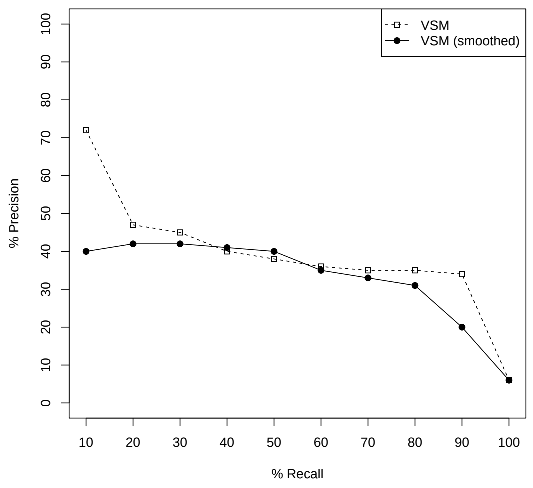
Figure 7.12: De Lucia et al. TraceLab experiment



(a) Use cases to code classes



(b) Interaction diagrams to code classes



(c) Test cases to code classes

Figure 7.13: De Lucia et al. Precision-recall curves of smoothing filter

7.3.2 Using Structural Information and User Feedback to Improve IR-based Traceability Recovery

Panichella et al [33] investigated improving the results of traceability link recovery by modifying the resultant ranklist with structural information. They proposed two methods to do this, Optimistic Combination of Structural and Textual Information (O-CSTI) and User-Driven Combination of Structural and Textual Information (UD-CSTI).

In O-CSTI, structural relationships between artifacts (such as method call dependencies or inheritance relationships) are used to increase the score (and thus the position) of related links in the ranklist. For example, if use case A is related to code class B, C, and D, and the link from A to C is at the top of the ranklist, then the related code classes B and D are given a bonus to their similarity score, moving them higher in the list. This bonus is computed automatically – for each source artifact s_i in the ranklist, δ_i is computed from the maximum and minimum similarity scores for links involving s_i (Equation 7.1). Then the overall bonus used is the median value of these deltas. The bonus is applied to related links via Equation 7.2.

$$\delta_i = \tfrac{s_{i,max} - s_{i,min}}{2} \tag{7.1}$$

$$bonus = score + score * \delta \tag{7.2}$$

In UD-CSTI, this approach is complemented by user feedback. After computing the initial ranklist, the user is presented with a link and classifies it as true or false. The classified link is removed from the ranklist (and added to a new one if it was true), and the remaining links are updated with the relationship bonus and reordered. The process continues until all correct links have been retrieved.

The authors performed an evaluation of their approach on EasyClinic, eTour, and SMOS. They compared their technique using basic VSM and JS and report precision-recall curves of their results. They concluded that their approach improves the results of traceability link recovery.

This paper was published with an implementation in TraceLab. I had access to the original experiment and data. I perform an evaluation using basic VSM and EasyClinic. Figure 7.15 shows the experiment in TraceLab. After importing the data and performing various preprocessing techniques, the initial IR technique is run. Then the results of the IR technique are sent to the O-CSTI and UD-CSTI components. The user feedback aspect of UD-CSTI is simulated using the oracle. Finally, the precision and recall of each technique is computed and displayed in a GUI.

Figure 7.14 shows the precision-recall curves of tracing use cases to code classes for basic VSM, O-CSTI, and UD-CSTI. The graph shows that both O-CSTI and OD-CSTI outperform the standalone IR technique. These results are consistent with the paper.

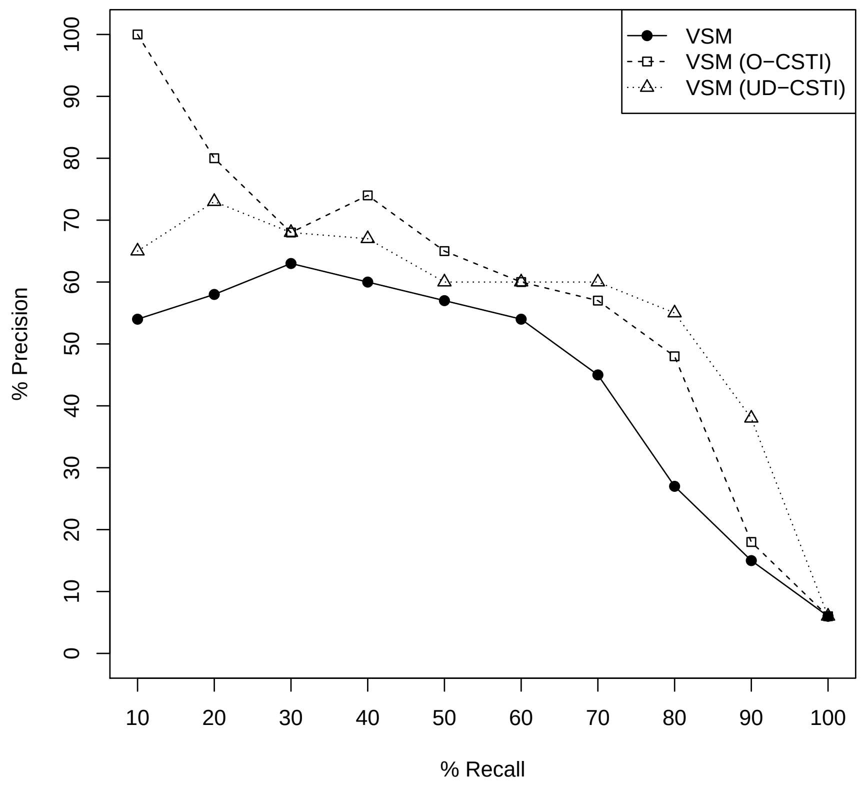


Figure 7.14: Panichella et al. Precision-recall curve of evaluation in TraceLab

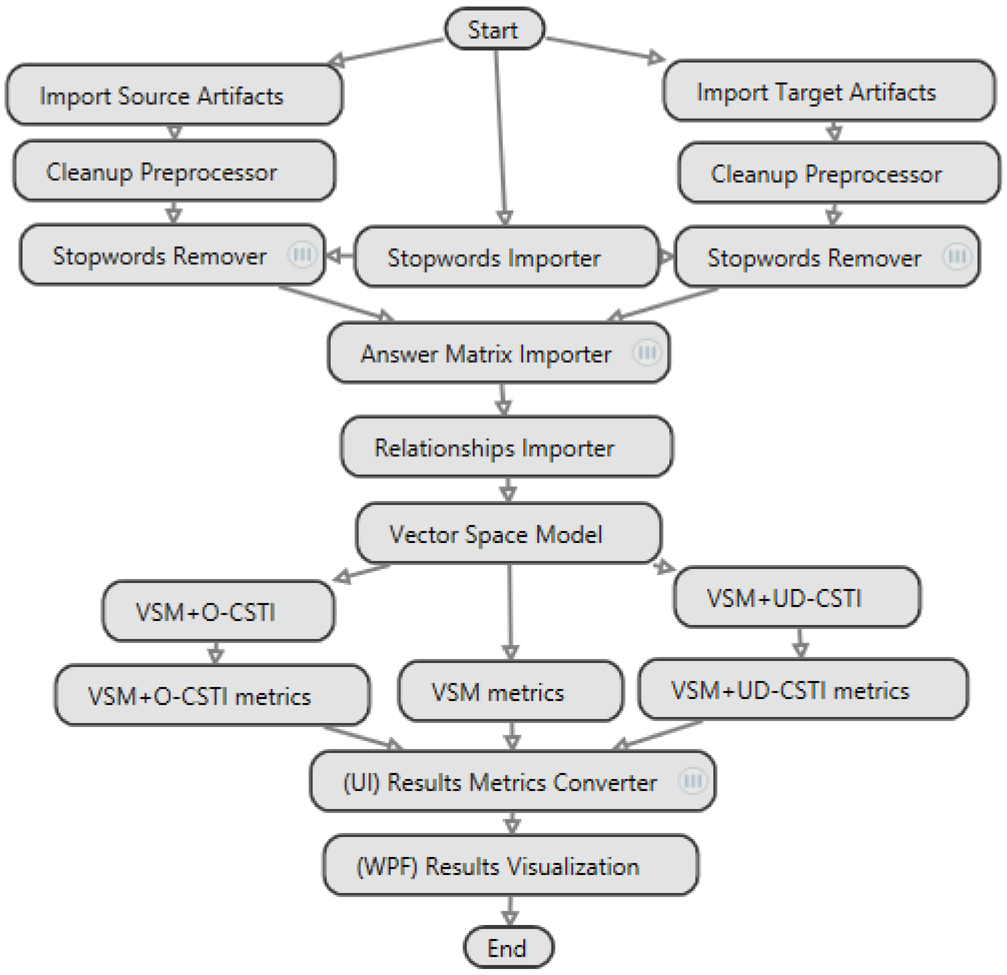


Figure 7.15: Panichella et al. TraceLab experiment

7.3.3 Configuring Topic Models for Software Engineering Tasks in TraceLab

Dit et al. [34] presented a method for configuring topic models to improve traceability link recovery by using genetic algorithms. Such an approach allows topic models to be configured without know the oracle or information about the dataset *a prioiri*. Given a fitness function for the topic model, the genetic algorithm randomly chooses configuration parameters and runs the model for a set population. The configurations with the highest fitness score are carried over to the next iteration, with a chance of mutation to discover new configurations. After a certain number of iterations, the configuration with the highest fitness funtion is returned.

The authors applied the genetic algorithm to Latent Dirichlet Allocation (LDA), using a measure from the model's internals based on the Silhouette coefficient [73] as the fitness function. They performed an evaluation on EasyClinic, comparing their approach to a configuration of LDA used from previous work [43]. They concluded that their approach greatly improves the results of traceability link recovery when using LDA.

This paper was published with an implementation in TraceLab. I had access to the original experiment and data. I perform the evaluation on both EasyClinic and eTour to demonstrate its effectiveness. Figure 7.16 shows the experiment in TraceLab. After importing the data and performing only basic preprocessing, the genetic algorithm runs and feeds the resulting configuration parameters to LDA. At the same time, the baseline LDA from the previous work is computed. Finally, the precision-recall curves for each technique are calculated and displayed in a GUI.

Figures 7.17a and 7.17b show the precision-recall curves of tracing use cases to code classes for EasyClinic and eTour using baseline LDA and configured LDA. The results show that using the genetic algorithm to compute an ideal set of configuration parameters greatly improves the results of traceability link recovery when using LDA.

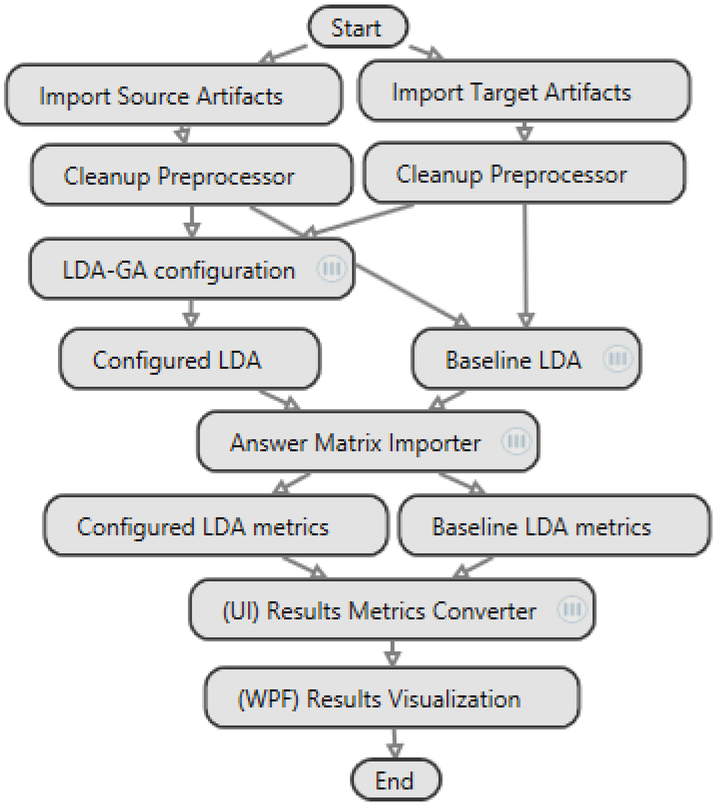
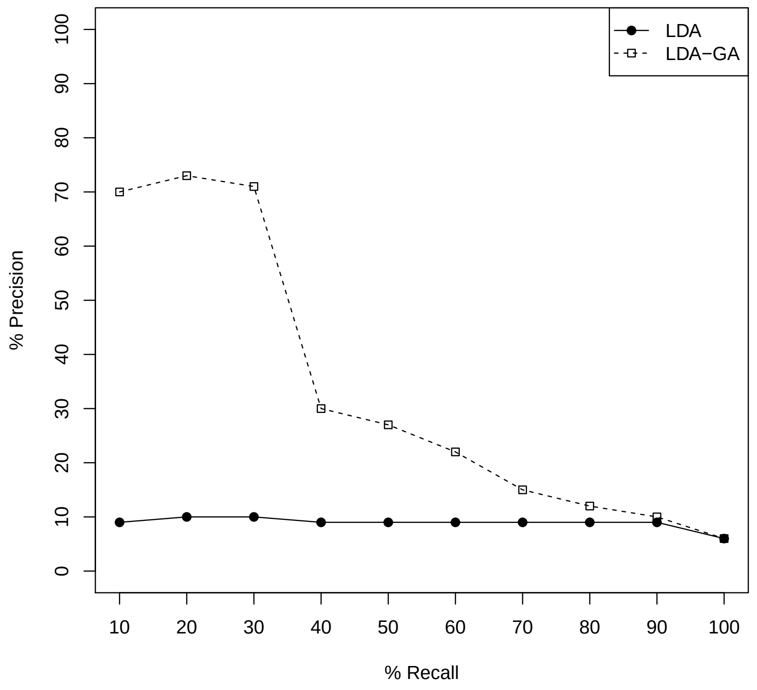
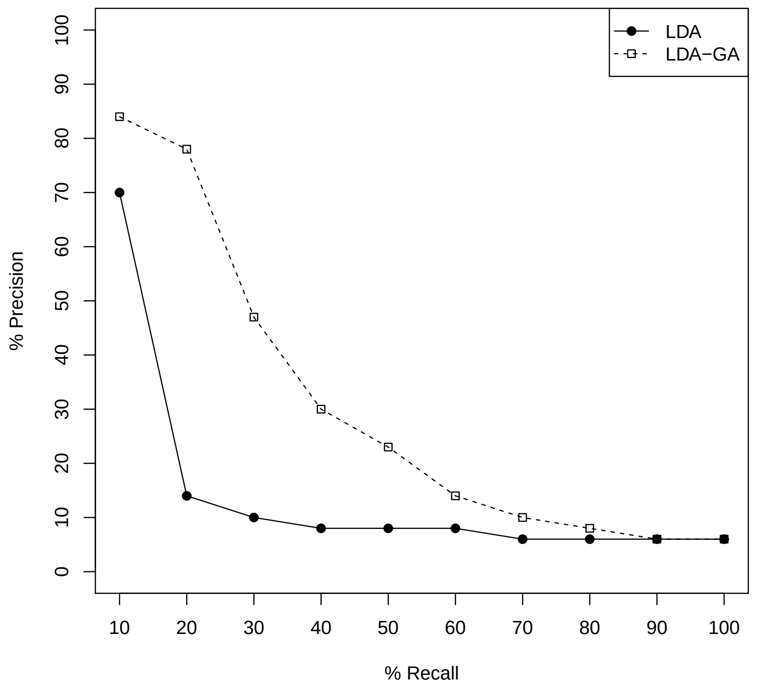


Figure 7.16: Dit et al. TraceLab experiment (LDA genetic algorithm)



(a) EasyClinic



(b) eTour

Figure 7.17: Dit et al. Precision recall curves of using genetic algorithm for LDA configuration

Chapter 8

Conclusions

In this thesis, I addressed one of the hidden problems in software engineering research. The inability to reproduce published research – due to lack of detail, tool or data availability, or even minute settings within an environment – creates a serious roadblock to validating approaches and driving new ideas. Using the TraceLab experimental workbench and the software development tools it provides, I realized the overarching goal of providing a set of clear, precise, and available tools for performing software engineering research. The advantages such a collection provides, combined with the usability of TraceLab, allows researchers to execute and publish experiments that can be validated and built upon by anyone using the TraceLab framework.

In Chapter 2, I described previous meta-studies investigating issues with reproducibility and validity in software engineering research. Furthermore, I compared TraceLab with commonly-used research tools and enumerated why they did not solve these issues. Chapter 3 reiterated the necessity of approaching this problem, giving concrete examples of why even the smallest details can vastly change the results of an approach and omitting them creates unknowns.

In Chapter 4, I detail the workings of the TraceLab project. TraceLab is funded by the NSF and developed by DePaul University in collaboration with many other universities around the world, including the College of William & Mary. TraceLab is a workbench designed to create, execute, and share research within the software engineering community. I take advantage of the extensibility of this framework to design and implement a new component library in order to provide the necessary tools used in software engineering research.

In Chapter 5, I performed a systematic mapping study of software engineering papers published in top international software engineering conferences in the past 10 years, focusing on the areas of traceability link recovery. I identified the most common techniques used and analyzed patterns within each area. This information was used to drive the creation of the Component Library.

Chapter 6 details the Component Library and the underlying Component Development Kit, which contains many of the tools and techniques identified in the mapping study as well as other useful tools. Using the CL and CDK, it is possibly to completely implement 37% of the approaches identified in the mapping study, with an additional 37% missing only 1 technique. I provide the implementation, source code, and documentation of the CL and CDK for public download for researchers to use.

In Chapter 7, I use the CL and CDK in TraceLab to provide concrete examples of reproducing previous research. I show the use of information retrieval techniques for traceability link recovery and how they came to be used in combination, showing the ease of which a TraceLab experiment can be modified to try new ideas. I show the application of LSI for feature location and how it has evolved into the basis for extensive and effective data fusion techniques. Finally, I provide additional examples of previous published research that can be reproduced in TraceLab.

Reproducibility is a major tenet of scientific research. Without it, new ideas cannot be validated and built upon. My work attempts to address this issue within the software engineering community by providing a set of tools in a framework that contains every detail of an approach, which can be shared and built upon by others. While the Component Library and CDK do not provide every tool necessary to a researcher, I believe it provides a strong body of existing knowledge and lays the foundation for future software engineering research – a future of transparency and accelerated learning.

BIBLIOGRAPHY

[1] T. Menzies and M. Shepperd, "Special issue on repeatable results in software engineering prediction," Empirical Software Engineering, vol. 17, pp. 1–17.

[2] National Science Foundation, "MRI-R2: Development of a software traceability instrument to facilitate and empower traceability research and technology transfer," http://nsf.gov/awardsearch/showAward?AWD_ID=0959924.

[3] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, O. Gotel, J. H. 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 Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, ser. TEFSE '11, 2011, pp. 17–23.

[4] 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 Proceedings of 34th IEEE/ACM International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, June 2012, pp. 1329–1332.

[5] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. 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 Proceedings of 34th IEEE/ACM International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, June 2012, pp. 1375–1378.

[6] 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, 2013.

[7] G. Robles, "Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings," 2010, pp. 171–180.

[8] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," Empirical Software Engineering, vol. 17, no. 4-5, pp. 531–577, 2012.

[9] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "The effect of omitted-variable bias on the evaluation of compiler optimizations," Computer, vol. 43, no. 9, pp. 62–67, 2010.

[10] E. Barr, C. Bird, E. Hyatt, T. Menzies, and G. Robles, "On the shoulders of giants," in Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, 2010, pp. 23–28.

[11] 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, vol. 17, no. 1-2, pp. 75–89, 2012.

[12] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability," Empirical Software Engineering, pp. 1–52, 2013.

[13] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The PROMISE repository of empirical software engineering data," http://promisedata.googlecode.com/.

[14] S. J. Sayyad and T. J. Menzies, "The PROMISE repository of software engineering databases," http://promise.site.uottawa.ca/SERepository.

[15] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on. IEEE, 2007, pp. 9–9.

[16] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empirical Software Engineering, vol. 10, no. 4, pp. 405–435, 2005.

[17] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, "A dataset from change history to support evaluation of software maintenance tasks," in Proceedings of the Tenth International Workshop on Mining Software Repositories. IEEE Press, 2013, pp. 131–134.

[18] R Project, "The R Project for statistical computing," http://www.r-project.org/.

[19] Mathworks, "Matlab," http://www.mathworks.com/products/matlab/.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," SIGKDD Explorations, vol. 11, no. 1, 2009.

[21] Rapid-I, "RapidMiner," http://rapid-i.com/content/view/181/190/.

[22] Mathworks, "Simulink," http://www.mathworks.com/products/simulink/.

[23] The University of Sheffield, "GATE: General Architecture for Text Engineering," http://gate.ac.uk/.

[24] Yahoo!, "Yahoo! pipes," http://pipes.yahoo.com/pipes/.

[25] University of California, "The Kepler Project," https://kepler-project.org/.

[26] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval, 1999.

[27] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," Information Processing & Management, vol. 24, no. 5, pp. 513–523, 1988.

[28] 5th International Workshop on Traceability in Emerging Forms of Software Engineering, http://web.soccerlab.polymtl.ca/tefse09/Challenge.htm, 2009.

[29] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on. IEEE, 2009, pp. 71–80.

[30] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. IEEE, 2011, pp. 11–20.

[31] M. F. Porter, "An algorithm for suffix stripping," Program: electronic library and information systems, vol. 14, no. 3, pp. 130–137, 1980.

[32] B. Dit, E. Moritz, and D. Poshyvanyk, "A Tracelab-based solution for creating, conducting, and sharing feature location experiments," in Proceedings of 20th IEEE International Conference on Program Comprehension (ICPC'12), 2012, pp. 203–208.

[33] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Using structural information and user feedback to

improve IR-based traceability recovery," in Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13), 2013.

[34] 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 Proceedings of 7th ICSE'13 International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13), San Francisco, California, 2013, pp. 105–109.

[35] 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.

[36] M. Jørgensen, "A review of studies on expert estimation of software development effort," Journal of Systems and Software, vol. 70, no. 1, pp. 37–60, 2004.

[37] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in Proceeding of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), vol. 17, 2008, p. 1.

[38] O. Gotel and A. Finklestein, "An analysis of the requirements traceability problem," in Proceedings of the 1st IEEE International Conference on Requirements Engineering (ICRE'94). Colorado Springs, CO: IEEE CS Press, 1994.

[39] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in Reverse Engineering, 2004. Proceedings. 11th Working Conference on. IEEE, 2004, pp. 214–223.

[40] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007, pp. 499–510.

[41] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in Proceedings of the 16th IEEE Conference on Program Comprehension (ICPC'08), pp. 103–112.

[42] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in IR-based traceability link recovery," in Proceedings of the 17th International Conference on Program Comprehension (ICPC'09), 2009, pp. 148–157.

[43] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in

Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, June 30 - July 2 2010.

[44] H. Asuncion, A. Asuncion, , and R. Taylor, "Software traceability with topic modeling," in Proceedings of the 32nd IEEE International Conference on Software Engineering (ICSE10), 2010.

[45] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based traceability recovery using smoothing filters," in Proceedings of the 19th International Conference on Program Comprehension (ICPC'11), Kingston, Canada, 2011, pp. 21–30.

[46] X. Chen, J. Hosking, and J. Grundy, "A combination approach for enhancing automated traceability:(nier track)," in Software Engineering (ICSE), 2011 33rd International Conference on. IEEE, 2011, pp. 912–915.

[47] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On integrating orthogonal information retrieval methods to improve traceability link recovery," in Proceedings of 27th IEEE International Conference on Software Maintenance (ICSM'11), Williamsburg, VA, September 2011.

[48] 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 Proceedings of 35th IEEE/ACM International Conference on Software Engineering (ICSE'13), San Francisco, CA.

[49] K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent dirichlet allocation for automatic categorization of software," in Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on. IEEE, 2009, pp. 163–166.

[50] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol. 2. IEEE, 2010, pp. 223–226.

[51] 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 Program Comprehension (ICPC), 2012 IEEE 20th International Conference on. IEEE, 2012, pp. 193–202.

[52] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace,"

in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007, pp. 234–243.

[53] 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,” Software Engineering, IEEE Transactions on, vol. 33, no. 6, pp. 420–432, 2007.

[54] M. Revelle and D. Poshyvanyk, “An exploratory study on assessing feature location techniques,” in Program Comprehension, 2009. ICPC’09. IEEE 17th International Conference on. IEEE, 2009, pp. 218–222.

[55] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, “On the use of relevance feedback in ir-based concept location,” in Software Maintenance, 2009. ICSM 2009. IEEE International Conference on. IEEE, 2009, pp. 351–360.

[56] G. Scanniello and A. Marcus, “Clustering support for static concept location in source code,” in Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. IEEE, 2011, pp. 1–10.

[57] A. Wiese, V. Ho, and E. Hill, “A comparison of stemmers on source code identifiers for software search,” in Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, 2011, pp. 496–499.

[58] 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, no. 2, pp. 277–309, 2013.

[59] 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 Proceedings of the 30th international conference on Software engineering. ACM, 2008, pp. 461–470.

[60] N. Kaushik and L. Tahvildari, “A comparative study of the performance of ir models on duplicate bug detection,” in Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, 2012, pp. 159–168.

[61] http://www.crc.ca/en/html/crc/home/research/satcom/rars/sdr/products/scari_open/scari_open, 2008, website unavailable, 15 May 2013.

[62] http://cvs.savannah.gnu.org/viewvc/classpath/, 2008.

[63] 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, no. 6, pp. 391–407, 1990.

[64] 6th International Workshop on Traceability in Emerging Forms of Software Engineering, http://www.cs.wm.edu/semeru/tefse2011/Challenge.htm, 2011.

[65] J. Chang and D. M. Blei, "Hierarchical relational models for document networks," Annals of Applied Statistics, 2010.

[66] G. Salton and M. J. McGill, "Introduction to modern information retrieval," 1986.

[67] Mosaic, "Mosaic source code v2.7b5," ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/.

[68] "jEdit," http://www.jedit.com/.

[69] T. E. Foundation, "Eclipse," http://www.eclipse.org/.

[70] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," Computer networks and ISDN systems, vol. 30, no. 1, pp. 107–117, 1998.

[71] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," Journal of the ACM (JACM), vol. 46, no. 5, pp. 604–632, 1999.

[72] Mozilla, "Rhino," https://developer.mozilla.org/en-US/docs/Rhino.

[73] J. Kogan, Introduction to clustering large and high-dimensional data. Cambridge University Press, 2006.

[74] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), 2001, pp. 103–112.

[75] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining textual and structural analysis of software artifacts for traceability link recovery," in Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering, 2009, pp. 41–48.