CAN CLUSTERING IMPROVE REQUIREMENTS TRACEABILITY? A TRACELAB-ENABLED STUDY

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Brett Armstrong

December 2013

COMMITTEE MEMBERSHIP

TITLE:	Can Clustering Improve Requirements Traceability? A Tracelab-enabled Study
AUTHOR:	Brett Armstrong
DATE SUBMITTED:	December 2013
COMMITTEE CHAIR:	Professor Alexander Dekhtyar, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor David Janzen, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor Foaad Khosmood, Ph.D., Department of Computer Science

ABSTRACT

Can Clustering Improve Requirements Traceability? A Tracelab-enabled Study

Brett Armstrong

Software permeates every aspect of our modern lives. In many applications, such in the software for airplane flight controls, or nuclear power control systems software failures can have catastrophic consequences. As we place so much trust in software, how can we know if it is trustworthy? Through software assurance, we can attempt to quantify just that.

Building complex, high assurance software is no simple task. The difficult information landscape of a software engineering project can make verification and validation, the process by which the assurance of a software is assessed, very difficult. In order to manage the inevitable information overload of complex software projects, we need software traceability, "the ability to describe and follow the life of a requirement, in both forwards and backwards direction."

The Center of Excellence for Software Traceability (CoEST) has created a compelling research agenda with the goal of ubiquitous traceability by 2035. As part of this goal, they have developed TraceLab, a visual experimental workbench built to support design, implementation, and execution of traceability experiments. Through our collaboration with CoEST, we have made several contributions to TraceLab and its community.

This work contributes to the goals of the traceability research community. The three key contributions are (a) a machine learning component package for TraceLab featuring six (6) classifier algorithms, five (5) clustering algorithms, and a total of over 40 components for creating TraceLab experiments, built upon

the WEKA machine learning package, as well as implementing methods outside of WEKA; (b) the design for an automated tracing system that uses clustering to decompose the task of tracing into many smaller tracing subproblems; and (c) an implementation of several key components of this tracing system using TraceLab and its experimental evaluation.

ACKNOWLEDGMENTS

Thanks to:

- My advisor Alex Dekhtyar, who has been awesome in every regard.
- Jane Cleland-Huang, Piotr Pruski, and the TraceLab team for including and supporting me on their project.
- My girlfriend Giselle Griffin, who has encouraged me every step of the way.
- My parents and my family for supporting me always.

TABLE OF CONTENTS

List of Tables ix

List of Figures x

1 Introduction 1

2 Background and Related Work 6

2.1 Requirements Tracing 6

2.2 Center of Excellence for Software Traceability (CoEST) 9

2.2.1 Grand Challenges of Traceability 9

2.2.2 Tracy Project 10

2.3 TraceLab 12

2.3.1 Building TraceLab Experiments 14

2.3.2 TraceLab Components and Workspace Types 15

2.4 Information Retrieval 16

2.4.1 Modeling 17

2.4.2 Measures 19

2.4.3 Automated Traceability and Information Retrieval 23

2.5 Machine Learning 25

2.5.1 Classification 25

2.5.2 Clustering 27

2.5.3 Automated Traceability and Machine Learning 27

2.5.4 WEKA 28

3 TraceLab Machine Learning Component Package 30

3.1 Writing Code for TraceLab 30

3.2 Using WEKA 31

3.3 Data Types 33

3.3.1 TLarff 37

3.3.2 TLCluster and TLClustersCollection 38
3.3.3 TLDoubleTree 38
3.3.4 TLConfusionMatrix 39
3.4 Components 39
3.4.1 Importers 41
3.4.2 Exporters 42
3.4.3 Classifiers 43
3.4.4 Clusterers 46
3.4.5 Preprocessors 49
3.4.6 Postprocessors 54
3.4.7 Helper Components. 56
4 Clustering and Automated Tracing 59
4.1 Clustering 63
4.2 Cluster-To-Cluster Tracing 63
4.3 Intercluster Tracing 65
5 Experimental Design 67
5.1 Datasets 67
5.2 Setup 68
5.3 Implementation 69
5.4 Results 79
5.5 Analysis 81
6 Conclusion 85
Bibliography 88
Appendix A Results Graphs 93

LIST OF TABLES

2.1 Acceptable levels for recall precision and lag [21] 21

3.1 Machine learning component package data types. 34
3.2 Machine learning component package components. 35
3.3 Machine learning component package components (Continued). . . 36
3.4 Classifier Component Descriptions. 44
3.5 Clusterer Component Descriptions. 47
3.6 Preprocessor Component Descriptions. 50
3.7 Preprocessor Component Descriptions. 53
3.8 Postprocessor Component Descriptions. 55
3.9 Helper Component Descriptions. 57

5.1 Results Evaluation 76

LIST OF FIGURES

2.1 The Tracy Project [8]. 11
2.2 The CoEST.org community component directory [11]. 13
2.3 TraceLab architecture. 15
2.4 Sample traceLab experiment [11]. 17
2.5 Possible tracing outcomes. 20
2.6 Example ARFF file. 29

3.1 Sample C# code using Java objects with IKVM [24]. 32
3.2 Sample code for a workspace type. 33
3.3 Sample code for a component. 40
3.4 Sample code for a configuration class. 42
3.5 Sample 4 fold cross-validation process. 45
3.6 Hierarchy tree divided into clusters via fixed cutoff and fixed cluster count. 52

4.1 Tracing problem decomposition using clustering. 62

5.1 TraceLab experiment for creating a cluster tree from a set of artifacts. 70
5.2 Example Hierarchical Cluster Tree of WV_CCHIT Source Artifacts. 71
5.3 TraceLab experiment for evaluating clustered tracing. 73
5.4 TraceLab experiment for evaluating clustered tracing. 74
5.5 Clustered Tracing on CM1 dataset - Precision at 97% Cluster Recall and 80% Overall Recall. 77
5.6 Clustered Tracing on CCHIT dataset - Precision at 97% Cluster Recall and 80% Overall Recall. 78
5.7 Clustered tracing on WV_CCHIT dataset - Precision at 80% Recall with various cluster recalls. 80

5.8 Clustered tracing on CM1 dataset - Precision at 80% Recall with various cluster recalls. 80

5.9 Clustered Tracing on CM1 dataset - Precision Improvement at 97% Cluster Recall and 80% Overall Recall. 83

5.10 Clustered Tracing on CCHIT dataset - Precision Improvement at 97% Cluster Recall and 80% Overall Recall. 84

A.1 Clustered Tracing on CM1 dataset - Precision at 100% Cluster Recall and 80% Overall Recall. 94

A.2 Clustered Tracing on CM1 dataset - Precision at 97% Cluster Recall and 80% Overall Recall. 95

A.3 Clustered Tracing on CM1 dataset - Precision at 95% Cluster Recall and 80% Overall Recall. 96

A.4 Clustered Tracing on CM1 dataset - Precision at 90% Cluster Recall and 80% Overall Recall. 97

A.5 Clustered Tracing on CM1 dataset - Precision at 85% Cluster Recall and 80% Overall Recall. 98

A.6 Clustered Tracing on CM1 dataset - Precision at 80% Cluster Recall and 80% Overall Recall. 99

A.7 Clustered Tracing on CCHIT dataset - Precision at 100% Cluster Recall and 80% Overall Recall. 100

A.8 Clustered Tracing on CCHIT dataset - Precision at 97% Cluster Recall and 80% Overall Recall. 101

A.9 Clustered Tracing on CCHIT dataset - Precision at 95% Cluster Recall and 80% Overall Recall. 102

A.10 Clustered Tracing on CCHIT dataset - Precision at 90% Cluster Recall and 80% Overall Recall. 103

A.11 Clustered Tracing on CCHIT dataset - Precision at 85% Cluster Recall and 80% Overall Recall. 104

A.12 Clustered Tracing on CCHIT dataset - Precision at 80% Cluster Recall and 80% Overall Recall. 105

A.13 Clustered Tracing on CM1 dataset - Precision at 100% Cluster Recall and 80% Overall Recall. 106

A.14 Clustered Tracing on CM1 dataset - Precision at 97% Cluster Recall and 80% Overall Recall. 107

A.15 Clustered Tracing on CM1 dataset - Precision at 95% Cluster Recall and 80% Overall Recall. 108

A.16 Clustered Tracing on CM1 dataset - Precision at 90% Cluster Recall and 80% Overall Recall. 109

A.17 Clustered Tracing on CM1 dataset - Precision at 85% Cluster Recall and 80% Overall Recall. 110

A.18 Clustered Tracing on CM1 dataset - Precision at 80% Cluster Recall and 80% Overall Recall. 111

A.19 Clustered Tracing on CCHIT dataset - Precision at 100% Cluster Recall and 80% Overall Recall. 112

A.20 Clustered Tracing on CCHIT dataset - Precision at 97% Cluster Recall and 80% Overall Recall. 113

A.21 Clustered Tracing on CCHIT dataset - Precision at 95% Cluster Recall and 80% Overall Recall. 114

A.22 Clustered Tracing on CCHIT dataset - Precision at 90% Cluster Recall and 80% Overall Recall. 115

A.23 Clustered Tracing on CCHIT dataset - Precision at 85% Cluster Recall and 80% Overall Recall. 116

A.24 Clustered Tracing on CCHIT dataset - Precision at 80% Cluster Recall and 80% Overall Recall. 117

CHAPTER 1

Introduction

Software is ubiquitous. Since the term was coined in 1958 [1], software has worked its way into every aspect of our modern lives. It has become the link between our steering wheels and our axles, between our deposits and our withdrawals, between our takeoffs and our landings, even between our pacemakers and our hearts. We place so much trust in software, how can we know if it is trustworthy? Through software assurance, we can attempt to quantify just that.

Software assurance, defined in the National Information Assurance Glossary [34], is the

> Level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle and that the software functions in the intended manner.

Software assurance is assessed by a process known as verification and validation (V&V) which, as simplified by Boehm [5], answers the following two questions: "Was the software built correctly?" (Verification) and "Was the correct software built?" (Validation). If we can confidently answer "Yes" to both of these questions, then we can build high-assurance software.

Before answering either question, we must first understand the information landscape of a software project. As a project progresses through the software development lifecycle, many documents are produced to record how the project is

planned, designed, implemented, tested, and released. These documents include requirements specifications, design specifications, source code, test cases, bug reports, change reports, user manuals, and many more. Collectively, the elements in these documents are referred to as artifacts [6]. Artifacts are so important to V&V, and therefore software assurance, that many safety standards organizations, such as the Food and Drug Administration (FDA) and Federal Aviation Adminstration (FAA), mandate that software projects properly maintain these artifacts throughout their life cycle [6, 11].

Even a complete set of artifacts is not enough to properly verify and validate a software project. For example, how can we verify that a requirement is correct if we do not even know what design elements satisfy the requirement or where they are implemented in the source code? To solve this, we must trace requirements through each stage of the lifecycle and establish trace links between the related artifacts. This is known as requirements traceability, defined by Gotel and Finkenstein [16] as

> The ability to follow the life of a requirement, in both a 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).

Many safety standards organizations also require that traceability information be properly maintained along with the artifacts they link [6, 11].

Assuming the presence of properly maintained software artifacts and their links, it is now possible to verify and validate the software project to be confident that it is free of vulnerabilities and that it functions as intended. But where do trace links come from? Trace recovery tools generally fall into three categories: fully automated, semi-automated, and manual [4]. Automated methods often

employ information retrieval (IR) techniques to recover trace links [6]. In semi-automated methods, these links are then verified by a human. Manual methods require a human to establish each trace link; a process that is found to be tedious, error prone, and simply not scalable [6].

The traceability research community has been challenged to solve these problems. The grand challenge of traceability [15] demands ubiquitous traceability, defined as

> Traceability which is always there, without having to think about getting it there. It is neither consciously established nor sought; it is built in and effortless.

Manual methods do not fulfill this challenge, so the community must look to fully automated techniques. A systematic mapping (SM) of fully automated IR methods for trace recovery [6] examines how far we have to go to reach this lofty goal. The SM compares 132 empirical studies in 79 publications and identifies several pitfalls in the methodologies of these publications.

One major problem encountered by the traceability community is the lack of "large industrial datasets from various domains" [6]. Most publications used bipartite datasets with fewer than 500 datasets. The few publications that did use large scale industrial datasets used proprietary datasets that were not released with their results. Another issue found was the lack of reproducibility of the experiments. Either due to lack of time, lack of space, or simple omission by the authors, many reports failed to specify their methods with enough detail to reproduce the experiments [6].

In response to these shortcomings, the Center of Excellence for Software Traceability [11] has developed TraceLab, a visual experimental workbench for designing and executing traceability experiments [8]. TraceLab supports the

traceability research community by providing an easy way to share and reuse methodologies, datasets, and experiments.

Components are the building blocks of a TraceLab experiment. As such, the long term success of TraceLab is largely dependent on the quality and breadth of its component library. For this reason, CoEST has commissioned the authors to build a component package of machine learning algorithms. This package will be released through the CoEST website for anyone interested in using machine learning as part of their TraceLab experiment.

The TraceLab machine learning component package features six (6) classifier algorithms and five (5) clustering algorithms. Most of these algorithms use implementations in the Waikato Environment for Knowledge Analysis (WEKA). One of the classifier algorithms used, BPNB (see Section 2.5), has been implemented by the authors. The machine learning package includes a total of over 40 components to assist with the creation of TraceLab experiments. The component package also introduces four (4) new TraceLab data types. For more details about the machine learning component package, see Chapter 3.

We also contribute the design of an automated tracing system that applies machine learning to requirements tracing. This system uses clustering to decompose the task of tracing into many smaller tracing subproblems. This system is then implemented and evaluated using TraceLab experiments and the machine learning component package.

The rest of this work is organized as follows. Chapter 2 covers background and related work. Next, Chapter 3 gives a detailed overview of the machine learning component package. Then, Chapter 4 describes the design of the cluster based automated tracing system. After that, Chapter 5 discusses our TraceLab

implementation of the automated tracing system. Finally, Chapter 6 finishes with the conclusion and future work.

CHAPTER 2

Background and Related Work

2.1 Requirements Tracing

Requirements tracing is an important task in the software development life cycle. It supports many key functions such as change impact analysis, bug fixes, and verification and validation (V&V). Adequate tracing data is needed in order to answer common software engineering questions like "Where is this requirement implemented in source code?" or "Does this test case match its specification?"

The process of requirements tracing consists of establishing traceability relationships between a set of source artifacts and a set of target artifacts, such as tracing high-level requirements to low-level requirements, tracing high-level requirements to design specifications, or tracing design specifications to test cases. Traceability data is represented a requirements traceability matrix (RTM) or a list of trace links. It is a mapping between items in the two sets of artifacts.

During the tracing process, any links that have not been validated are candidate trace links. Candidate trace links can be generated by automated methods or by an intermediate step of a manual tracing process. Once the candidate RTM has been approved by a human analyst, it becomes a final RTM. A trace link that is included in the final RTM is a true link. A possible trace link that is not included in the final RTM is a false link.

Without tracing data, the vast number of requirements, specifications, code classes, and other software artifacts can cause information overload, especially on large software projects [25]. The difficult information landscape is tough for software engineers to navigate and can lead to project overruns and failures [6]. For these reasons, traceability is a major component of software development standards, such as CMMI and ISO 9001:2000. It is also mandated by many safety standards organizations, such as the FDA for software in medical devices, and the FAA for software onboard airplanes [6, 25].

With so many benefits to be gained from requirements tracing, why aren't more projects taking advantage of traceability data? Kannenberg and Saiedian [25] identify cost, change management, organizational problems, and poor tool support as the main obstacles impeding industry support of traceability practices. As software projects increase in size and complexity, the number of trace links significantly increases; as does the cost of manually finding each of those trace links. In addition, tracing data must be continually maintained throughout the life of a project, incurring even more cost to the project after every changed requirement or added feature.

Although the arguably high cost of traceability can save projects from even higher costs, budget overruns and failures by catching potential problems early, often project sponsors or upper management fail to understand the benefits of traceability. This results in a culture where traceability is not valued, except for standards compliance, leading to ad-hoc or haphazard trace creation and maintenance processes that incur much of the same costs, with little benefit [25].

Among all the challenges facing requirements traceability, many cite poor tool support as the most difficult to overcome. Although many tools claim to have full traceability support, several studies have found that adoption of these

tools in industry is only at about 50% [25, 26]. Possible reasons for the limited adoption of traceability tools are that these tools are usually not standalone, but part of a larger requirements management system that may not integrate with how existing projects already store requirements and specifications. Licenses for requirements management systems are often very expensive, especially if they are only going to be used for traceability. Also, full traceability support does not mean the tool has fully automated tracing. Most tools, though better than ad-hoc manual tracing efforts, still require a lot of work to establish traces and keep them up to date [35].

Big firms that value traceability often are left implementing in-house solutions that provide traceability custom tailored to their specific project workflows [33, 19]. Companies that do not have the budget for such solutions are either left with simplistic manual tracing solutions such as spreadsheets, or have no traceability at all.

If a cost-effective, fully automated traceability tool were to become available, the landscape of traceability would be drastically improved. First, the cost of achieving traceability for projects would drop significantly. This would likely change the opinions of project sponsors and upper management who, no longer blinded by high costs, would begin to see how beneficial tracing data really is. Eventually, this would improve tracing practices across the board resulting in higher quality, high assurance software systems with fewer budget overruns and failures.

2.2 Center of Excellence for Software Traceability (CoEST)

In response to the many shortfalls of traceability in industry, a group of academic, government, and industry researchers established the Center of Excellence for Software Traceability (CoEST) in 2005. The goal of CoEST is to bring together traceability researchers and experts in the field, encourage research collaborations, assemble a body of knowledge for traceability, and develop new technology to meet tracing needs [11]. The two flagship projects of CoEST thus far are the Grand Challenges of Traceability and the Tracy Project.

2.2.1 Grand Challenges of Traceability

The Grand Challenges of Traceability [15, 8] is a road map of critical research and practice goals. Grand challenges are meant to inspire researchers to achieve a difficult but significant goal. Originally, CoEST researchers defined eight grand challenges. This was later revised to be a single overarching grand challenge with seven sub-challenges. The grand challenge of traceability is ubiquitous traceability, previously defined as

> Traceability which is always there, without having to think about getting it there. It is neither consciously established nor sought; it is built in and effortless.

The seven sub-challenges that reinforce ubiquitous traceability are:

1. Purposed
2. Cost-effective
3. Configurable
4. Trusted
5. Scalable
6. Portable
7. Valued

Each of these challenges is rigorously defined at a very high level [15, 8]. For example, portable traceability is defined as follows:

> Traceability information is exchanged, merged, and reused across projects, organizations, domains, product lines, and supporting tools.

In addition, each challenge consists of concrete goals that outline research tasks. One example goal for ubiquitous traceability is

> Total automation of trace creation and trace maintenance, with quality and performance levels superior to manual efforts.

CoEST has set the goal of fulfilling the Grand Challenges of Traceability by the year 2035.

2.2.2 Tracy Project

The Tracy Project [8], driven by the grand challenges, focuses on creating research infrastructure for the traceability community. This includes collecting datasets, creating benchmarks, and building TraceLab, a tool for designing and conducting a range of traceability experiments. The major components of the Tracy Project are depicted in Figure 2.1

A systematic mapping (SM) of tracing approaches from May of 2013 [6] identified dataset quality as a major area of improvement for the traceability research community. The study classified 79 publications and only found 2 that used more than 10,000 artifacts[1] in their evaluations. The majority of evaluations only traced bipartite datasets, meaning they only contained a set of source and target artifacts to generate a single set of trace links, such as tracing high level

[1] An artifact refers to a single element in a requirements or design specification.

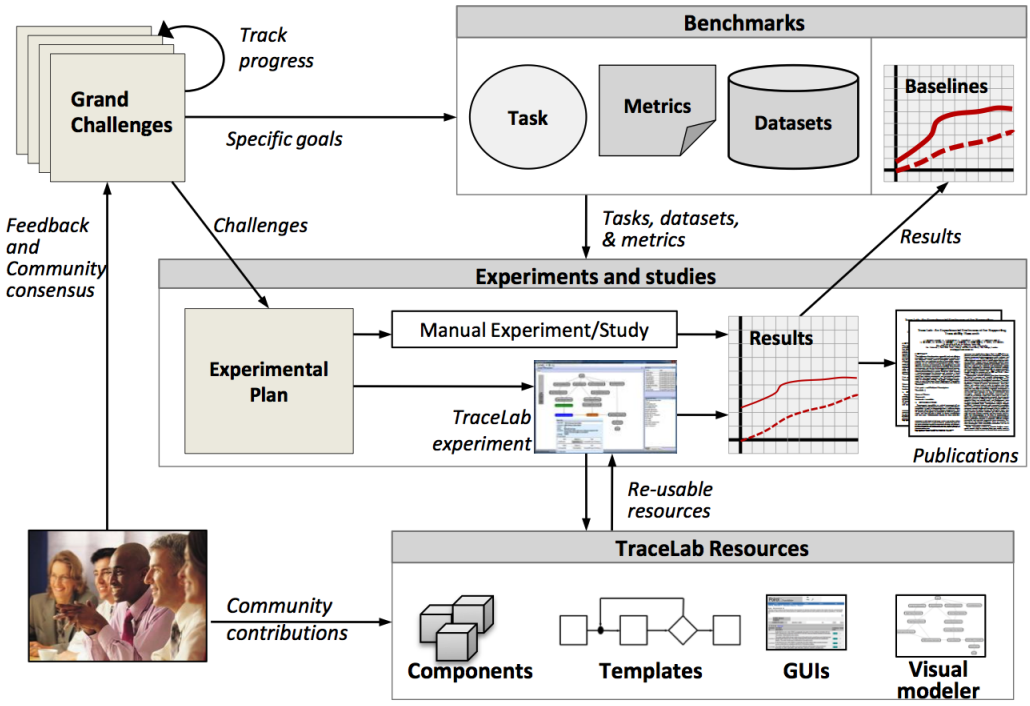

Figure 2.1: The Tracy Project [8].

to low level requirements. Many of the evaluations using bipartite datasets contained less than 500 artifacts. The SM calls for diverse datasets with a higher number of artifacts in order to capture the heterogeneous information landscape of a real software engineering project. For example, evaluations that trace not only requirements to requirements, but requirements to code, requirements to test cases, and even trace code to test cases are likely to have much more relevance to industry [6].

A lack of industrial datasets is not a problem unique to traceability. This issue affects a large portion of the software engineering community because the very nature of these datasets makes them the companies' most proprietary information. For example, source code files can be a significant portion of a dataset. Unless the project is already open source, by releasing its source code to the re-

search community, a company is also releasing it to competitors and customers, thus effectively giving its product away. The Tracy Project has made it a priority to find quality datasets so all researchers can better evaluate and compare their methods.

The Tracy Project also includes the process of defining specific benchmarks for each challenges' goals. By standardizing evaluations of approaches to goals through benchmarks, CoEST can track the progress of each goal, and thus each challenge. Each benchmark consists of a task, the metrics to compare, the datasets with which to compare metrics, and previous results or baselines. A benchmark's task includes the specific traceability goal, the motivation behind the benchmark, the grand challenge(s) the benchmark contributes to, and its current status [15, 8].

Finally, the Tracy Project has led the development of TraceLab, a research tool for traceability experiments. The goal of TraceLab is to promote collaboration in the community, improve reproducibility of experiments and reduce the effort required to establish a research environment. TraceLab is covered in greater detail in the next section.

2.3 TraceLab

TraceLab [8, 11] is a visual experimental workbench built to support the design, implementation, and execution of traceability experiments. It is designed to become the standard way by which empirical studies comparing tracing methods and approaches are conducted [13]. Several universities have contributed to the TraceLab project, including DePaul University, College of William and Mary, University of Kentucky, Kent State University and Cal Poly. It is funded by a

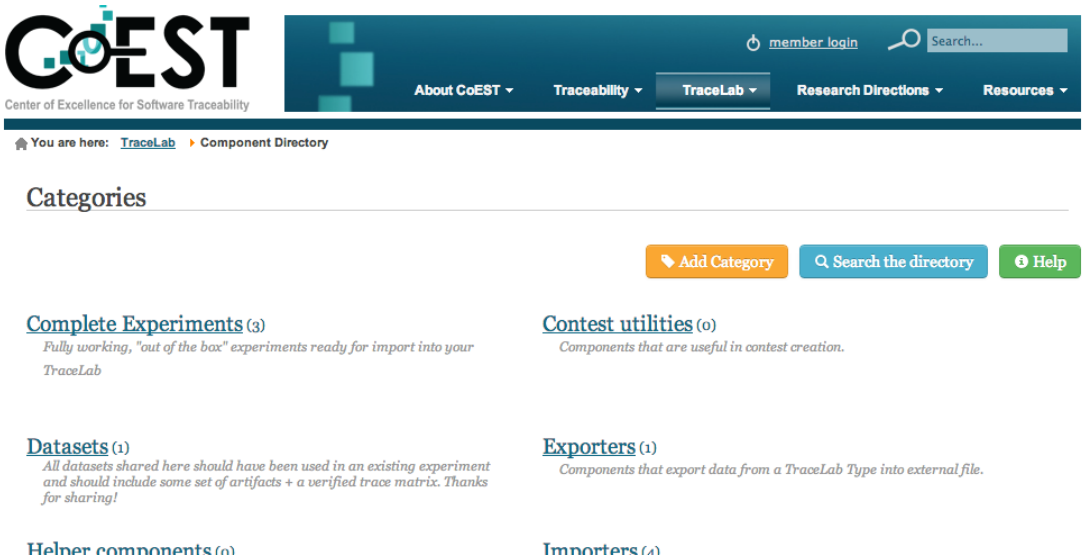

Figure 2.2: The CoEST.org community component directory [11].

National Science Foundation (NSF) Major Research Instrumentation grant.

Members of CoEST created TraceLab to address many of the problems limiting the advancement of traceability research, such as poor reproducibility of experiments, incomplete reporting of results, difficulty in establishing a research environment, and lack of collaboration in the community. TraceLab promotes collaboration through the CoEST website [11], a hub through which the traceability research community can share knowledge, methods and TraceLab experiments, shown in Figure 2.2.

For an experiment to be useful to the traceability research community, other researchers must be able to repeat the experiment and get comparable results. This can often be difficult due to poor reporting of procedures, methods used, or datasets [6]. Many factors contribute to poor reporting such as limited time because of paper deadlines, limited space because of length limits set by conferences, and non-disclosure agreements, which prohibit publishing information about proprietary datasets. These shortfalls not only affect reported procedures,

but reporting of results as well, making it difficult to compare results from one study to the next. By sharing TraceLab experiments through the CoEST website, researchers can ensure others in the community will be able to reproduce and, therefore, validate their work.

TraceLab also improves the ease of setting up a research environment. Researchers can use existing TraceLab experiments as a base for new research, saving a considerable amount of setup time. Even when a user starts a new experiment from scratch, they can reuse existing components in the component library to reduce boilerplate code such as reading datasets from file and preprocessing tasks like stemming.

2.3.1 Building TraceLab Experiments

A TraceLab experiment is a collection of components that, when executed, perform the procedure of a specific task. Each component represents a single action with strictly defined inputs and outputs. During execution, all inputs and outputs are resolved by the Workspace, a temporary data store. Components in an experiment connect to other components in a precedence graph. Execution begins at the start node and progresses through the precedence graph until all paths have reached the end node.

Once components have been placed in the graph, they must be configured. All outputs must be assigned a label and all inputs must select the label of a prior component's output. The data type of an input or output is strictly defined by the component. Many components also have configuration options that must be set, such as the file name to use for import or export components.

When all components have been placed, connected, and configured, the ex-

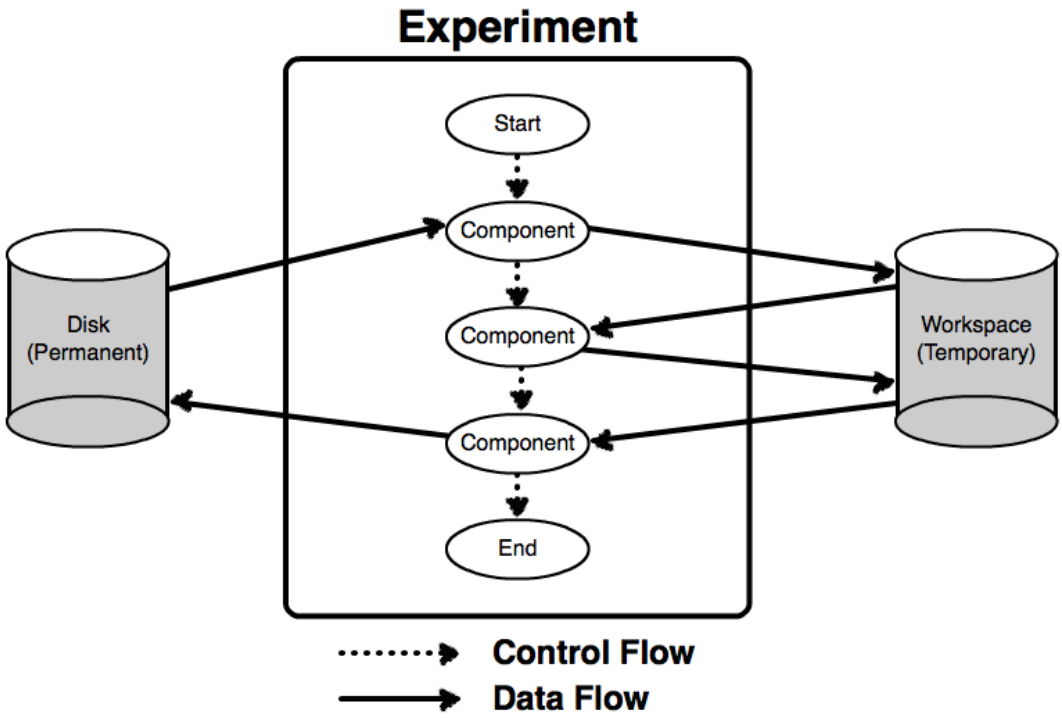

Figure 2.3: TraceLab architecture.

periment is ready for execution. When the play button is pressed, TraceLab highlights each component green as it executes. Errors appear in red and halt execution of the experiment. Upon successful completion of the experiment, users can view their data in its output file, through a visualizer component, or in the workspace.

2.3.2 TraceLab Components and Workspace Types

Components are the building blocks of any TraceLab experiment. All components can be found in the component library, which has all the standard components as well as any added component packages. The set of standard components includes the following types:

Importers read files into the TraceLab workspace.

Exporters write results to disk.

Preprocessors clean up and format the input data.

Tracers generate a set of trace links from input artifacts.

Postprocessors gather useful metrics from the results.

Visualizers display the results.

Helper components assist setting up the experiment.

Decision components support loop and if constructs.

Every component must strictly define its inputs and outputs, including their data types. Any data type that is compatible with the Workspace is known as a workspace type and may be used by components. TraceLab comes with several default workspace types:

- TLArtifact represents a single artifact with ID and text fields.
- TLArtifactsCollection is a collection of TLArtifact.
- TLSimilarityMatrix is a matrix of trace link confidence values between source and target TLArtifactCollection.

Users can expand their component library with custom components and workspace types. Components may be written in C#, Java, or C++. Users define inputs and outputs using annotations and access the Workspace through the given API. Users are encouraged to share components on the CoEST website [11].

2.4 Information Retrieval

Information retrieval (IR) is the process of finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need, a query, from within large collections (usually stored on computers) [28]. Prior

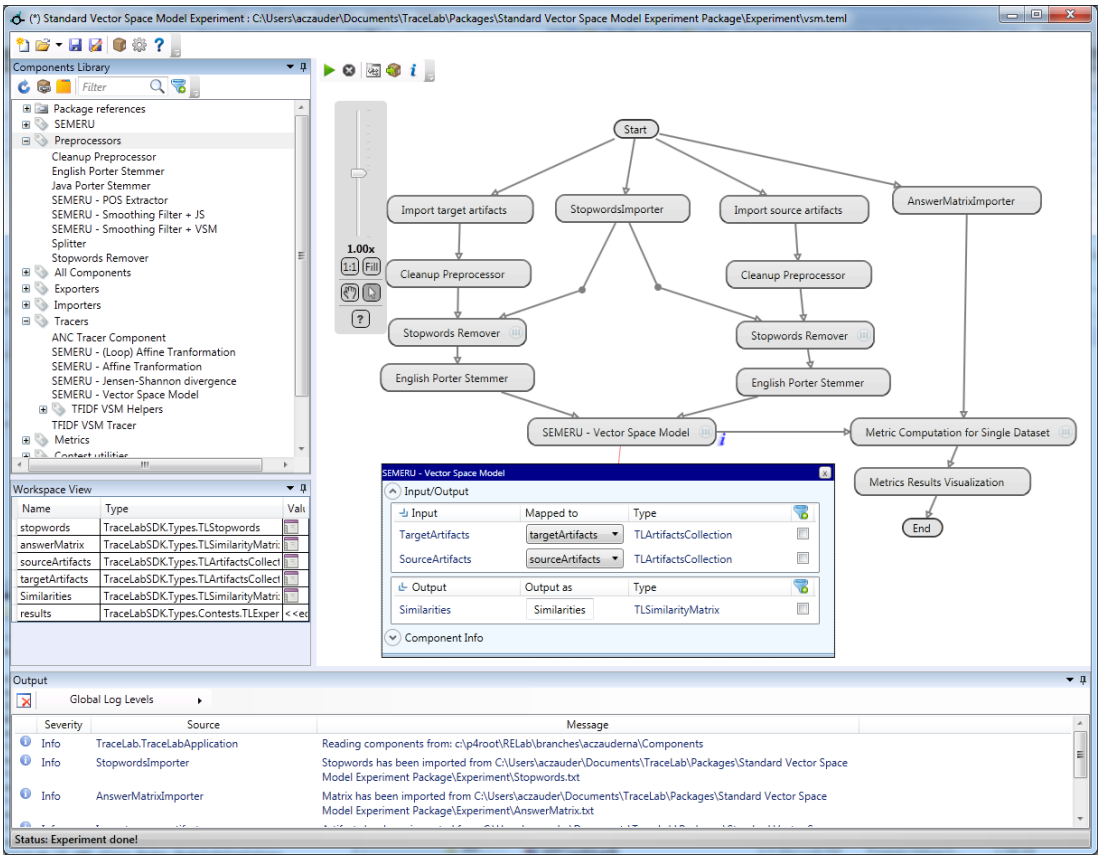

Figure 2.4: Sample traceLab experiment [11].

research [29, 18, 2, 10] has established IR as a feasible approach to automating traceability. To treat traceability as an IR problem, consider that each source artifact is a query, and the target artifacts are the collection of documents. Each relevant target artifact returned is considered a candidate trace link to the source artifact acting as the query.

2.4.1 Modeling

For the most part, information retrieval methods do not operate on raw text. The data is transformed from raw text to a vector of features by a series of preprocessors [37]. Most commonly, the input string is converted to an unordered

structure known as a bag of words. Let $V = \{k_1, ..., k_N\}$ be the vocabulary (set of keywords) of a given document collection D. Each document d is represented by a vector of feature weights, $d = (w_1, ..., w_N)$. Each feature weight is the number of times the keyword appears in the document, or some transformation thereof.

Several more steps may be applied to the bag of words model including stopword removal, stemming, and weight transformations. Stopword removal is process of omitting common words that appear in the majority of documents because they provide no semantic meaning, such as articles and prepositions.

There may be several duplicates of keywords in the vocabulary V due to differences in usage like singular vs. plural, or past vs. present. Stemming resolves these differences by truncating each word to a root form [28]. For example, 'console', 'consoled', 'consoling', and 'consolingly' can all be stemmed to the root 'consol'.

In a bag of words, raw keyword counts may not be the best weighting scheme [28]. Several term frequency transformations can be applied to the values of each document d in D to elicit a better representation of the original text $d' = (w'_1, ..., w'_N)$. Such transformations include boolean frequencies, logarithmically scaled frequencies, and augmented normalized frequencies.

Boolean term frequency: $w'_i = 1$ if $w_i > 0$ and 0 otherwise.
Logarithmically scaled term frequency: $w'_i = log(w_i + 1)$
Augmented normalized term frequency: $w'_i = 0.5 + \frac{0.5 * w_i}{max\{w\}}$

TF-IDF.

Although it is a transformation much like the list above, the term frequency - inverse document frequency (TF-IDF) transformation merits a more detailed de-

scription due to its popularity in automated tracing and other IR tasks. TF-IDF is different from the other transformations in that a term's weight is proportional to the inverse of that term's document frequency, the percentage of documents that contain the term. TF-IDF is used because it can reduce the weight of terms that are found in the majority of documents. These terms, much like stopwords, do not provide useful information to distinguish relevant from non-relevant documents.

$$w_i' = tf(w_i) * idf_i$$

where $tf(w_i)$ is one of the above term frequency transformations and

$$idf_i = log_2(\frac{|D|}{|\{d \in D : k_i \in d\}|})$$

In the end, the raw input string of each document is turned into a vector of features that can be used in IR computations. The methods shown here are only a small subset of the possible ways to process raw text into a more machine friendly format. Given the popularity of TF-IDF and bag of words models in traceability, we have chosen to focus on these methods [37].

2.4.2 Measures

We must establish metrics to assess the quality of automated tracing approaches. To measure accuracy, we use the standard metrics of precision, recall, and their harmonic mean f-measure. Huffman Hayes et al. [21] also define two metrics, selectivity and lag, that are useful to automated tracing. Table 2.1 shows acceptable levels for recall, precision, and lag according to Hayes et al., based on their industry experience

Consider a tracing task with a set of high level requirements H traced to a set of low level requirements L. There are a total of $|H| * |L|$ possible links. The job

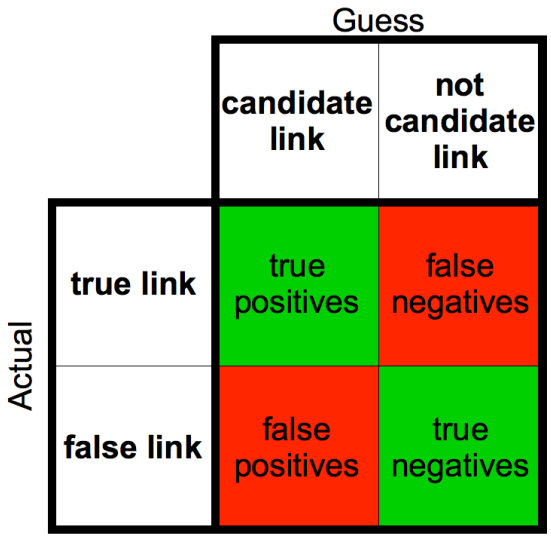

Figure 2.5: Possible tracing outcomes.

of automated tracing is to make a binary decision on each possible link whether it is a candidate link or not. This leads to one of four outcomes, see figure 2.5.

True Positive (TP): A true link that was correctly identified as a candidate link.

False Positive (FP): A false link that was incorrectly identified as a candidate link.

False Negative (FN): A true link that was incorrectly identified as not a candidate link.

True Negative (TN): A false link that was correctly identified as not a candidate link.

Recall.

Recall is the percentage of correct links that are retrieved [21]. Recall is affected by errors of omission, a true link that was not found.

$$recall = \frac{TP}{TP + FN}$$

Metric	Acceptable	Good	Excellent
Recall	60-69%	70-79%	80-100%
Precision	20-29%	30-49%	50-100%
Lag	3-4	2-3	0-2

Table 2.1: Acceptable levels for recall precision and lag [21]

Precision.

Precision is the percentage of retrieved links that are correct [21]. Precision is affected by errors of commission, a false link that was incorrectly included.

$$precision = \frac{TP}{TP + FP}$$

F-measure.

F-measure is the harmonic mean of recall and precision. It can be weighted toward either precision or recall. $b = 1$ weighs recall and precision the same, whereas $b < 1$ favors precision and $b > 1$ favors recall. As a rule, human analysts are better at detecting errors of commission over errors of omission [21]. For these reasons, we favor recall and use $b = 2$.

$$f_b = \frac{(1 + b^2) * precision * recall}{b^2 * precision + recall}$$

$$f_2 = \frac{5 * precision * recall}{4 * precision + recall}$$

Probability of Failure.

Probability of Failure (PF) is the percentage of false links that are retrieved. It can be thought of as a complement of Recall.

$$pf = \frac{FP}{FP + TN}$$

Selectivity.

Selectivity is the percentage of possible links that are candidate links [21]. Selectivity is not a measure of the quality of candidate link generation, but rather of the effort reduction on the part of the user who must vet all candidate links. Compared to manual tracing, where all possible links are candidate links, automated tracing can save user effort by presenting a smaller set of candidate links. Selectivity measures how much smaller the candidate link set is relative to manual tracing. Smaller values are better because they indicate fewer links must be evaluated.

$$selectivity = \frac{TP + FP}{|H| * |L|}$$

The outputs of IR methods are not generally binary decisions as modeled above. Rather, IR methods often return a continuous distribution of confidence values. Continuous distributions can easily be made into binary decisions by the use of a cutoff value. Once a cutoff value is selected, the possible links with a confidence score higher than the cutoff become candidate links, and the possible links below the cutoff value are not candidate links. Our next metric, lag, evaluates IR methods by examining confidence values, not just binary decisions like the previous metrics.

Lag.

Lag measures the separation of true (TP) and false (FP) candidate links. Let T be the set of true links in the set of candidate links. For each link t in T, $lag(t)$ is the number of false links with a higher confidence value than t. The overall lag is the average of the individual lags for each true link in the candidate link set.

$$lag = \frac{\sum_{t \in T} lag(t)}{|T|}$$

2.4.3 Automated Traceability and Information Retrieval

Current research in automated tracing using information retrieval methods has proven to be effective and efficient at generating candidate trace links [29, 18, 2, 10]. Although these techniques are not yet reliable enough to be used as fully automated solutions, they can provide significant time and effort savings in a semi-automated environment, where automatically generated candidate RTMs are verified by human analysts [12]. Current attempts and their results are summarized below.

Antoniol et al. [2] used term frequency - inverse document frequency (TF-IDF) [37] and a probabilistic IR method [36] to trace source code to documentation and requirements. In their first experiment, performed on LEDA (Library of Efficient Data Types and Algorithms), they traced C++ classes to manual pages using this method. This dataset included 95 KLOC, 208 classes, and 88 manual pages. Their second experiment, performed on the Albergate system, traced 60 code classes to 16 functional requirements. In both experiments, accuracy was reported in terms of recall and precision, which measure the percentage of correct links that were found, and the percentage of links found that were correct,

respectively. Antonial et al. achieved high levels of recall (86-100%) but low precision (6-19%) for both methods.

Marcus and Maletic [29] applied latent semantic indexing (LSI), another IR technique, to trace from documentation to source code. Using the same datasets as Antoniol et al. [2], LSI achieved comparable recall (96-97%) and higher precision (18-25%) for LEDA, partially due to the inclusion of source code comments. For the Albergate dataset, recall and precision were comparable to both the TF-IDF and probabilistic model (91-100% and 16-17% respectively).

Hayes et al. [18, 22, 21] used TF-IDF, TF-IDF with a simple thesaurus, and LSI for requirements to requirements traceability. Using an automated tracing tool they built called RETRO [20], the authors also incorporated user feedback analysis to improve accuracy. The studies by Hayes et al. were performed on the MODIS [32] and CM-1 [30] datasets. Recall and precision for the MODIS dataset were 63% and 39% respectively for the TF-IDF approach. When the simple thesaurus was added, recall increased to 85% and precision remained comparable at 40%. When user feedback was added, recall increased to 90% and precision increased to 80%. It should be noted that by soliciting user feedback, this is no longer a fully-automated technique and thus results cannot really be compared.

Cleland-Huang et al. [10] automated tracing of non-functional requirements using a probabilistic retrieval alorithm. Non-functional requirements were traced to UML class diagrams for the Ice Breaker System. Non-functional requirements were modeled as a Softgoal Interdependency Graph. The authors trained their methods on a subset of the data and achieved 87% recall and 51% precision on the full dataset. The authors have used their techniques to build an automated tracing tool Poirot [27] and study impact change analysis [10].

2.5 Machine Learning

Machine learning (ML) was defined by Arthur Samuel [38] as the "field of study that gives computers the ability to learn without being explicitly programmed." It is a branch of artificial intelligence (AI) concerned with making predictions about new data based on trends in prior data. We will concentrate on two categories of ML algorithms, classification and clustering.

Machine learning algorithms generally come in one of two flavors, supervised or unsupervised [42]. Supervised algorithms, such as most classifiers, require a training set of labeled data that the algorithm uses as a gold standard. The algorithm is then presented with a test set of unlabeled data for which decisions must be made, extrapolating from the training set examples. Unsupervised algorithms on the other hand, such as most clusterers, do not require a training set and can perform their function on a test set without a gold standard to base decisions on.

2.5.1 Classification

The goal of classification is to determine from $C = \{c_1, ..., c_n\}$, a set of categories, which category c best fits a given data point d [42]. Classification algorithms, known as classifiers, are generally supervised algorithms, where each data point in the training set $T = \{(d, c)|c \in C\}$ is labeled according to the category it best fits. The classifier then makes a prediction about d by extrapolating from the data in the training set. For example, a classifier of animals has the set of categories {dog, cat}. The classifier needs to train on examples of known dogs and cats. It would then attempt to label a test set of unknown animals as either a dog or a cat. The classification algorithms used in this work are [42, 43, 7]:

1. Naive Bayes
2. Bayesian Network
3. Support Vector Machines (SVM) - Sequential Minimal Optimization (SMO)
4. K Nearest Neighbors (KNN)
5. C4.5 Decision Tree - J48
6. BPNB

BPNB.

The first 5 algorithms in the list are very well documented [42, 43] and will not be covered here. BPNB however, requires a bit more explanation as it is not as widely known. Developed by Chu [7] for Chinese text classification, BPNB is an adaptation of Naive Bayes that accounts for the relative probability of each feature. Given a set classes C and a set of documents D where each document is a collection of features $d = (w_1, ..., w_n)$.

The algorithm states that $Pr(c|d)$, the probability of a given document d has a certain class c, is defined as:

$$Pr(c|d) = Pr(c) * \prod_{i=1}^{n} g(w_i, c)$$

Where $g(w_m, c)$ is the weight of feature w_m in class c and $Pr(c)$ is the probability of class c, determined by a training set $T = \{(d, c)|c \in C\}$.

$$Pr(c) = \frac{|\{d|(d, c) \in T\}|}{|T|}$$

$$g(w_m, c) = \beta^{1-\frac{Pr(f_m|c)}{Ave(f_m)}}, 0 < \beta < 1$$

$$Ave(f_m) = \frac{\sum_{i=1}^{|C|} Pr(f_m|c_i)}{|C|}, c_i \in C$$

For each document d in a test set, the classifier calculates the probability of each class c in C and selects the class with the largest probability.

2.5.2 Clustering

The goal of clustering is to group data points so that each data point is more similar to others in its group than it is to data points in other groups. Clustering algorithms, known as clusterers, are generally unsupervised algorithms. This means that a set of data points can be clustered with no prior knowledge of the data set. The clustering algorithms used in this work are [42, 43, 23]:

1. K Means
2. Hierarchical Tree
3. Expectation Maximization (EM)
4. Cobweb
5. Farthest First

Each of these clustering algorithms is well documented [42, 43, 23] and the specifics of the algorithms will not be covered here.

2.5.3 Automated Traceability and Machine Learning

Although machine learning algorithms have not been used for automated tracing as extensively as IR methods, there have been several applications of ML to improve automated tracing methods. Wieloch et al. [41] present trace-by-classification (TBC), a machine learning approach to classifying non-functional requirements into categories such as fault tolerance, performance, security, etc. Their classifier has also been used to trace regulatory codes to requirements [9] and trace quality concerns via architectural tactics to code [31].

Chuan Duan et al. [14] used clustering to enhance their automated trace-

ability tool Poirot. Their focus, however, was to improve the human side of traceability with clustering. Once the tool generated a set of candidate trace links using IR techniques, they were clustered in order to present them to the user in a more coherent form. Through clustering, the authors were able to reduce the effort required to verify a set of candidate trace links and provide better context when evaluating links.

These studies show the potential machine learning has to improve automated tracing methods. We hope to encourage more research in this avenue through our contribution of a machine learning package for TraceLab.

2.5.4 WEKA

WEKA (Waikato Environment for Knowledge Analysis) [40, 17] is a library of machine learning algorithms assembled by the Machine Learning Group at University of Waikato, New Zealand. WEKA is freely available under the GNU General Public License. Written in Java, WEKA can be used through its graphical interface, command line, and programmatic API. The machine learning library contains an extensive set of algorithms including classification, clustering, preprocessing, attribute selection, and visualization. Whenever possible, our contributions are built upon the algorithms in WEKA.

Attribute-Relational File Format (ARFF)

WEKA uses the attribute-relational file format (ARFF) to represent data as text. ARFF is very flexible, well defined, and easy to use. ARFF text files consist of two sections, header and data. The header contains a relation tag and a list of attributes. The relation tag captures the history of changes made to the data.

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature numeric
@attribute humidity numeric
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
...
```

Figure 2.6: Example ARFF file.

Each attribute in the list has a unique name and a type, which can be any of the following: numeric, string, date, or nominal. Each nominal attribute must also specify its possible values.

Next is the data section. Each line in the data section is a set of comma separated values corresponding to a single entry. The data section of an ARFF file can be dense or sparse. Given an ARFF file with N attributes, for dense representation each entry in the data section has exactly N values where the index of value matches the index of the attribute. In a sparse representation, the data section has up to N values, where each value is preceded by the index of the attribute it corresponds to. The remaining attribute values are set to null.

All of the algorithms in WEKA use ARFF files and its corresponding Java object weka.core.Instances. An example ARFF file is shown in Figure 2.6.

CHAPTER 3

TraceLab Machine Learning Component Package

This section is an overview of the machine learning component package for TraceLab. The component package will be shared with the TraceLab community through the CoEST website. Anyone interested in using machine learning algorithms in their TraceLab experiment need only retrieve the package from the website and include it in their component library.

The long-term success of TraceLab is largely dependent on the quality of its component library. As the building blocks of any experiment, a comprehensive set of components is paramount to TraceLab providing utility to its users and gaining traction within the traceability research community. Given that machine learning algorithms have already been proven useful to traceability efforts [14, 41] and have a wide range of other applications, CoEST has requested that we create a machine learning component package.

3.1 Writing Code for TraceLab

TraceLab is designed to be very extensible through the creation of components and data types, also known as workspace types. Everything needed to create custom components and workspace types are found in the TraceLabSDK DLL library that comes with every TraceLab installation. Users can also link the TraceLabSDK.Types DLL library for access to all of TraceLab's standard workspace types

(see section 2.3). TraceLab supports components written in C#, Java, and C++.

3.2 Using WEKA

The component package in this work is primarily built upon the Waikato Environment for Knowledge Analysis (WEKA) [17] implementations of machine learning algorithms. Using WEKA has many advantages over in house implementations of the machine learning algorithms. First is the time and cost savings of not having to research and implement the algorithms themselves. Also, WEKA's algorithms are more reliable because they are open source, widely popular and the Unviersity of Waikato has been working on WEKA for 20 years [40]. Another benefit of WEKA is their attribute-relational file format (ARFF), which is very flexible, well defined, and easy to use. By using WEKA, we need only worry about whether our usage of the algorithm is correct, and not worry about the algorithm itself.

While the decision whether to use WEKA was very simple, the question of how to use WEKA presented challenges. Namely, given that TraceLab is a C# system and WEKA is a Java library poses some difficulties. The connection between Java and C# is done through a tool called IKVM.NET [24]. IKVM is a .NET (C#) implementation of a Java VM. IKVM supports interoperability in both directions meaning users can use Java objects and functions in .NET code as well as use .NET objects in Java code.

Since the algorithms we need are written in Java, Java may seem like the obvious choice for writing components. For the most part, IKVM does a great job of providing a seamless interface between C# and Java code, but a few small issues made a big impact causing Java components to be nearly infeasible. For

```
public void ReadData(BinaryReader reader)
{
   idIndex = reader.ReadInt32();
   int classIndex = reader.ReadInt32();
   string arff = reader.ReadString();

   if (arff.Length > 0)
   {
      instances = new weka.core.Instances(new java.io.StringReader(arff));
      instances.setClassIndex(classIndex);
   }
}
```

Figure 3.1: Sample C# code using Java objects with IKVM [24].

one, IKVM has very little support for generics. As a result, TraceLab must include several seemingly extraneous data types, such as ListOfStrings, that are wrappers of their corresponding generic, List¡strings¿, and are needed for Java developers. This can work in most cases, but existing usage of generics triggered compatibility problems. We also ran into issues with TraceLab's component scanner when using Java classes as configuration types.

In the end, we found that C# components had many advantages over Java components. For one, the complex interactions between components and TraceLab's component scanner inherently work better in C#. Also, debugging within TraceLab worked best with .NET components. Additionally, WEKA objects worked nearly seamlessly within C# classes. Although generics in WEKA objects still posed some difficulty, we were always able to find a reasonable workaround by using a different function. As a point of style, we refrain from importing Java classes, opting instead to refer to the full package name to make Java objects easier to recognize in code, see Figure 3.1.

```
using TraceLabSDK;

namespace TLWeka.Types
{
   [Serializable]
   [WorkspaceType]
   public class TLDoubleTree : IRawSerializable
   {
      /// <summary>
      /// Mandatory Parameterless constructor
      /// </summary>
      public TLDoubleTree() : base()
      {
         ...
      }
      ...
   }
}
```

Figure 3.2: Sample code for a workspace type.

3.3 Data Types

TraceLab supports any data type for the inputs and outputs of components. However, data types that are specifically labeled as a workspace type have several advantages in TraceLab. Workspace types are recognized by the component scanner for better compatibility with experiments. In addition, workspace types can achieve better performance than standard data types through more efficient serialization.

Creating workspace types.

Creating a data type for TraceLab is straightforward. The TraceLabSDK DLL library includes the TraceLabSDK.WorkspaceType attribute tag, used to denote a workspace data type. Once the attribute has been applied, the class will automatically be recognized by the Workspace. The data type must also have a default parameterless constructor and a System.Serializable attribute applied to it. TraceLab also supports the creation of Windows forms to visualize the data

Data Types
• TLarff: Attribute-relation file format (ARFF) representation for WEKA components. • TLCluster: A single cluster with a cluster ID and a list of artifact IDs with optional confidence values. • TLClustersCollection: A collection of TLCluster. • TLDoubleTree: A tree of double values where each node has a label, value, and 0 or more children. • TLConfusionMatrix: Adapted from prior work [41] to summarize the results of classifiers.

Table 3.1: Machine learning component package data types.

types within TraceLab. An example data type is shown in Figure 3.2.

Serialization.

The TraceLabSDK library also includes several interfaces for serialization. The two most common interfaces are IRawSerializable and IXMLSerializable. IRawSerializable includes a pair of functions WriteData and ReadData. Similarly, IXMLSerializable includes WriteXML and ReadXML.These functions give users an opportunity to perform custom serialization that is faster than standard serialization. Since objects are serialized and deserialized between every component, efficient serialization is critical for workspace types. If a workspace type does not use any of the TraceLabSDK serialization interfaces, the standard serialization is used, so long as the System.Serializable attribute is applied. IXMLSerializable is also used for importing and exporting XML files.

The following workspace types are a contribution of the machine learning

Components	
Importers: Bring data into Workspace.	
• TLarff ARFF Importer	• TLClusterCollection XML Importer
• TLDoubleTree Newick Importer	
Exporters: Save data from Workspace.	
• TLarff ARFF Exporter	porter
• TLDoubleTree Newick Exporter	• TLConfusionMatrix CSV Exporter
• TLClusterCollection XML Ex-	
Classifiers: Assign categories to artifacts.	
• Naive Bayes Classifier	• KNN Classifier
• Naive Bayes Cross-Validation	• KNN Cross-Validation
• Bayes Net Classifier	• SMO Classifier
• Bayes Net Cross-Validation	• SMO Cross-Validation
• J48 Classifier	• BPNB Classifier
• J48 Cross-Validation	• BPNB Cross-Validation

Table 3.2: Machine learning component package components.

Components	
Clusterers: Group similar artifacts.	
• Hierarchical Clusterer	• K-Means Clusterer
• Hierarchical Cluster Tree	• Cobweb Clusterer
• EM Clusterer	• Farthest First Clusterer
• EM Fuzzy Clusterer	
Preprocessors: Prepare data for processing.	
• Convert Artifacts to TLarff	• Normalize Tree Branch Lengths
• Slice Cluster Tree at Cutoff	• Merge Clusters of Artifacts
• Slice Cluster Tree into Clusters	• Get Test Set
• Bag of Words - Data Set	• Get Train Set
• Bag of Words - Test/Train Sets	• Randomize Order of Instances
• TF-IDF - Data Set	• Add Prefix to Cluster Names
• TF-IDF - Test/Train Sets	
Postprocessors: Cleanup and gather results.	
• Generate Confusion Matrix	dence
• Remove Clusters Below Confi-	• Remove Artifacts In Class
Helper Components: Perform simple tasks to setup Workspace.	
• Double Writer	• Get Cluster from Collection
• Increment Double by Value	• Count Number of Clusters
• Get TLarff Class Index	

Table 3.3: Machine learning component package components (Continued).

component package. They are summarized in Table 3.1.

3.3.1 TLarff

The first step was to create a TraceLab workspace type that is compatible with the WEKA library. For this, we created the TLarff workspace type, a C# wrapper around the WEKA Java object weka.core.Instances. Instances objects are the Java representation of ARFF text files (see Section 2.5.4) and are used by all of the algorithms in WEKA.

The primary purpose of the TLarff data type is to ensure the underlying weka.core.Instances object is properly serialized to and from the Workspace. For the most part, this is easily accomplished by using ARFF text and letting WEKA handle all generation and parsing of the text. However, not all of the information stored in a weka.core.Instances object is explicitly written to an ARFF file, and therefore must be serialized separately by the wrapping TLarff class.

One such case is the optional class attribute. The class attribute is a nominal or numeric type representing the possible labels a classifier can assign. One attribute can be designated the class attribute. Proper identification of the class attribute is critical to the operation of all WEKA classifiers. However, even though weka.core.Instances objects track the index of the class attribute, ARFF files do not. Therefore, we track of the index of the class attribute in the TLarff object so that it is not lost when serializing.

Another feature unique to TLarff is an optional ID attribute. Although it is not found in WEKA, ID attributes are a common use case and have special properties. An ID attribute must maintain its integrity throughout the life of the object. Users must protect the ID attribute against any function that would

attempt to modify it or strip it away. ID attributes must also be isolated from other attributes.

For example, given a set of artifacts has an ID attribute of type string and a classifier based on numeric vectors. Unless the ID attribute is isolated from the classifier, the classifier will fail when it finds a non numeric attribute. In addition, because many components rely on the ID attribute, the output of the classifier must still have the ID attribute in tact, with each entry having the correct, unmodified ID.

3.3.2 TLCluster and TLClustersCollection

The purpose of workspace types TLClustersCollection and TLCluster is to store the results from clusterer components. TLClustersCollection is a list of type TLCluster. Each TLCluster contains a cluster ID and a list of the artifact Ids that are in the cluster. Each artifact in the cluster can also be assigned a confidence value, the probability that the artifact belongs in that cluster. TLClustersCollection and TLCluster support serialization via IXMLSerializable and IRawSerializable.

3.3.3 TLDoubleTree

TLDoubleTree is a self referential tree of double values. Each TLDoubleTree contains a string label, double value, and a list of TLDoubleTree children. Leaf nodes are denoted by an empty list. For serialization, TLDoubleTree uses the Newick tree format [3], a concise text representation for trees. Serialization is done through IRawSerializable.

3.3.4 TLConfusionMatrix

TLConfusionMatrix is a workspace type that has been adapted from the Trace By Classification TraceLab experiment by Wieloch et al. [41]. A TLConfusionMatrix is used to display the results of a classifier.

3.4 Components

Components are the building blocks of any TraceLab experiment. The component contributions in this package are primarily classifier and clusterer components, as well as a collection of other components to facilitate building experiments. This work contributes a total of 47 TraceLab components, shown in Table 3.2.

There may be additional components that are used in the experiments performed in this work that have not been listed here as a contribution of the component package. It is because those components tend to lack the broad applicability that is ideal for a component library. They are useful serving a narrow purpose in a specific setting, but would not provide utility in many other scenarios, so they are omitted from this section. The components below strive for versatility and applicability. Thus, design decisions favor a wider set of use cases.

Creating components.

TraceLab components must extend the abstract class TraceLabSDK.BaseComponent and apply the TraceLabSDK.Component attribute in order to be registered with the component library by the component scanner. The Component attribute consists of the following properties:

```
[Component(Name = "TF-IDF Filter",
           Description = "Converts attributes to Term Frequency - Inverse Document Frequency vectors",
           Author = "Cal Poly - Brett Armstrong",
           Version = "1.0",
           ConfigurationType = typeof(TFIDFFilterConfig))]
[Tag("Filters.TLarff")]
[IOSpec(IOType = IOSpecType.Input, Name = "TLarff", DataType = typeof(TLarff))]
[IOSpec(IOType = IOSpecType.Output, Name = "TLarff", DataType = typeof(TLarff))]
public class TFIDFFilter : BaseComponent
{
   private TFIDFFilterConfig config;

   public TFIDFFilter(ComponentLogger log) : base(log)
   {
      config = new TFIDFFilterConfig();
      Configuration = config;
   }

   public override void Compute()
   {
      ...
   }
}
```

Figure 3.3: Sample code for a component.

- Component name
- Label
- Description
- Author
- Version
- Configuration type

The component class may also have several additional attributes added. The TraceLabSDK.Tag attribute tells TraceLab what category in the component library the component belongs. Without a TraceLabSDK.Tag attribute, the component will be "Uncategorized". The component also uses a TraceLabSDK.IOSpec attribute to specify each of the inputs and outputs of the component. Inputs and outputs have a name, description, and data type. An example component is in Figure 3.3.

Configuration.

Configuration is a way to set the static properties of a component that do not depend on input. For example, common configuration options are file

paths for input and output, or the maximum number of iterations of an algorithm. A configuration class is a simple class with public properties of either a primitive type, an enum type, or one of two special classes provided in the TraceLabSDK for browsing directories: FilePath and DirectoryPath. Each public property of the configuration class becomes an option on the info panel of a component. The attributes System.ComponentModel.DisplayName and System.ComponentModel.Description can be added to each property to guide users about what the options do. An example configuration class can be seen in Figure 3.4.

The entry point for all components is the Compute() method that must be overridden from the abstract superclass BaseComponent. Per recommendations from CoEST, the contents of the Compute() method should handle all interactions with the workspace, and delegate as much work as possible to utility classes that can be run and tested outside of TraceLab. This also has the benefit of reducing code when multiple similar components are needed, which commonly occurs when the same operation is needed for more than one combination of inputs and outputs.

3.4.1 Importers

TraceLab experiments rely on import components to read data from the disk to the workspace. Importers generally start off any experiment, bringing everything needed to run such as data sets, previously generated clusters, or answer sets. Importers take no Workspace inputs and have one output, the data that was imported. The component package includes three Importers.

```
using System.ComponentModel;
using TraceLabSDK.Component.Config;

namespace TLWeka.Components.Config
{
   class OutputFileConfig
   {
      public OutputFileConfig() : base() { }

      [DisplayName("Output File")]
      [Description("File to output")]
      public FilePath outputFile
      {
         get;
         set;
      }
   }
}
```

Figure 3.4: Sample code for a configuration class.

Configuration.

All three Importers use a FilePath to select which file to import. The ARFF file to TLarff component also accepts two additional configuration options, Class Index and ID Index. Each of these options is an integer corresponding to the indexes of the class and ID attributes. As both, the ID and class attributes are optional, -1 may be used to indicate that the attribute does not exist.

3.4.2 Exporters

Exporters copy data from the workspace to disk. They are generally found at the end of experiments when the results are ready to be stored. As the opposite of Importers, Exporters have no Workspace outputs and take only one input, the data to write to disk. The output file is determined by a FilePath configuration option. The component package includes four Exporters.

3.4.3 Classifiers

This package contributes 12 classifier components using 6 different classification algorithms, shown in Table 3.4. Five of the algorithms reference the WEKA implementation. The sixth algorithm, BPNB, has been implemented in this package by the authors (see Section 2.5). There are two types of components for each algorithm, classifier and cross-validation.

The classifier components take as input a labeled training set and an unlabeled test set. Both inputs are of type TLarff. The classifier learns the training set and classifies the test set. The output is the TLarff test set with classifier-assigned labels added in an additional attribute. The index of the attribute is returned through an output parameter.

The cross-validation components differ from classifiers in that they accept a training set only. The training set is classified by training and testing several classifiers on subsets of itself in a process known is cross-validation. The output is the training set with an additional attribute of classifier-assigned labels. As with the classifier componentes, the index of the attribute is returned through an output parameter.

Cross-Validation.

Cross-validation is performed by dividing the dataset D into N folds and training N classifiers. For each classifier, one fold is selected as the test set and the remaining $N - 1$ folds are combined into the training set. In the end, each fold has been tested once, and been in the training set $N - 1$ times. The test results of each fold are aggregated and returned. Cross-validation has a minimum

Classifiers	
Naive Bayes Cross-Validation, Bayes Net Cross-Validation, J48 Cross-Validation, KNN Cross-Validation, SMO Cross-Validation	
Inputs • TLarff: Labeled Data Set Configuration • Integer: Number of Folds	Outputs • TLarff: Labeled Data Set • Integer: Result Attribute Index
Naive Bayes Classifier, Bayes Net Classifier, J48 Classifier, KNN Classifier, SMO Classifier	
Inputs • TLarff: Labeled Train Set • TLarff: Test Set	Outputs • TLarff: Labeled Test Set • Integer: Result Attribute Index
BPNB Cross-Validation	
Inputs • TLarff: Labeled Data Set Configuration • Integer: Number of Folds • Double: Beta	Outputs • TLarff: Labeled Data Set • Integer: Result Attribute Index
BPNB Classifier	
Inputs • TLarff: Labeled Train Set • TLarff: Test Set Configuration • Double: Beta	Outputs • TLarff: Labeled Test Set • Integer: Result Attribute Index

Table 3.4: Classifier Component Descriptions.

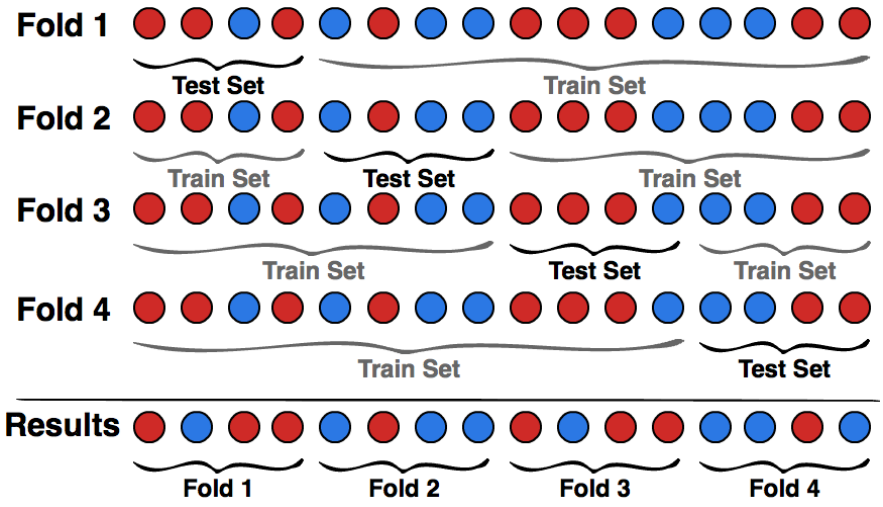

Figure 3.5: Sample 4 fold cross-validation process.

of 2 folds and a maximum of $|D|$ folds.

Configuration.

All five algorithms found in WEKA share the same configuration. The classifier components do not require any configuration options and the cross-validation components only require the user to configure N, the number of cross-validation folds to perform. The BPNB components differ in that they each have an additional configuration option, β (see Section 2.5).

Construction.

The classifier components in this package are backed by a common classifier utility class. Each component class performs a few setup operations, then passes the execution of the classifier to the utility class. Setup includes importing the data from the Workspace, constructing the classifier that will be used, and setting up any configuration options, like the number of folds for cross-validation, or the

value of the parameter β for BPNB. The common utility class is used by all 12 classifiers.

3.4.4 Clusterers

The component package in this work contributes 7 clusterer components to the TraceLab community, shown in Table 3.5. All 7 components use WEKA implementations of 5 different clusterer algorithms. Five of the clusterer components accept an input TLarff dataset, group the entries into clusters of similar artifacts, and return the clusters as a TLClustersCollection.

The other two components execute slightly differently. The EM Fuzzy Clusterer has the same inputs and outputs as the other clusterers, but differs because each data point may appear in more than one cluster. The output also contains the confidence values of data points in each cluster. The Hierarchical Cluster Tree component also takes as input a TLarff dataset, but the output is the hierarchy tree generated by the hierarchical clustering algorithm, rather than the clusters themselves.

Configuration.

Configurations vary widely from component to component because clusterers have many algorithm-specific options. The Hierarchical Clusterer has three configuration options, the desired number of clusters, the distance function to use, and the link type. The Hierarchical Cluster Tree component accepts the same options, except that it does not require a target number of clusters.

Clusterers	
Hierarchical Clusterer	
Inputs	Configuration
• TLarff: Dataset	• Double: Number of Clusters
Outputs	• Enum: Distance Function
• TLClustersCollection	• Enum: Link Type
Hierarchical Cluster Tree	
Inputs	Configuration
• TLarff: Dataset	• Enum: Distance Function
Outputs	• Enum: Link Type
• TLDoubleTree: Cluster Tree	
EM Clusterer, EM Fuzzy Clusterer	
Inputs	Configuration
• TLarff: Dataset	• Integer: Max Iterations
Outputs	• Integer: CV Folds
• TLClustersCollection	• Integer: Max Clusters
K-Means Clusterer, Farthest First Clusterer	
Inputs	Configuration
• TLarff: Dataset	• Integer: Num Clusters
Outputs	
• TLClustersCollection	
Cobweb Clusterer	
Inputs	Configuration
• TLarff: Dataset	• Double: Acuity
Outputs	• Double: Cutoff
• TLClustersCollection	

Table 3.5: Clusterer Component Descriptions.

Link Type Enum:

- Single
- Complete
- Average
- Mean
- Centroid
- Ward
- Adjcomplete
- Neighbor Joining

Distance Function Enum:

- Euclidean
- Manhattan
- Chebyshev

Both EM Clusterer and EM Fuzzy Clusterer components use the same four configuration options. Expecation Maximization is an iterative algorithm; the first option is the maximum number of iterations the algorithm may perform. The next option is the ideal number of clusters to create. This is optional because the EM algorithm can automatically detect the correct number of clusters using cross-validation. There are two additional options if cross-validation is used: the number of folds and the maximum number of clusters to consider.

K-Means Clusterer and Farthest First Clusterer also share common configuration options. Each of these algorithms must be told how many clusters to generate, which is specified through a configuration option. The final clusterer component, Cobweb Clusterer, has two configuration options for the control parameters of the Cobweb algorithm: Acuity and Cutoff.

Construction.

Each clusterer component is backed by a common clustering utility class. The component classes handle communicating with the Workspace, initializing

the clusterer with any algorithm specific configuration options, and then passing the work off to the utility class. Using the common utility class reduces code duplication and simplifies testing.

3.4.5 Preprocessors

The components that clean up, format, and prepare the data for classifiers, clusterers, or tracers are known as Preprocessors. This component package contributes a wide variety of preprocessors such as data conversion, modeling and randomization. The Preprocessors in this package are shown in Tables 3.6 and 3.7.

Convert Artifacts to TLarff.

First, we need a data conversion component to connect the standard TraceLab workspace type TLArtifactsCollection to the WEKA components. Each artifact in the collection has an ID and Text field that are converted to TLarff attributes. The ID field is marked as the ID attribute to preserve its values.

Bag of Words and TF-IDF.

There are four text modeling preprocessors for TLarff datasets, shown in Table 3.6. All four components are built upon the WEKA filter weka.filters.-unsupervised.attribute.StringToWordVector.
The components use two different weighting schemes: bag of words and TF-IDF.

For each weighting scheme, the package includes two components in order to support a broader set of use cases. The first component accepts one TLarff

Bag of Words - Data Set, Bag of Words - Test/Train Sets, TF-IDF - Data Set, TF-IDF - Test/Train Sets	
Data Set	Test/Train Sets
Inputs • TLarff: Data Set	Inputs • TLarff: Test Set • TLarff: Train Set
Outputs • TLarff: Modified Data Set	Outputs • TLarff: Modified Test Set • TLarff: Modified Train Set
TF-IDF	Bag of Words
Configuration • Int: Min Term Frequency • Int: Max Num Terms	Configuration • Int: Min Term Frequency • Int: Max Num Terms • Boolean: Term Presence

Table 3.6: Preprocessor Component Descriptions.

dataset for which all string attributes, except the Id attribute (see Section 3.3.1), are converted to a vector of keyword weights. The converted dataset is returned as a TLarff. The second component differs in that it accepts and returns two TLarff datasets. One dataset is the training set, which defines the vector of keywords. The other dataset is the test set, which is converted using the same vector of keywords to ensure the attributes in both sets match.

The components have several configuration options. The first two options appear in all four components: minimum term frequency, the number of times a term must appear to qualify to be a feature; and maximum word count, the maximum number of features allowed. The third configuration option is only used by the bag of words components: term presence, which converts term counts to a binary appearance.

Slice Cluster Trees.

There are two components available to convert an input TLDoubleTree to a TLClustersCollection: Slice Cluster Tree into Clusters and Slice Cluster Tree at Cutoff. These components are used in conjunction with the Hierarchy Cluster Tree component. Both components make the conversion by slicing the hierarchy tree at a fixed point and forming clusters from the sub trees that are still in tact, shown in Figure 3.6.

The two components differ only in how they determine the location to slice the hierarchy tree. The first component, Slice Cluster Tree into Clusters, slices based on a fixed number of clusters. The cutoff increases until the configured number of clusters is reached. The Slice Cluster Tree at Cutoff component uses a fixed cutoff. Cutoff values start at 0 at the root, and increase (to 1 if normalized)

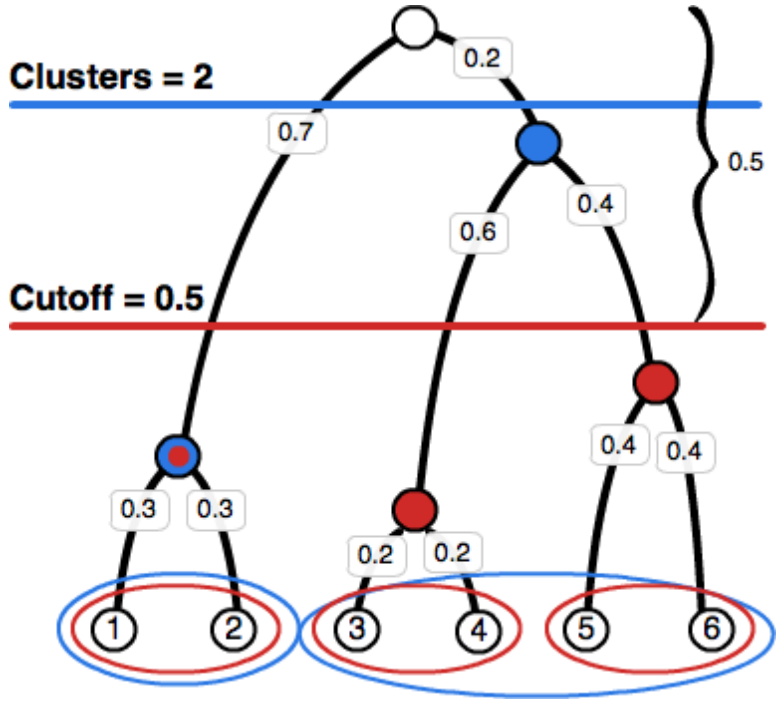

Figure 3.6: Hierarchy tree divided into clusters via fixed cutoff and fixed cluster count.

toward the leaf nodes. any (non-leaf) nodes below the cutoff (closest to the root) are discarded, and the remaining subtrees are the output clusters.

Normalize Tree Branch Lengths.

This component modifies a TLDoubleTree so that the longest branch from root to leaf is of length 1. This is useful when slicing a tree into clusters at a fixed cutoff because the depth of the tree will be a known range.

Merge Clusters of Artifacts.

This component generates as output a TLArtifactsCollection from inputs TLArtifactsCollection and TLClustersCollection. The resulting TLArtifactsCol-

Preprocessors	
Slice Cluster Tree at Cutoff, Slice Cluster Tree into Clusters	
Inputs • TLDoubleTree: Cluster Tree Outputs • TLClustersCollection	Configuration • Double: Cutoff • Integer: Num Clusters
Normalize Tree Branch Lengths	
Inputs • TLDoubleTree: Tree	Outputs • TLDoubleTree: Normalized Tree
Merge Clusters of Artifacts	
Inputs • TLArtifactsCollection • TLClustersCollection	Outputs • TLArtifactsCollection
Get Test Set, Get Train Set	
Inputs • TLarff: Data Set Outputs • TLarff: Test Set | Train Set	Configuration • Integer: Number of Folds • Integer: Current Fold
Randomize Order of Instances	
Inputs • TLarff: Data Set	Outputs • TLarff: Randomized Data Set
Add Prefix to Cluster Names	
Inputs • TLClustersCollection Outputs • TLClustersCollection	Configuration • String: Prefix

Table 3.7: Preprocessor Component Descriptions.

lection contains one artifact for each cluster in the collection. Each artifact shares and ID with cluster it is derived from and is contents are the concatenation of every source artifact in that cluster.

Get Test Set and Get Train Set.

These components takes as input a TLarff dataset and extract a test set and training set, respectively, for a given cross-validation fold. Both components also require two more integer inputs: the total number of cross-validation folds; and the current cross-validation fold.

Randomize Order of Instances.

This component takes as input and output a TLarff and randomly reorders the artifacts inside. Randomization is sometimes needed when running multiple iterations of the same test so that artifacts are not placed in the same test and training sets over and over again.

Add Prefix to Cluster Names.

This component takes as input and output a TLClustersCollection. The component also has a string configuration option, which is prepended to the ID of each cluster in the collection before being returned.

3.4.6 Postprocessors

Postprocessors represent the cleanup and results gathering components that run after tracing, clustering, or classifying is complete. The component package con-

Postprocessors	
Generate Confusion Matrix	
Inputs	Outputs
• TLarff: Labeled Data Set • Integer: Result Attribute Index	• TLConfusionMatrix
Remove Clusters Below Confidence	
Inputs • TLClustersCollection Outputs • TLClustersCollection	Configuration • Boolean: Remove Unknown Confidences • Double: Min Confidence
Remove Artifacts In Class	
Inputs • TLArtifactsCollection • TLarff: Labeled Data Set • Integer: Result Attribute Index Outputs • TLArtifactsCollection	Configuration • Integer: Class Index • Boolean: Invert Selection • Boolean: Remove Missing Artifacts

Table 3.8: Postprocessor Component Descriptions.

tributes three Postprocessors: Generate Confusion Matrix, Remove Clusters Below Confidence, and Remove Artifacts In Class.

Generate Confusion Matrix.

This component takes as input the TLarff and Integer outputs of a classifier and generates a TLConfusionMatrix. The input must have a class attribute with the known classes for each entry. The Integer attribute index input must line up

with the classifier's class predictions.

Remove Clusters Below Confidence.

This component prunes a given TLClustersCollection based on the confidence values of the artifacts in the clusters. Confidence values are optional for artifacts, so there is a configuration option to decide if artifacts without confidence values are kept or pruned. The minimum confidence value is also determined by a configuration option.

Remove Artifacts In Class.

This component prunes a given TLArtifactsCollection based on the results of a classifier. The component also takes as input a TLarff with the classifier results. The ID attribute is used to link the TLarff artifacts to the original TLArtifactsCollection.

3.4.7 Helper Components.

Helper Components implement simple tasks, such as instantiating a variable, incrementing a variable, or accessing a property of an object. Each of these operations assists with the setup of the Workspace in TraceLab. This package includes five helper components, shown in Table 3.9.

The tasks that the five components execute are: Double Writer instantiates a double to the Workspace; Increment Double by Value increments a double in the Worksapce by a value specified in its configuration; Get TLarff Class Index accesses a TLarff object for the index of the class attribute; Get Cluster from Col-

Helper Components	
Double Writer	
Outputs	Configuration
• Double	• Double
Increment Double by Value	
Inputs	Configuration
• Double	• Double: Increment
Outputs	
• Double	
Get TLarff Class Index	
Inputs	Outputs
• TLarff	• Integer: Class Attribute Index
Get Cluster from Collection	
Inputs	Outputs
• TLClustersCollection	• TLCluster
• Integer: Cluster Index	
Count Number of Clusters	
Inputs	Outputs
• TLClustersCollection	• Integer: Num Clusters

Table 3.9: Helper Component Descriptions.

lection retrieves a TLCluster at the requested index of the TLClustersCollection; and Count Number of Clusters returns the number of clusters in the collection.

CHAPTER 4

Clustering and Automated Tracing

The purpose of this section is to ask the question "Can clustering improve automated requirements tracing?" We present a solution where clustering facilitates decomposing a large tracing problem into many smaller tracing subproblems. This is meant to be a high-level description of one possible approach, and as such, many of the implementation details are left to future research, as these decisions have not been empirically determined. The next section contains our specific implementation.

For the remainder of this section, we simplify the process by tracing a single set of source artifacts $S = \{s_1, ..., s_n\}$ to a single set of target artifacts $T = \{t_1, ..., t_m\}$. In reality, the information landscape is far more complex and requires tracing many sets of source artifacts to many sets of target artifacts. We believe this solution will generalize to fit more diverse use cases.

Before diving into our solution, we take a look at how automated tracing is currently performed. Each set of artifacts, S and T, go through a series of preprocessors that clean up and transform the data into representations S' and T' respectively. The goal of preprocessing is to convert the text in an artifact to a more machine-friendly format such as TF-IDF.

After preprocessing comes the actual tracing. A similarity score $sim(S'_i, T'_j)$ is computed for every source artifact to every target artifact. Next, a cutoff

value is selected such that every source and target artifact with a similarity score greater than the cutoff becomes part of the set of candidate links $L = \{(s_i, t_j)|sim(s'_i, t'_j) > cutoff\}$ In an ideal scenario, the similarity scores for every true link would be above the cutoff and the similarity scores for every false link would be below the cutoff.

Conventional tracing suffers from many limitations. Primarily, the need for very high recall comes at the cost of low precision. In order to achieve high recall, tracing processes traditionally rank all of the links, and then set the bar very low to make sure no true links were missed. These methods then rely on human effort to sort out the true links from the false. What these approaches lack is a way to be inclusive enough to achieve high recall, without having to accept all of the false links in between.

In our modified approach, defined in Algorithm 1, we use clustering to break up the task of tracing into smaller tracing subproblems. This approach is divided into three steps: clustering, cluster-to-cluster tracing, and intercluster artifact tracing. By dividing up the task of tracing into subproblems, we can approach each subproblem with a different strategy. This allows for the possibility of being very inclusive in some areas in order to achieve the high recall needed, while being very selective in other areas to cut down on false positives.

The solution is outlined in Figure 4.1. Figure 4.1a. is the entire search space of possible links. Figure 4.1b. shows the results of conventional tracing, while 4.1c. is the results of the second step, cluster-to-cluster tracing. Finally, Figure 4.1d. shows two example subproblems for the final step, intercluster tracing.

Algorithm 1 Trace source artifacts to target artifacts using clustering

procedure ClusteredTracing($Sartifacts, Tartifacts$)

$Sclusters \leftarrow cluster(Sartifacts)$ ▷ Step 1: Clustering.

$Tclusters \leftarrow cluster(Tartifacts)$

▷ Step 2: Cluster-to-cluster tracing.

$SCartifacts \leftarrow aggregate(Sartifacts, Sclusters)$

$TCartifacts \leftarrow aggregate(Tartifacts, Tclusters)$

$Ctraces \leftarrow trace(SCartifacts, TCartifacts)$ ▷ Trace clustered artifacts.

$Atraces \leftarrow \emptyset$ ▷ Initialize Results

for all $Ctrace \in Ctraces$ **do**

if $Ctrace.score > Ccutoff$ **then**

$Scluster \leftarrow Sclusters[Ctrace.Sid]$ ▷ Step 3: Intercluster tracing.

$Tcluster \leftarrow Tclusters[Ctrace.Tid]$

$ICtraces \leftarrow trace(Scluster.artifacts, Tcluster.artifacts)$

for all $ICtrace \in ICtraces$ **do**

▷ $Acutoff$ may be different for every $Ctrace$

if $ICtrace.score > Acutoff$ **then**

$Atraces.add(ICtrace)$ ▷ Collect Results.

end if

end for

end if

end for

return $Atraces$

end procedure

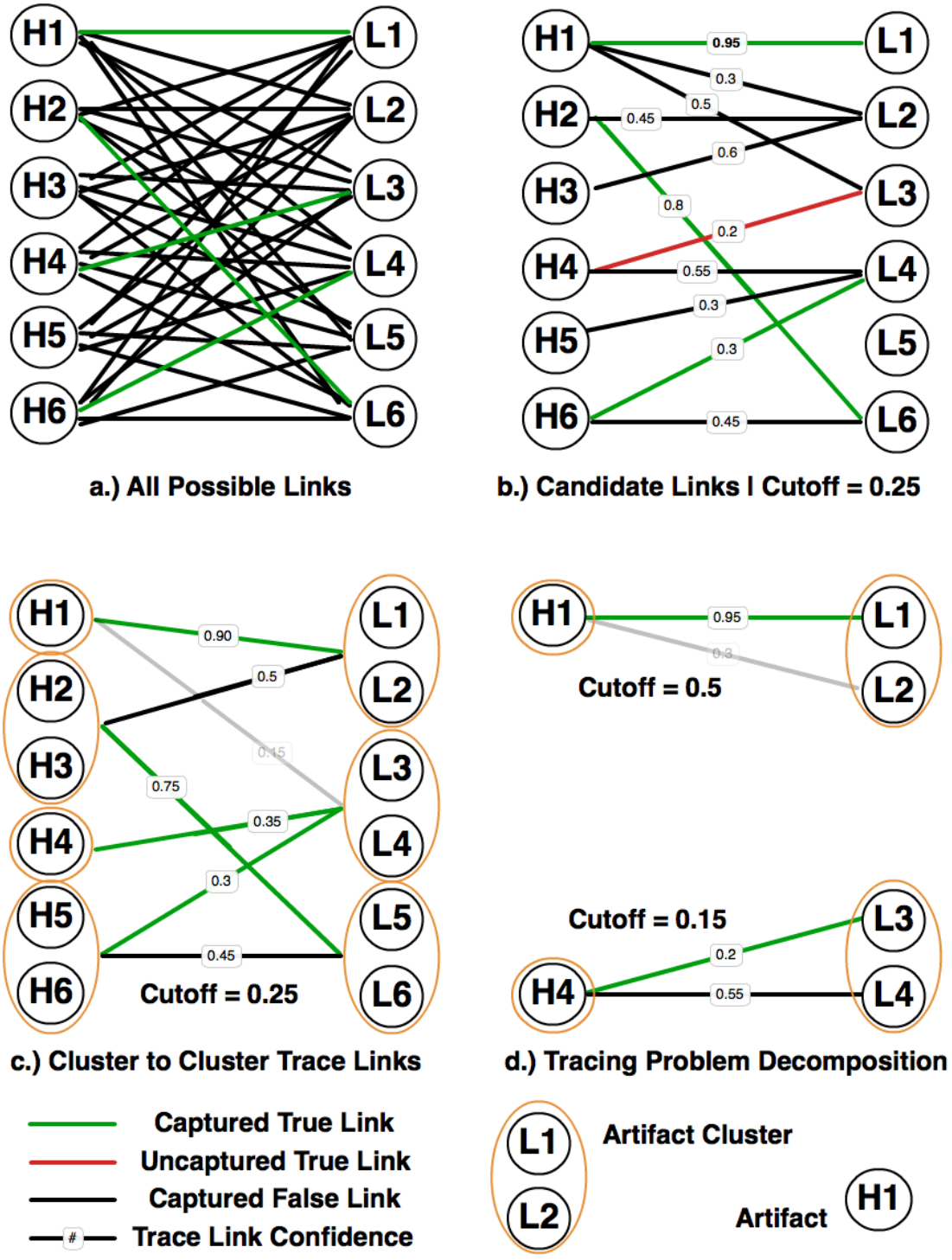

Figure 4.1: Tracing problem decomposition using clustering.

4.1 Clustering

This approach begins by generating sets of clusters. For each set of artifacts S and T, we generate the sets of clusters SC and TC, respectively, where each cluster in the set is a collection of one or more artifacts united by content similarity. The goal of clustering is to divide the data into groups such that each item is more similar to the other items in its group (cluster) than to items in the other groups.

4.2 Cluster-To-Cluster Tracing

Once clusters SC and TC have been established, it is time to begin tracing. This step is very similar to conventional tracing, except instead of tracing source artifact S to target artifacts T, we trace source clusters SC to target clusters TC. Much like the way conventional tracing requires artifacts to be modeled by vectors S' and T', cluster-to-cluster tracing requires clusters to be modeled by vectors SC' and TC'.

The result of cluster-to-cluster tracing is very similar to the output of conventional tracing. A similarity score $sim(sc'_i, tc'_i)$ is computed for every source cluster to every target cluster. Again, a cutoff is determined such that every cluster-to-cluster trace with a similarity score higher than the cutoff becomes a subproblem in the intercluster tracing step, while cluster-to-cluster traces below the cutoff are not considered. To clarify, when a cluster-to-cluster trace falls below the cutoff, none of the possible links from any of the artifacts in the source cluster to any of the artifacts in the target cluster will be included in the final set of candidate trace links.

In an ideal cluster-to-cluster tracing scenario, every cluster-to-cluster trace

above the cutoff value would contain at least one true link, while every cluster-to-cluster trace below the cutoff would only contain false links. We define the following win scenarios that demonstrate how cluster-to-cluster tracing can improve results over conventional tracing methods. It is important to note that these scenarios would likely only affect marginal cases, as a false link with a sufficiently high similarity score will almost always become a false positive while a true link with a sufficiently low similarity score will almost always become a false negative.

The first win scenario involves a false link in cluster of all false links. If the false link by itself, in conventional tracing, has a high enough score to be considered a candidate link, precision drops as a result. However, if by clustering, the other artifacts in the source and target clusters introduce enough separation that the cluster-to-cluster score falls below the cutoff and is therefore not considered a candidate link, precision increases over traditional tracing. This win scenario is illustrated by Figures 4.1b. and 4.1c. The captured $(H1, L3)$ false link in 4.1b. is discarded 4.1c because the score of the $(\{H1\}, \{L3, L4\})$ cluster link is significantly lower.

The second win scenario involves a true link with a marginally low score. In traditional tracing, if this score is below the cutoff, recall drops as a result. However, if by clustering, the other artifacts in the source and target clusters reduce the separation such that the cluster-to-cluster score rises above the cutoff, the true link potentially becomes a candidate link, which improves recall. It is important to note that this win scenario is somewhat dependent on the decisions made during intercluster tracing. While recall improves as a result of the inclusion of a true link, precision may be lowered by the simultaneous inclusion of several other false links within the same cluster-to-cluster trace. Generally, this is acceptable

as recall is favored over precision because humans have a harder time catching errors of omission [21]. This win scenario is illustrated by Figures 4.1b. and 4.1c. The missed $(H4, L3)$ true link in 4.1b. is caught by the $(\{H4\}, \{L3, L4\})$ cluster link in 4.1c.

4.3 Intercluster Tracing

Inside every cluster-to-cluster trace is a subset of the possible trace links between the source and target artifacts. The goal of intercluster tracing is to determine which of those possible links belongs in the set of candidate trace links. Intercluster tracing is the same as conventional tracing, except that it is performed on a single cluster-to-cluster trace at a time. This means that decisions made for links in one cluster-to-cluster trace do not have to affect links in another. Different subproblems can call for different algorithms or cutoffs. This is illustrated in Figure 4.1d.

An example of how this can benefit tracing is when there are two trace links, one true and one false, with the same similarity score in conventional tracing. With conventional tracing, there would be no way to separate the true from the false link. However, with intercluster tracing if the two links belonged to different cluster-to-cluster traces, they could be assessed separately. Ideally, the subproblem with the false link would be assigned a higher cutoff such that the false link was not included, improving precision. Meanwhile, the subproblem with the true link would be assigned a lower cutoff such that the true link became a candidate link, improving recall. This is shown in Figure 4.1d. where the false link $(H1, l2)$ is excluded, even though it has a higher similarity score than the true link $(H3, L3)$ which is included.

We recognize that for each of the win scenarios above, there is a contrasting loss scenario. We consider loss scenarios to be any false link that is included only if it would have been excluded by conventional methods. Similarly, any true link that is excluded that would have been caught by traditional methods is also a loss scenario in this approach.

Given the vast number of parameters than can be modified in this solution, we believe this opens up many new avenues of research that are worth pursuing. Some of them, undoubtedly, will result in more losses than wins and will not be effective over conventional tracing methods. However, many others have the potential to outperform conventional tracers by being selective with higher scoring false links, and inclusive with lower scoring true links. The focus here is to improve performance on the marginal cases, where traditional methods have trouble distinguishing true links from false links.

CHAPTER 5

Experimental Design

This section is an overview of the implementation and evaluation of the cluster based tracing system described in Chapter 4. The system is implemented as a collection of TraceLab experiments. Our implementation attempts to answer the question "Can clustering improve automated requirements tracing?" Our goal is to observe the effects that clustering has on the tracing process. We define an improvement over conventional tracing as any increase in precision for equal recall.

5.1 Datasets

Our implementation is evaluated using two datasets, CM-1 [30] and WV_CCHIT [39]. Both datasets are available through the CoEST website [11] and are compatible with TraceLab's built-in data types and importers.

CM-1.

The CM-1 dataset contains two sets of artifacts: a set of high-level requirements and a set of low-level design elements. The artifacts originated from the design of a NASA scientific instrument. The CM-1 dataset has been modified by NASA prior to public release in order to hide the identity of the instrument. The dataset consists of 235 high-level requirements and 220 low-level design elements.

We rely on a ground truth traceability matrix that was created and validated by prior work [21], which contains 361 true links.

WV_CCHIT.

The WV_CCHIT dataset is a set of requirements from WorldVista (WV), an electronic healthcare information system developed by the USA Veterans Administration. The requirements are traced to a set of regulatory requirements developed by the Certification Commission for Healthcare Information Technology (CCHIT). The dataset contains 116 WorldVista requirements and 1,064 CCHIT regulatory requirements. We rely on a ground truth traceability matrix that was created and validated by prior work [39], which contains 587 true links.

5.2 Setup

The tracing system in this work consists of three steps: clustering, cluster-to-cluster tracing, and intercluster tracing.

First, we cluster both the source and target artifacts. This includes choosing which clustering algorithm to use and setting any algorithm-specific properties, such as the number of clusters to make. We do not use any formal measures to evaluate the first step in isolation because the clusters by themselves do not say much with respect to traceability. We opt instead for a visual inspection to make sure the clusterings are at least realistic. For example, a clustering algorithm that clusters 100 items into 20 clusters where 19 clusters have only a single item and the 20th cluster has the remaining 81 items would raise a red flag and should not be used.

Second, we perform cluster-to-cluster tracing. Cluster-to-cluster tracing is, at its core, a tracing process, which means for each cluster we need a single traceable artifact representation to trace. Like most tracing processes, the parameters are an algorithm and a minimum similarity cutoff. Each cluster-to-cluster link that is above the cutoff is is evaluated using intercluster tracing. Intercluster tracing is also a tracing process and its parameters are an algorithm and a minimum similarity cutoff. The results from intercluster tracing for each cluster-to-cluster trace link are aggregated into a single candidate RTM that is evaluated for precision, recall, f_2, pf, and selectivity (see section 2.4).

5.3 Implementation

Our implementation is done entirely through TraceLab experiments. Most of the components used are from the machine learning component package described in Chapter 3 or the standard TraceLab components. A few of the components used have been developed specifically for this experiment. At this time, these components will only be released as part of this experiment because their scope is very narrow.

Clustering.

The first step, clustering, is performed in its own experiment, shown in Figure 5.1. The clustering experiment imports a single TLArtifactsCollection of the artifacts that are to be clustered. The experiment performs several preprocessing steps to clean up the input by removing non-characters such as punctuation, then removing all stop words, and finally stemming the input using the Porter stemming algorithm. All of components that perform these preprocessing steps

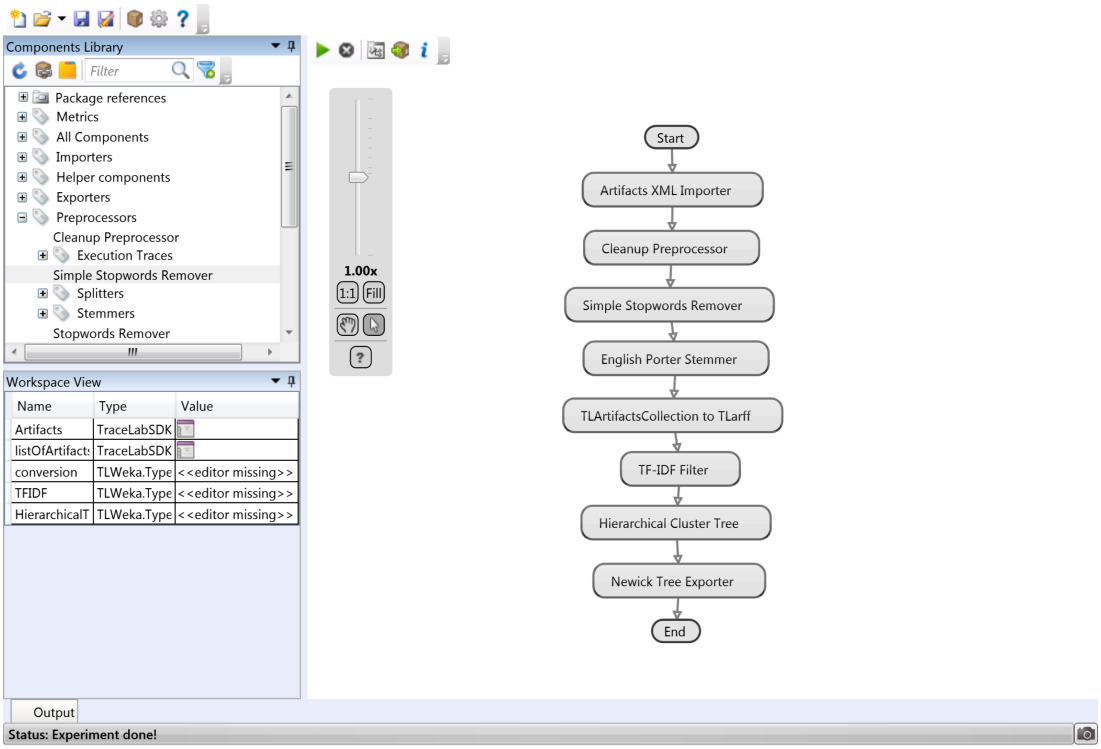

Figure 5.1: TraceLab experiment for creating a cluster tree from a set of artifacts.

are part of the standard TraceLab component library. Once the data has been preprocessed, it must be converted from type TLArtifactsCollection to type TLarff to be compatible with the WEKA based clusterers. We use the component provided in the machine learning component package to do the conversion. The final step before clustering is to transform the string to a vector of features. For this we chose TF-IDF, as it is the most popular representation.

Once the input data has been preprocessed and converted, it can be clustered. Because we do not know what size clusters will perform best, we chose the Hierarchical Cluster Tree component. The hierarchical clustering tree component is different from all the other clusterers in the component package because it returns a TLDoubleTree which can be sliced into any number of clusters (see Figure 3.6), instead of a TLClustersCollection which has a fixed number of clusters.

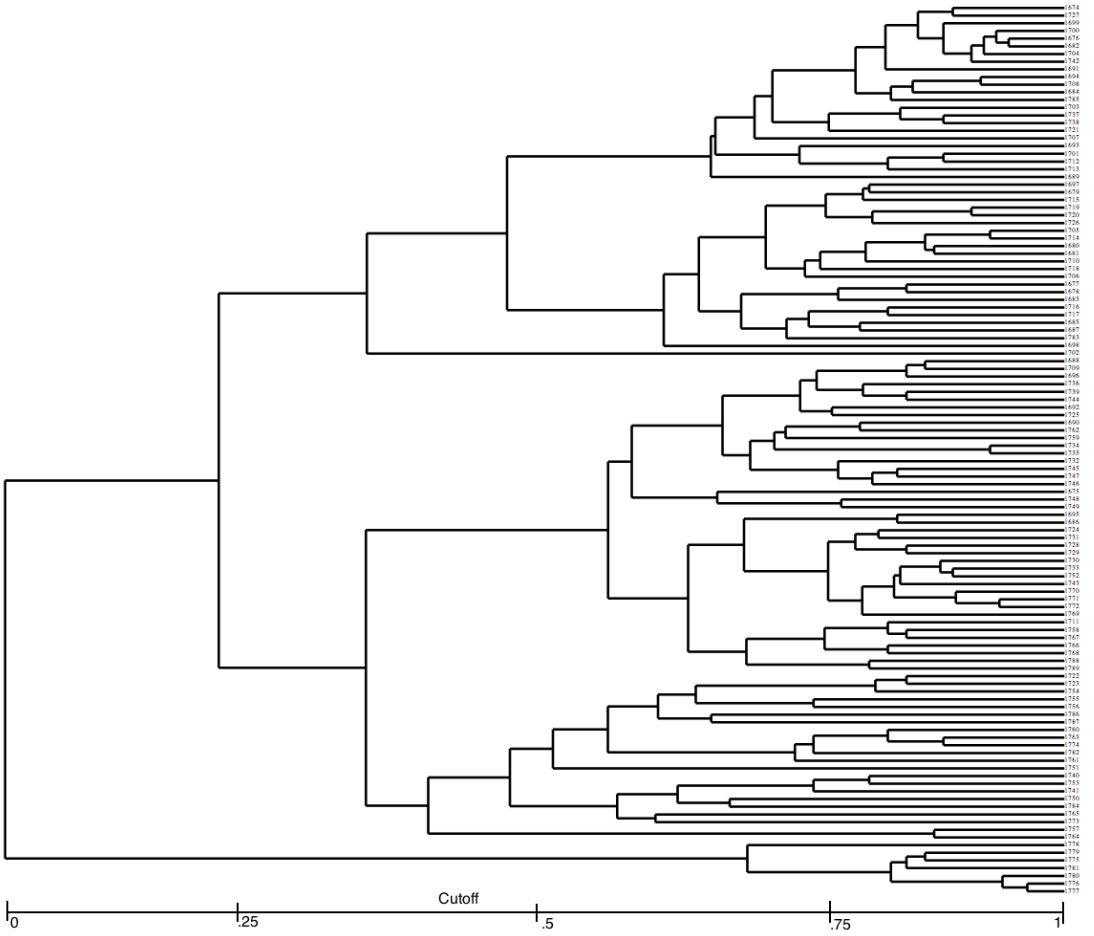

Figure 5.2: Example Hierarchical Cluster Tree of WV_CCHIT Source Artifacts.

An example of the cluster hierarchy tree can be seen in Figure 5.2. The root of the tree is visible on the left edge of the figure and the leaf nodes which correspond to the clustered artifacts are on the right. At a cutoff of 0, the root node is still intact so all artifacts in the tree will be in a single cluster. At a cutoff of 1, only the leaf nodes remain so each artifact will be in its own cluster.

The Hierarchical Cluster Tree component has two configuration options: link type and distance function. Through visual inspection, we determined that the Manhattan distance function paired with the Ward link type produced the most evenly distributed clusterings. Many of the other combinations produced skewed

results with one very large cluster and many singletons. Others produced a poor distribution of cutoff values that would make testing a variety of cluster sizes infeasible.

The result of the first experiment is a TLDoubleTree cluster tree that can be sliced at any cutoff to produce a TLClustersCollection set of clusters. This is performed for both source and target artifacts and the resulting cluster trees are stored to be used in the next step.

Clustered tracing setup.

The next experiment executes an evaluation of the clustered tracing process, shown in Figure 5.3. First, we must import all of the data needed for this experiment. This includes: the source and target TLArtifactsCollection, the source and target TLDoubleTree, and the answer TLSimilarityMatrix. Next, setup involves preprocessing the input, much like in the previous experiment. The artifacts are cleaned up by removing non-characters, removing stopwords, and Porter stemming. The input cluster trees are also preprocessed by normalizing the branch lengths so the cutoff range is always a predictable $[0, 1]$. The final setup step is to initialize the results container and loop variables.

To gain insight into optimal cluster sizes, we generate a TLClustersCollection for every possible cluster cutoff for both source and target artifacts. We then perform clustered tracing for every combination of source and target clusterings.

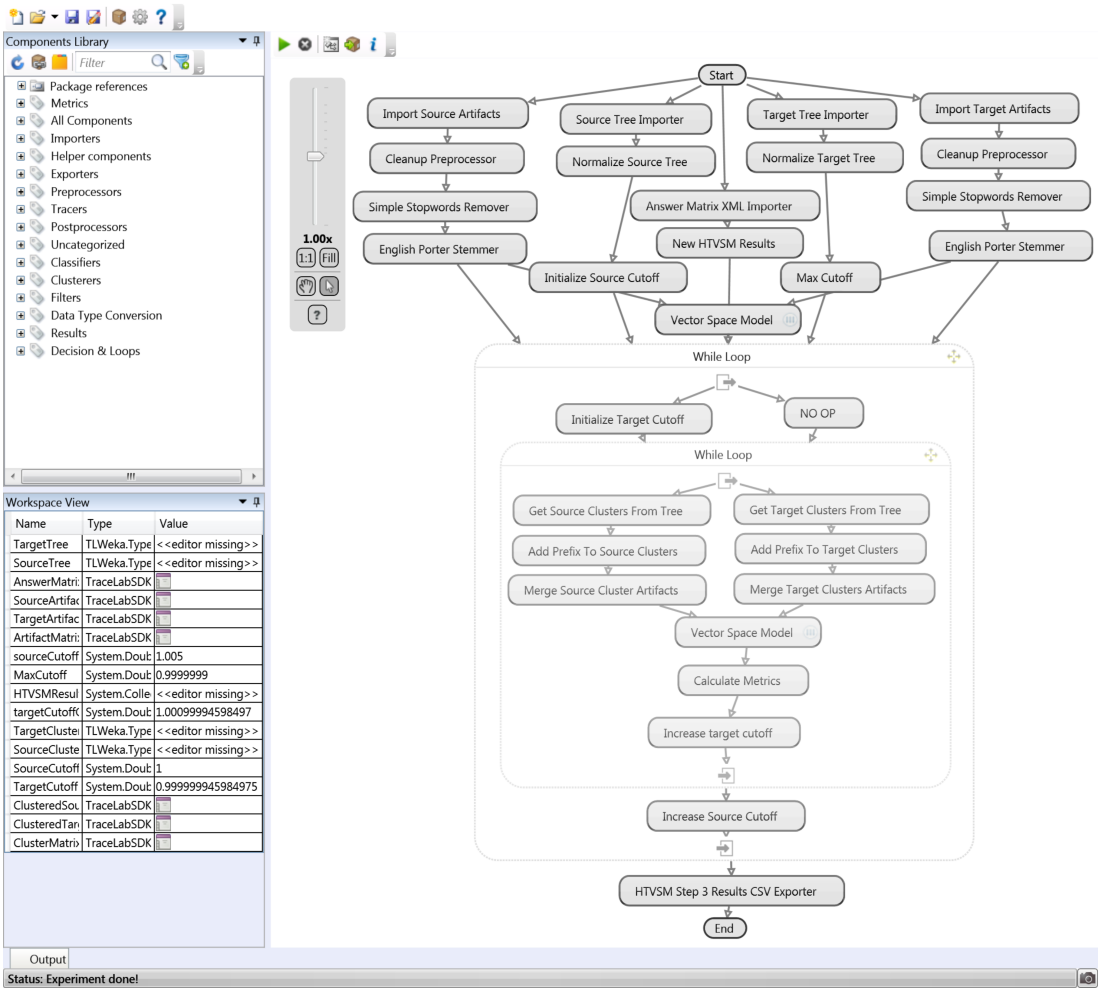

Figure 5.3: TraceLab experiment for evaluating clustered tracing.

Cluster-to-cluster tracing.

Given a set of source and target clusters and their artifacts, we perform cluster-to-cluster tracing. First, we generate a new set of source and target artifacts by aggregating the contents of each cluster into a single, traceable artifact. In this implementation, we aggregate the source and target artifacts by concatenating the preprocessed input strings. We then trace the new sets of source and target artifacts using VSM with TF-IDF weighting scheme.

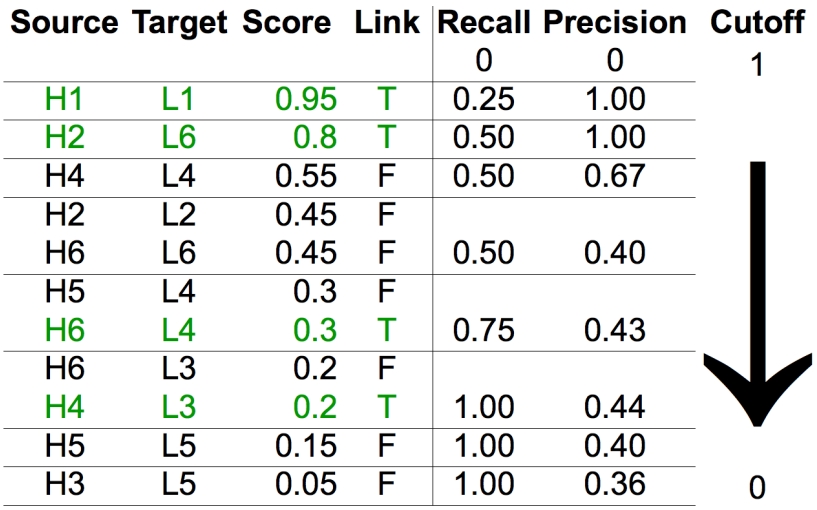

Figure 5.4: TraceLab experiment for evaluating clustered tracing.

Each iteration of cluster-to-cluster tracing produces a TLSimilarityMatrix candidate cluster RTM. A cluster RTM is only different from an artifact RTM in that the links are references from cluster to cluster instead of from artifact to artifact. For every link above a given score cutoff, we then perform intercluster tracing. For evaluations, the cutoff is determined by selecting a recall goal and reducing the cutoff until the target is reached. The cutoff selection is shown in Figure 5.4 for a sample of artifact to artifact links, but the same concept applies to cluster-to-cluster links.

It is important to note that if both the input source and target clusters are all singletons, cluster-to-cluster tracing regresses into conventional tracing. This is helpful in determining a baseline comparison for our methods. Similarly, if both the input source and target clusters are a single cluster that contains all the source or target artifacts, respectively, then the final step, intercluster tracing, regresses into conventional tracing.

When a cluster link falls above the cutoff, every interior artifact link, whether true or false, would be included in the candidate RTM. This generally results in

extremely low precision because every true link comes bundled with a handful of false links. For this reason, we do not evaluate cluster-to-cluster tracing on its own. Instead, we perform intercluster tracing on every cluster-to-cluster link to further refine the candidate RTM before evaluating

Intercluster tracing.

Given a source and target cluster for which a link has been established by cluster-to-cluster tracing, intercluster tracing determines which artifacts in the source cluster have links to artifacts in the target cluster. Our implementation uses VSM with TF-IDF to calculate similarity scores for the artifacts.

This approach attempts to validate one of the proposed win scenarios in Chapter 4. Namely, when a false link, which under conventional tracing has a sufficiently high score to be a positive link, is discarded as part of a low scoring cluster link. In order to observe the effects of this particular win scenario, we use the same algorithm and similarity cutoff when performing intercluster tracing for every cluster link in a given cluster-to-cluster tracing iteration.

As with cluster-to-cluster tracing, the intercluster similarity cutoff is defined by a recall goal. The similarity cutoff is reduced until enough links are included to reach the recall target, as shown in Figure 5.4. Because this step is based on the results of cluster-to-cluster tracing, the recall target for intercluster tracing must always be lower than the target for cluster-to-cluster tracing. In other words, once a candidate link has been discarded by cluster-to-cluster tracing, there is no way to get it back during intercluster tracing.

In the end, we assemble the candidate links from each iteration of intercluster tracing into a candidate RTM that is evaluated for precision, recall, f2, pf, and

Datasets:	CM1, WV_CCHIT	Cluster Recall:	80%, 85%, 90%, 95%, 97%, 100%
Source Cutoff:	[0.0, 1.0]	Overall Recall:	80%
Target Cutoff:	[0.0, 1.0]	Output:	Precision

Table 5.1: Results Evaluation

selectivity. This is done for every combination of source and target clusterings. We evaluate the results below.

Summary.

There are four (4) distinct cutoffs that must be set to get a candidate RTM. The first two are the cluster cutoffs that slice the source and target cluster hierarchies into clusterings. This experiment evaluates every possible source clustering against every possible target clustering.

The next two cutoffs are tracing similarity cutoffs for cluster-to-cluster tracing and intercluster tracing. For a given source and target clustering, we perform cluster-to-cluster tracing and select the first similarity cutoff so that the set of cluster links achieves a recall goal. Then, for every cluster link that is above the first similarity cutoff, we perform intercluster tracing and collect the candidate links. Again, we select a recall goal, which must always be less than or equal to the cluster-to-cluster target. This recall goal determines the second similarity cutoff, for which every artifact link in intercluster tracing that exceeds the cutoff is included in the candidate RTM. The candidate RTMs are evaluated below.

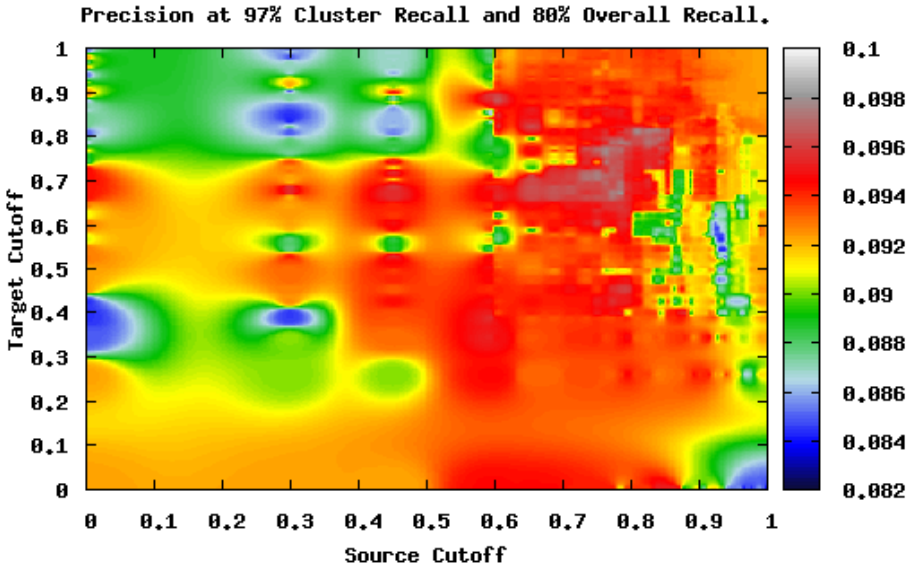

Figure 5.5: Clustered Tracing on CM1 dataset - Precision at 97% Cluster Recall and 80% Overall Recall.

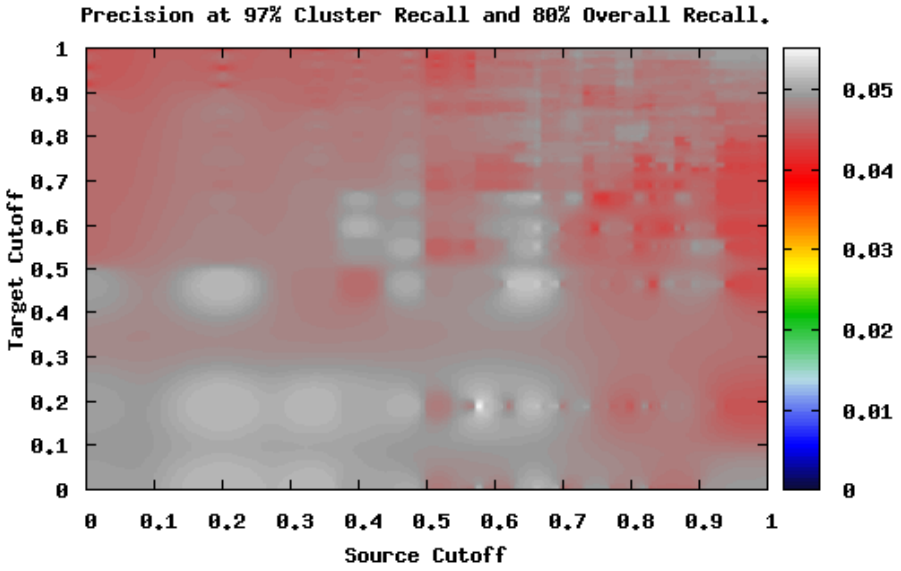

Figure 5.6: Clustered Tracing on CCHIT dataset - Precision at 97% Cluster Recall and 80% Overall Recall.

5.4 Results

To evaluate the results, we have selected a fixed recall goal of 80%, based on the recommendations of Hayes et al. [21] for excellent recall (see Table 2.1). We then evaluate the candidate RTMs using the metrics precision, recall, f_2, pf, and selectivity. However, because recall is held to a fixed constant, f_2 is proportional to precision and both pf and selectivity are inversely proportional to precision. For this reason, precision becomes our primary metric.

To clarify, the candidate RTM has a recall goal of 80%, meaning the results of intercluster tracing must identify 80% of the true links in the ground truth RTM. This means that the results of cluster-to-cluster tracing must identify at least 80% of the true links. Therefore, for cluster-to-cluster tracing, we compare recall goal values of 80%, 85%, 90%, 95%, 97%, and 100%. This is shown in table 5.1.

In order to view precision for all combinations of source and target clusterings, we use a heat map for specific overall and cluster recall goals. This is shown in Figure 5.6 and Figure 5.5. To simplify the visualization of the results, we primarily focus on the cases where the source cluster cutoff is equal to the target cluster cutoff. This is not necessarily the where optimal combination lies, but it is useful in reducing the dimensionality of our graphs. This allows us to compare the various cluster recall targets on a single graph, shown in Figure 5.8 and Figure 5.7.

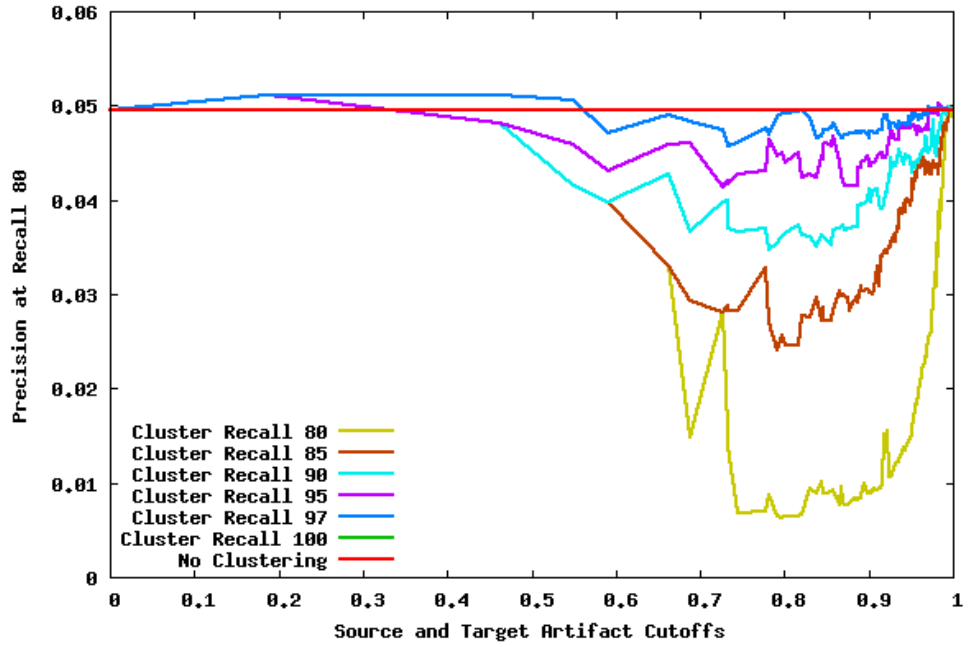

Figure 5.7: Clustered tracing on WV_CCHIT dataset - Precision at 80% Recall with various cluster recalls.

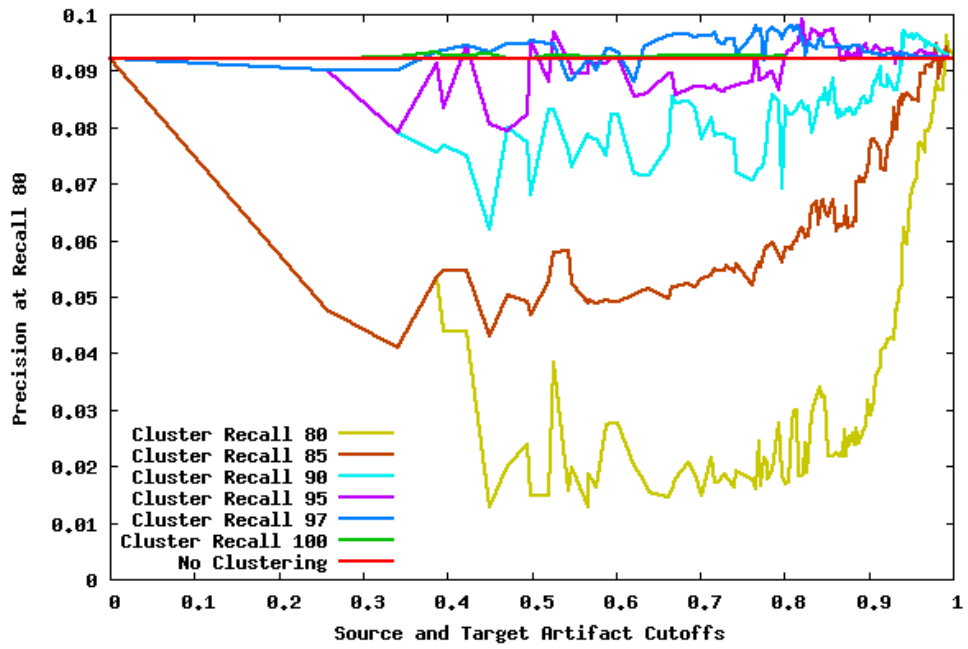

Figure 5.8: Clustered tracing on CM1 dataset - Precision at 80% Recall with various cluster recalls.

5.5 Analysis

Through this approach, we are able to achieve a modest improvement in precision. For CM1, our baseline precision is 9.23%. With our approach, precision peaked at 9.92%, an increase of .69% for 80% recall with 95% cluster recall. This is visible in Figure 5.8 at a source and target cutoff of 0.82. Interestingly, even though the peak performance for CM1 occurred with 95% cluster recall, when cluster recall was bumped to 97%, there was almost never a drop in precision below the baseline.

For the WV_CCHIT dataset, even though peak precision did exceed the baseline, clustering almost always hurt performance. At every cluster recall except 100%, the performance peaked at 5.11%, an increase of .15% over the baseline of 4.96%. The peak performance occurred with a very low cluster cutoff of 0.19. The reason for this is that there is a very small cluster attached to the root node with almost all false links. At very low cutoffs, this cluster is cleaved off, and the remaining artifacts are placed in a single cluster. While this improves precision at first, the effect is very minor. The results show this strategy is not effective for the WV_CCHIT dataset.

In both cases, as the cluster recall approaches overall recall, precision plummets. This behavior can be explained by the fact that it is very difficult to find every single link. There are some links that automated tracing algorithms simply cannot find. By setting the recall goal for cluster-to-cluster tracing too low, there is no ability for intercluster tracing to cut its losses when there are only a few true links left in the sea of false links. This is especially true when both cluster recall and overall recall are set to the same value, in this case 80%, meaning intercluster tracing must find every single true link not discarded by cluster-to-cluster

tracing.

In general, we are not interested in cases where precision is lower than our baseline. In order to focus on only the clusterings where performance is improved, we take a differential heat map of each combination of cluster and overall recall goals. The differential heat maps show positive differentials only. These are shown in Figure 5.9 and Figure 5.10.

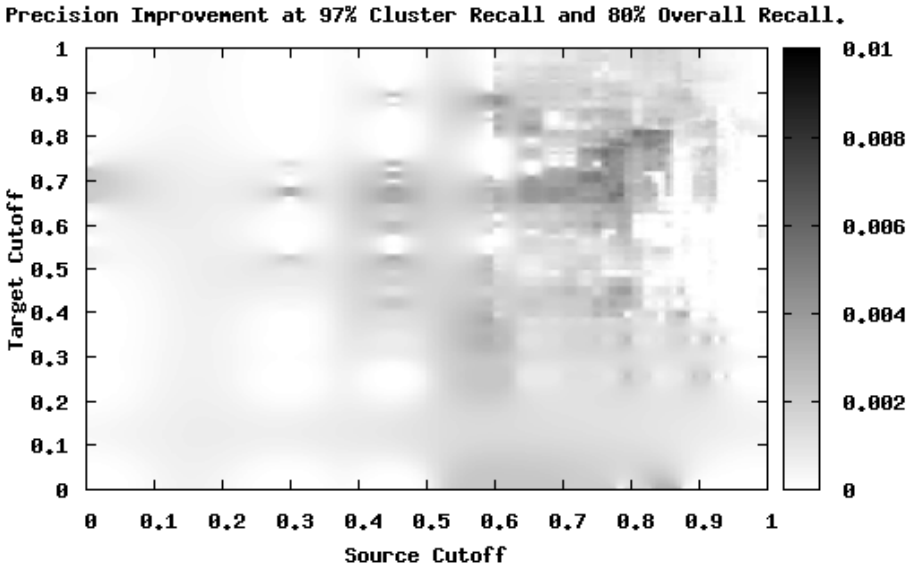

Figure 5.9: Clustered Tracing on CM1 dataset - Precision Improvement at 97% Cluster Recall and 80% Overall Recall.

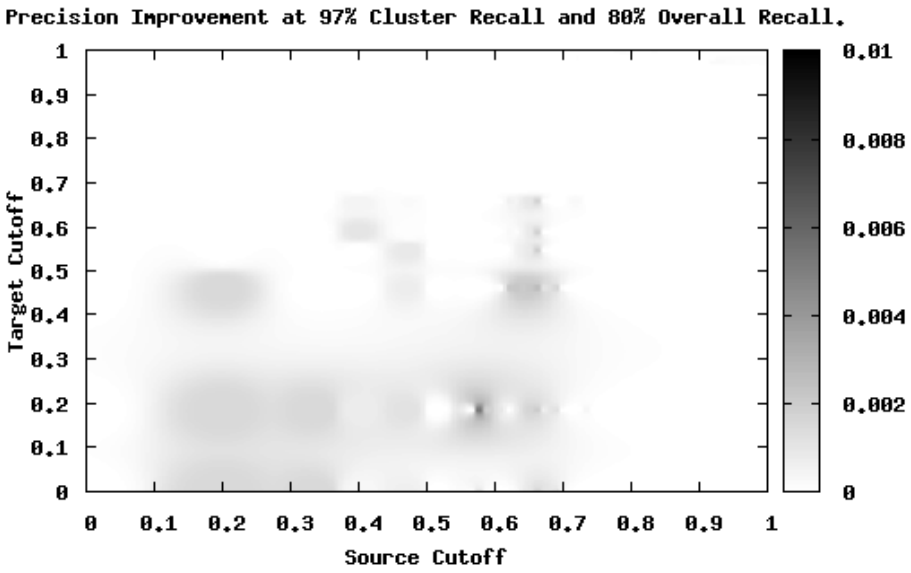

Figure 5.10: Clustered Tracing on CCHIT dataset - Precision Improvement at 97% Cluster Recall and 80% Overall Recall.

CHAPTER 6

Conclusion

The purpose of this work has been to ask the question "Can clustering improve automated requirements tracing?" To that end, we have contributed the design for an automated tracing system that uses clustering to decompose the task of tracing into many smaller tracing subproblems. We also contribute an implementation of several key components of this tracing system using TraceLab, a visual experimental workbench for conducting traceability experiments. Our final contribution is a machine learning component package for TraceLab featuring six (6) classifier algorithms, five (5) clustering algorithms, and a total of over 40 components for creating TraceLab experiments. This component package is to be made available through the CoEST website, so that anyone interested in adding clustering or classification to their TraceLab experiment can do so.

Our results show a modest improvement over the baseline, but we believe this system can do much better. The tracing solution presented in Chapter 4 has so much potential beyond what we were able to implement, particularly in the final step of intercluster tracing, that we believe more advanced approaches will yield better results. We also consider this work a success if other researchers are able to improve their results by using the clusterers and classifiers found in our machine learning component package in their own TraceLab experiments.

Given the exploratory nature of this work, there is a vast amount of future

work to complete. First and foremost is the release of the machine learning component package described in this work to the public. Although the components are functionally complete, there are still a few tasks that remain before the package can be published to the CoEST website. This is expected to be complete in early 2014. Once the component package has been released to the public, we hope that it will inspire a breadth of future work in the area of applying machine learning algorithms to requirements tracing.

The tracing solution outlined in Chapter 4 also provides many opportunities for future work in each of its three steps: clustering, cluster-to-cluster tracing, and intercluster tracing. This work focused on a small subset of clustering algorithms and tracing processes and as such, the first two steps, clustering and cluster-to-cluster tracing, would benefit greatly through future work that applies a wider variety of methods. The final step, intercluster tracing, would greatly benefit from a more advanced solution that treats each subproblem individually. This work is intended not to be a complete solution, but a challenge to the research community, as a complete solution is well beyond the scope and time constraints of this project.

This solution would also benefit from a greater variety of datasets. This is still an open problem for all of software engineering research and we hope that the work CoEST is doing with the Tracy Project (see Section 2.2) will result in the high quality industrial sized datasets needed to truly validate this and all future work.

We also recognize the importance of simulating real world scenarios in order to validate our methods. Although this solution is still in its early stages of development, we acknowledge the recommendations of Borg et al. [6] that experiments in "the cave of IR evaluation", of which this work certainly qualifies, should be

shifted towards case studies with a real world context.

Finally, we would like to highlight the CoEST "Grand Challenges" research agenda toward ubiquitous traceability [15]. We believe this agenda should be the guiding light of all future traceability research and that our contributions have been in line with the goals set forth by CoEST.

BIBLIOGRAPHY

[1] A. Abran and J. W. Moore. Guide to the software engineering body of knowledge. Technical report, IEEE, 2004.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

[3] J. Archie, W. H. Day, W. Maddison, C. Meacham, F. J. Rohlf, D. Swofford, and J. Felsenstein. Newick format. `http://evolution.genetics.washington.edu/phylip/newicktree.html`.

[4] M. F. Bashir and M. A. Qadir. Traceability techniques: A critical study. IEEE, 2006.

[5] B. W. Boehm. Guidelines for verifying and validating software requirements and design specifications. Technical report, TRW, 1979.

[6] M. Borg, P. Runeson, and A. Ardo. Recovering from a decade: A systematic mapping of information retrieval approaches to software traceability. Technical report, Springer Science, 2013.

[7] L. L. Chu. Research on chinese text categorization method oriented to imbalanced corpus., May 2007.

[8] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, O. Gotel, J. Huffman-Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman,

G. Antoniol, and B. B. AND. Grand challenges, benchmarks, and tracelab: Developing infrastructure for the software traceability research community. International Conference on Software Engineering, 2011.

[9] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. pages 155–164, 2010.

[10] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 362–371, 2005.

[11] Coest website. `http://www.CoEST.org`.

[12] D. Cuddeback. Automated requirements traceability: The study of human analysts. Master's thesis, California Polytechnic State University, 2010.

[13] A. Dekhtyar and M. Hilton. Human recoverability index: A tracelab experiment. TEFSE, 2013.

[14] C. Duan and J. Cleland-Huang. Clustering support for automated tracing. Automated Software Engineering Conference, 2007.

[15] O. Gotel, J. Cleland-Huang, J. Huffman-Hayes, A. Zisman, A. Egyed, P. Grunbacher, A. Dekhtyar, G. Antoniol, and J. Maletic. The grand challenge of traceability. Technical report, Center of Excellence for Software Traceability, 2011.

[16] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requiremens traceability problem. IEEE, 1994.

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. volume 11, 2009.

[18] J. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 138–147, 2003.

[19] J. H. Hayes. Risk reduction through requirements tracing. 1990.

[20] J. H. Hayes, A. Dekhtyar, S. Sundaram, A. Holbrook, S. Vadlamudi, and A. April. Requirements tracing on target (retro): Improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering: A NASA Journal*, 3(3):193–202, 2007.

[21] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[22] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, and S. Howard. Helpng analysts trace requirements: An objective look. *IEEE International Requirements Engineering Conference*, pages 249–259, 2004.

[23] S. Hochbaum. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[24] Ikvm website. `http://www.ikvm.net/`.

[25] A. Kannenberg and D. H. Saiedian. Why software requirements traceability remanins a challenge. The Journal of Defense Software Engineering, 2009.

[26] D. Lempia and S. Miller. Requirements engineering management. 2006.

[27] J. Lin, C. C. Lin, J. Cleland-Huang, R. Settimi, J. Amaya, G. Bedford, B. Berenbach, O. Khadra, C. Duan, and X. Zou. Poirot: A distributed tool supporting enterprise-wide automated traceability. pages 363–364, 2006.

[28] C. D. Manning, P. Raghavan, and H. Schutze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[29] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135, 2003.

[30] Mdp website, cm-1 project. `http://mdp.ivv.nasa.gov/mdp_glossary.html#CM1`.

[31] M.Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. pages 639–649, 2012.

[32] Modis science data processing software requirements specification version 2, 1997. SDST-089, GSFC SBRS.

[33] T. Mundie and F. Hallsworth. Requirements analysis using supertrace pc. 1995.

[34] C. on National Security Systems. National information assurance glossary. Technical report, Committee on National Security Systems, 2010.

[35] B. Ramesh. Factors influencing requirements traceability practice. *Commun. ACM*, 41(12):37–44, Dec. 1998.

[36] S. E. Robertson. The probability ranking principle in ir. *Journal of Documentation*, 33(4):294–304, 1977.

[37] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513 – 523, 1988.

[38] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[39] Y. Shin and J. Cleland-Huang. A comparative evaluation of two user feedback techniques for requirements trace retrieval. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1069–1074, New York, NY, USA, 2012. ACM.

[40] Weka website. `http://www.cs.waikato.ac.nz/ml/weka/`.

[41] M. Wieloch, S. Amornborvornwong, and J. Cleland-Huang. Trace-by-classification: A machine learning approach to generate trace links for frequently occurring software artifacts. TEFSE, 2013.

[42] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, second edition edition, 2005.

[43] X. Wu, V. Kumar, J. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinberg. Top 10 algorithms in data mining. Technical report, Springer, 2007.

APPENDIX A

Results Graphs

The following graphs display precision for a given source and target cluster cutoffs at a specific cluster recall and overall recall. There are 4 groups of 6 graphs. The first 2 groups display precision at various cluster recalls as heat maps for both CM1 and WV_CCHIT datasets. The second two groups of graphs display the differential precision over the baseline for each dataset as a heat map. Only positive differential is shown.

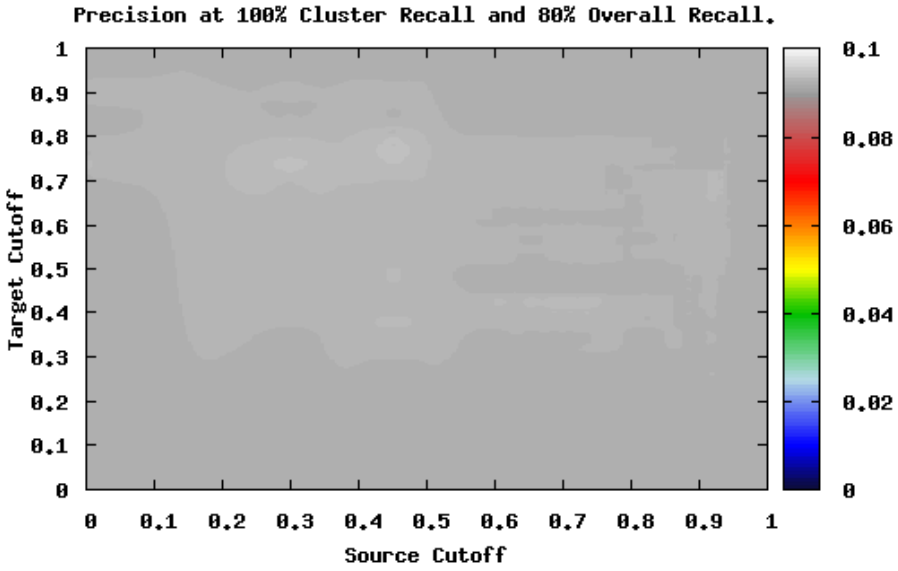

Figure A.1: Clustered Tracing on CM1 dataset - Precision at 100% Cluster Recall and 80% Overall Recall.

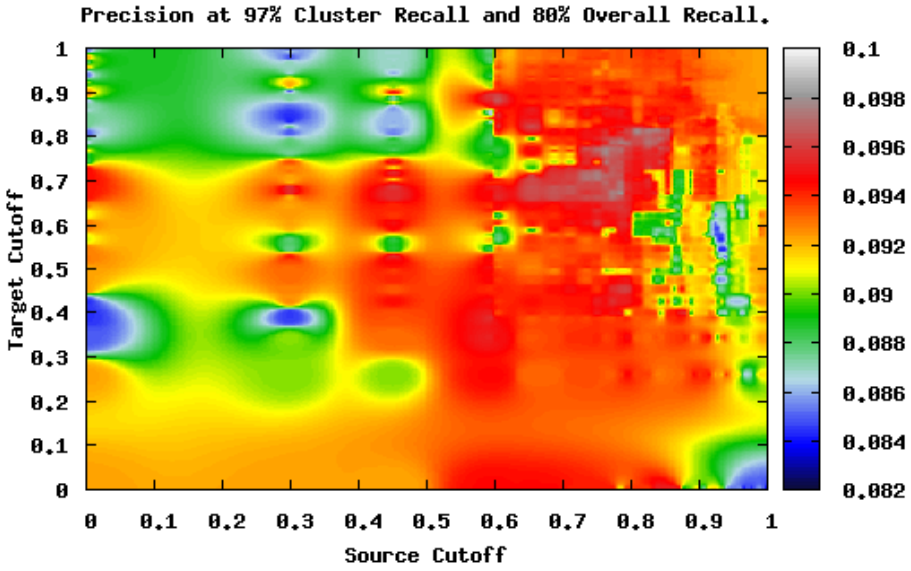

Figure A.2: Clustered Tracing on CM1 dataset - Precision at 97% Cluster Recall and 80% Overall Recall.

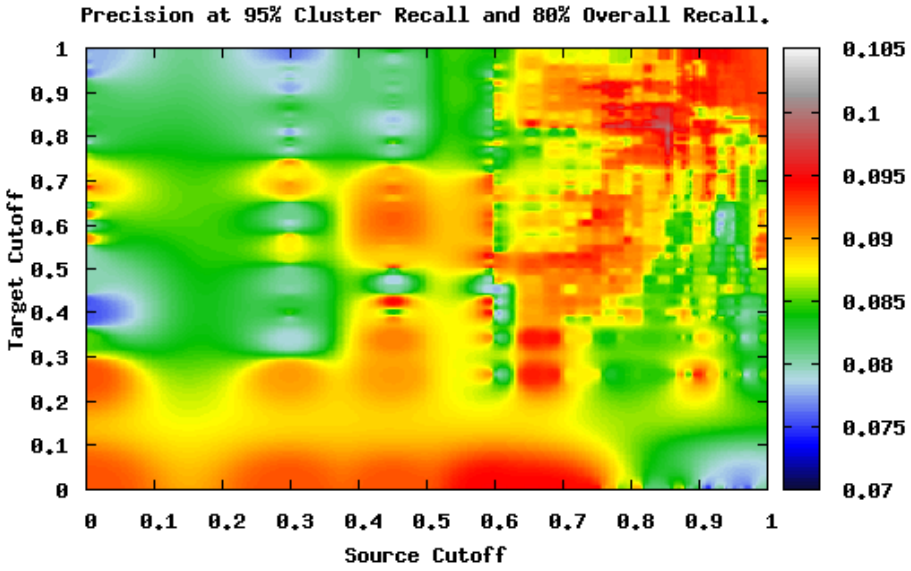

Figure A.3: Clustered Tracing on CM1 dataset - Precision at 95% Cluster Recall and 80% Overall Recall.

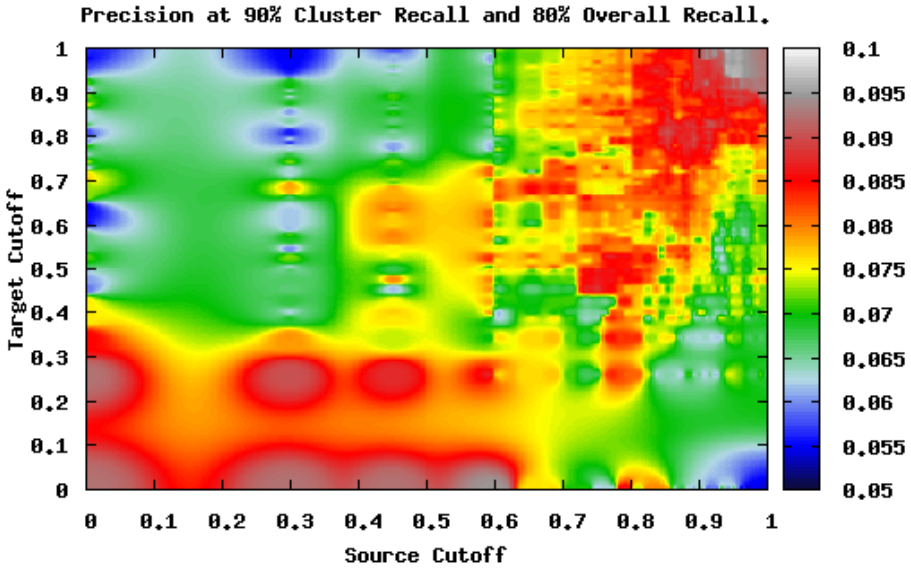

Figure A.4: Clustered Tracing on CM1 dataset - Precision at 90% Cluster Recall and 80% Overall Recall.

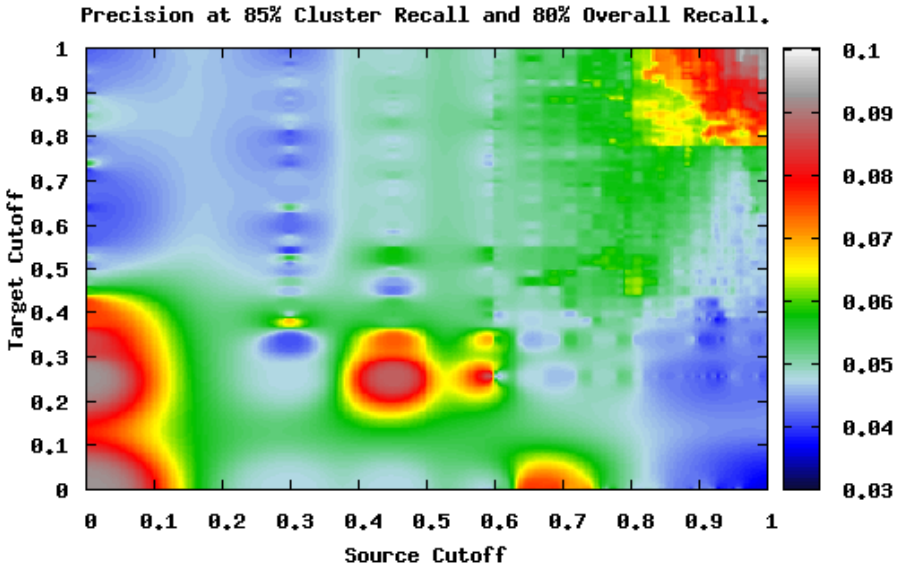

Figure A.5: Clustered Tracing on CM1 dataset - Precision at 85% Cluster Recall and 80% Overall Recall.

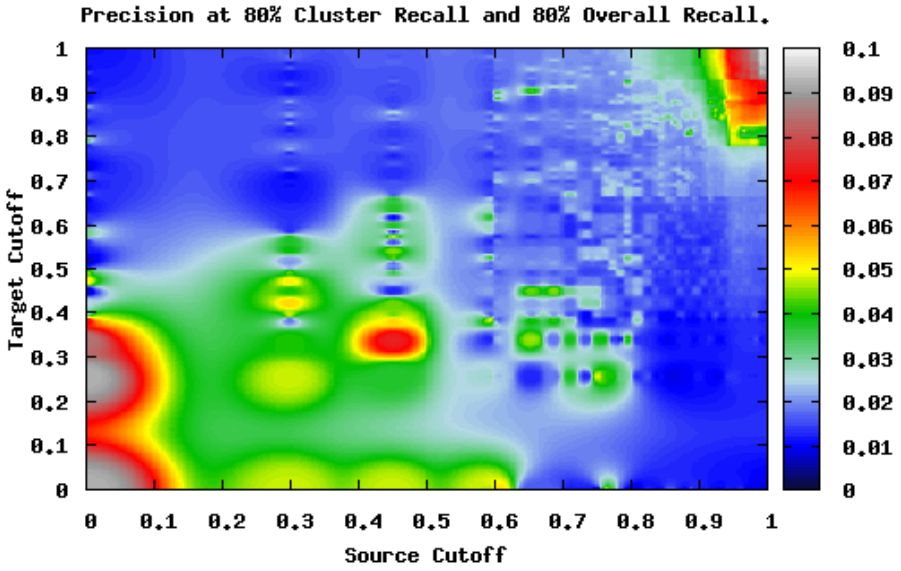

Figure A.6: Clustered Tracing on CM1 dataset - Precision at 80% Cluster Recall and 80% Overall Recall.

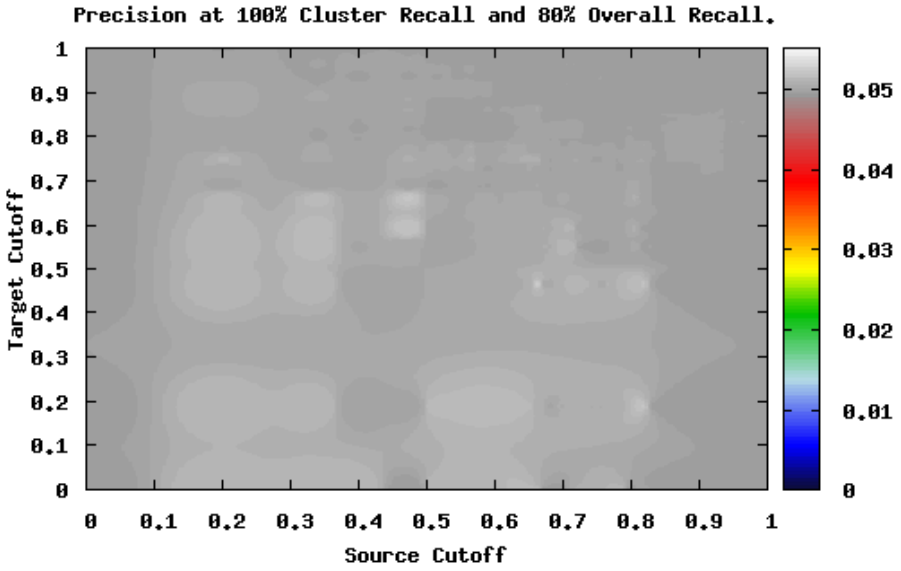

Figure A.7: Clustered Tracing on CCHIT dataset - Precision at 100% Cluster Recall and 80% Overall Recall.

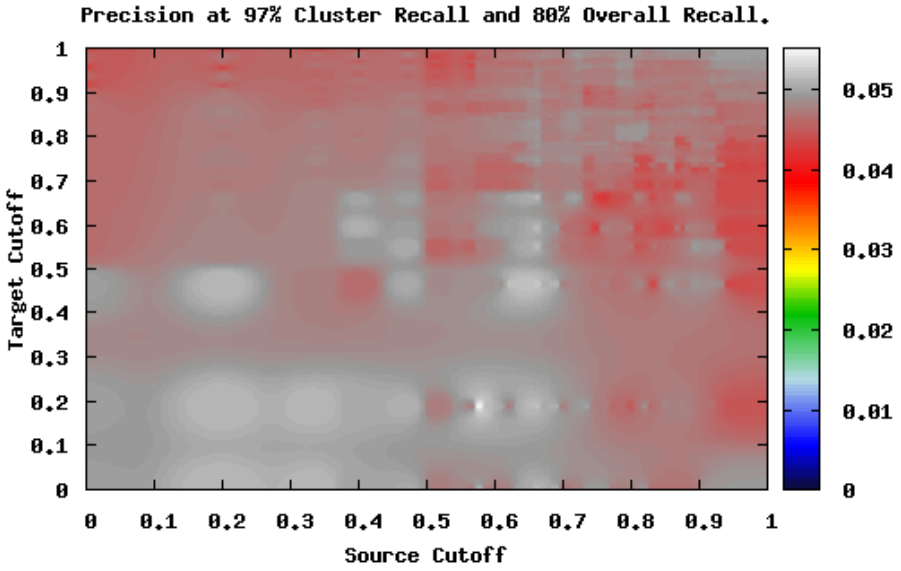

Figure A.8: Clustered Tracing on CCHIT dataset - Precision at 97% Cluster Recall and 80% Overall Recall.

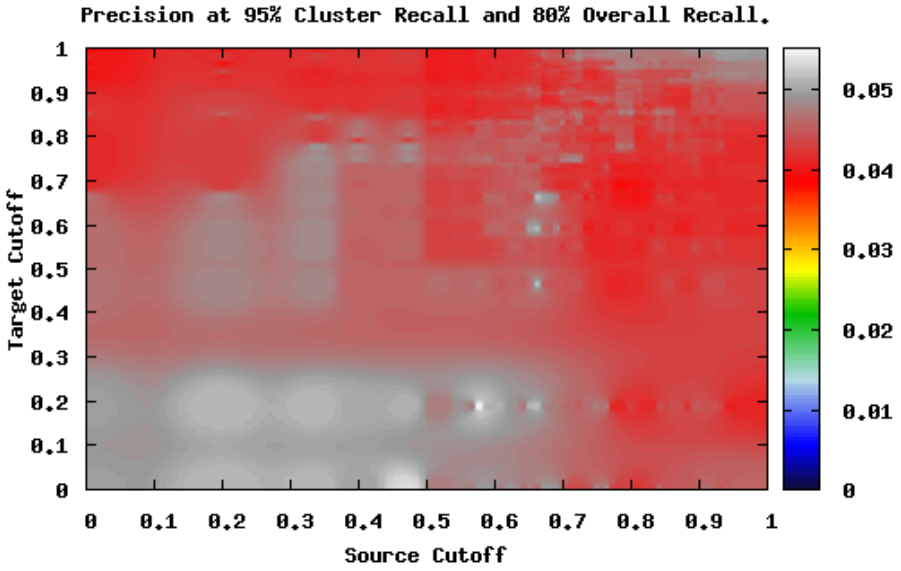

Figure A.9: Clustered Tracing on CCHIT dataset - Precision at 95% Cluster Recall and 80% Overall Recall.

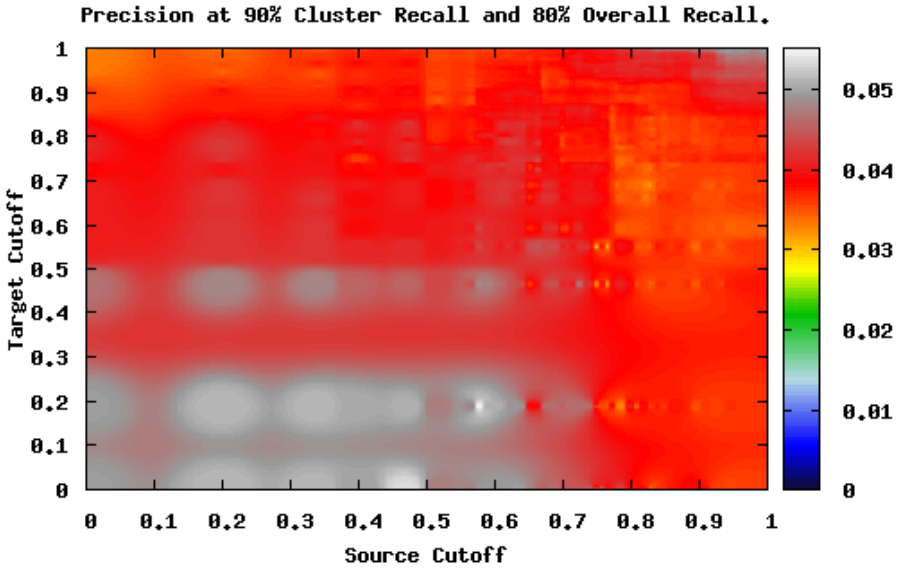

Figure A.10: Clustered Tracing on CCHIT dataset - Precision at 90% Cluster Recall and 80% Overall Recall.

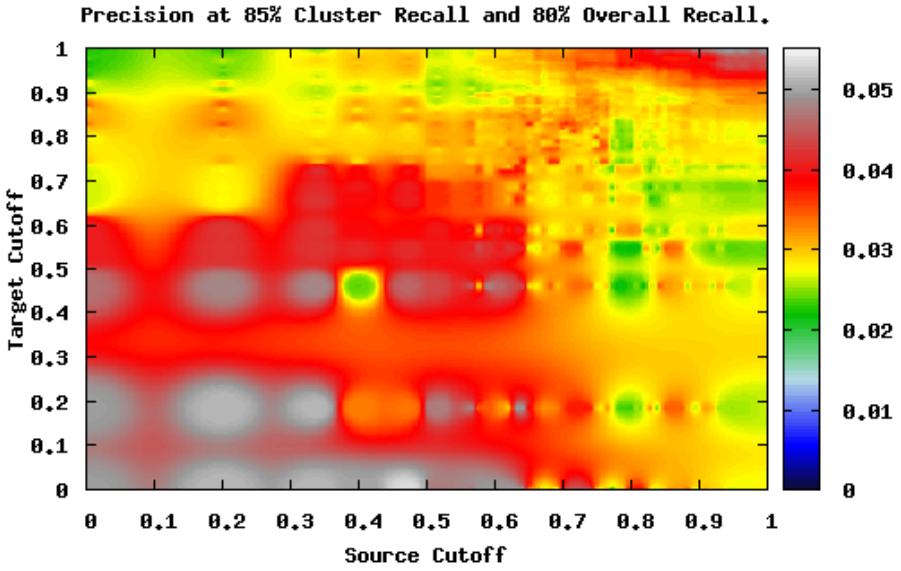

Figure A.11: Clustered Tracing on CCHIT dataset - Precision at 85% Cluster Recall and 80% Overall Recall.

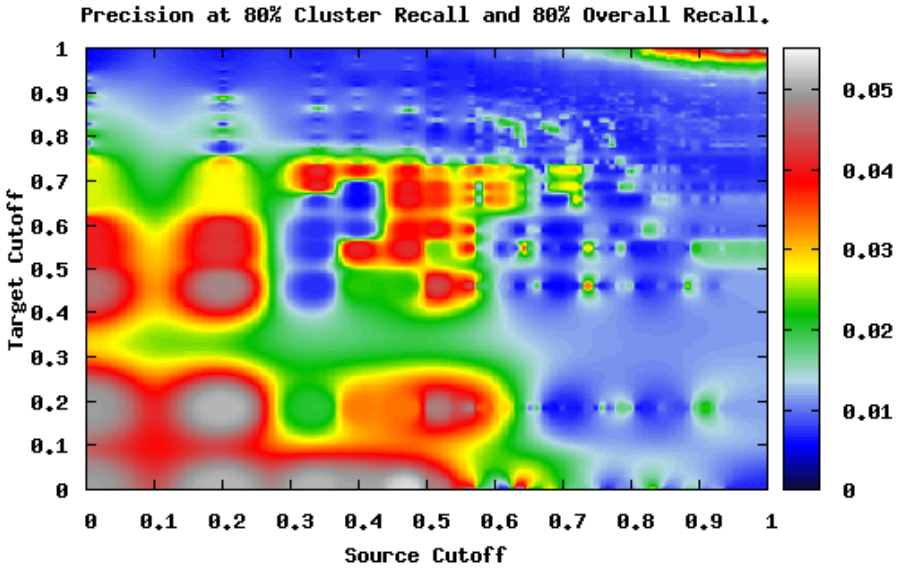

Figure A.12: Clustered Tracing on CCHIT dataset - Precision at 80% Cluster Recall and 80% Overall Recall.

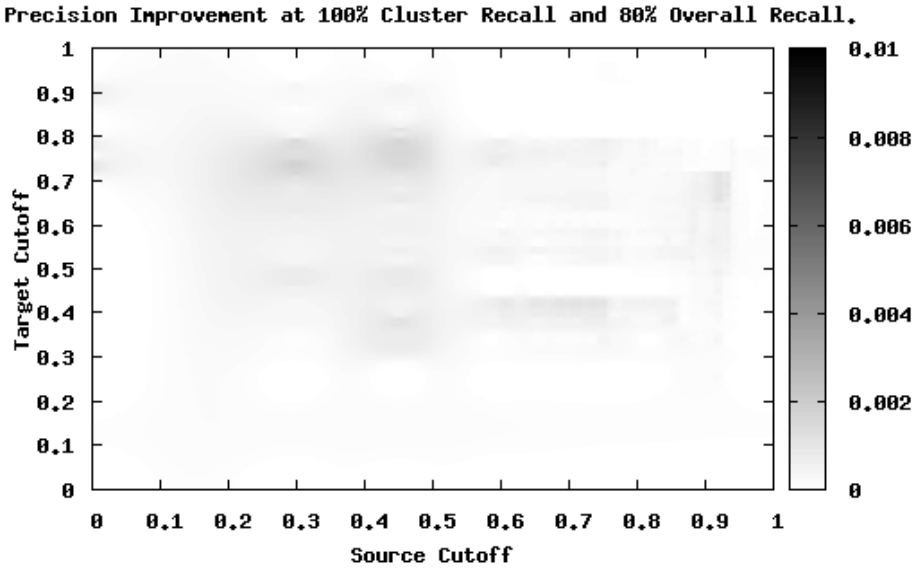

Figure A.13: Clustered Tracing on CM1 dataset - Precision at 100% Cluster Recall and 80% Overall Recall.

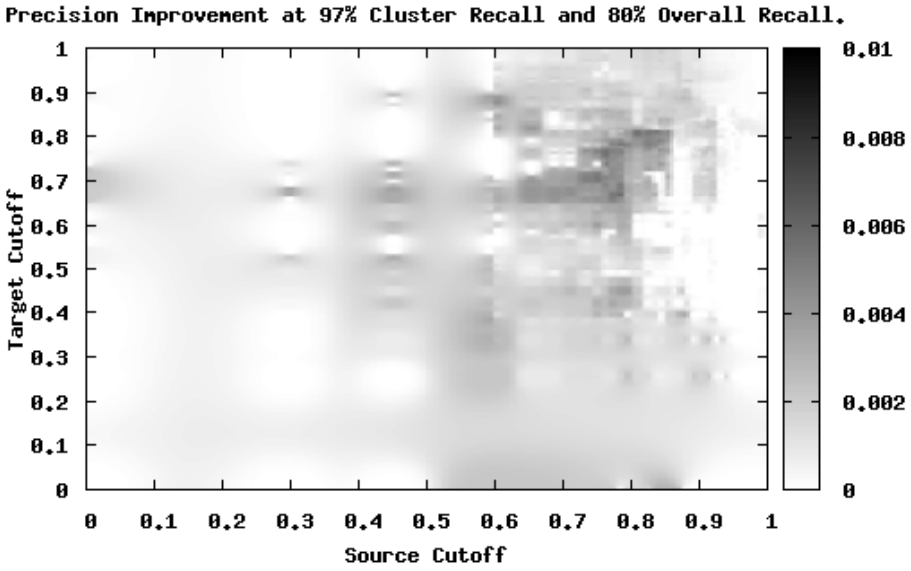

Figure A.14: Clustered Tracing on CM1 dataset - Precision at 97% Cluster Recall and 80% Overall Recall.

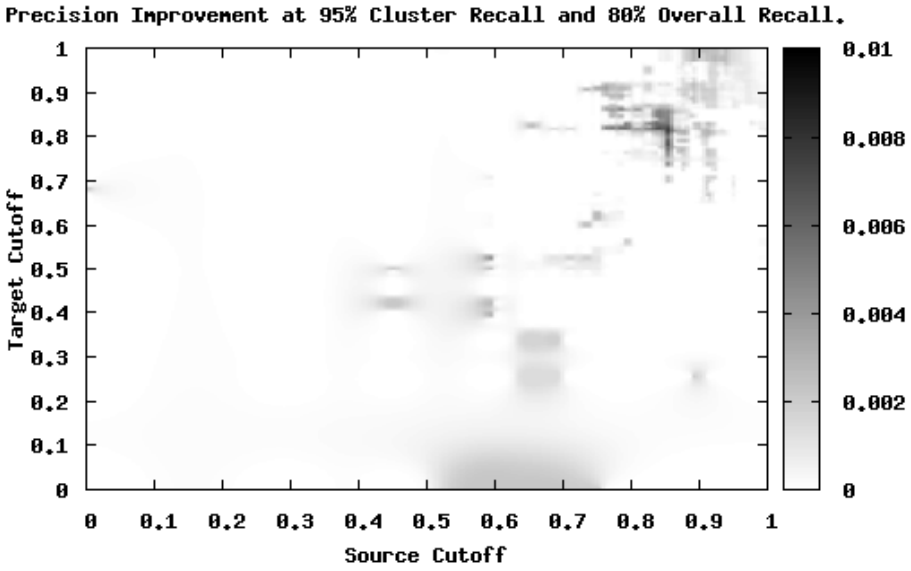

Figure A.15: Clustered Tracing on CM1 dataset - Precision at 95% Cluster Recall and 80% Overall Recall.

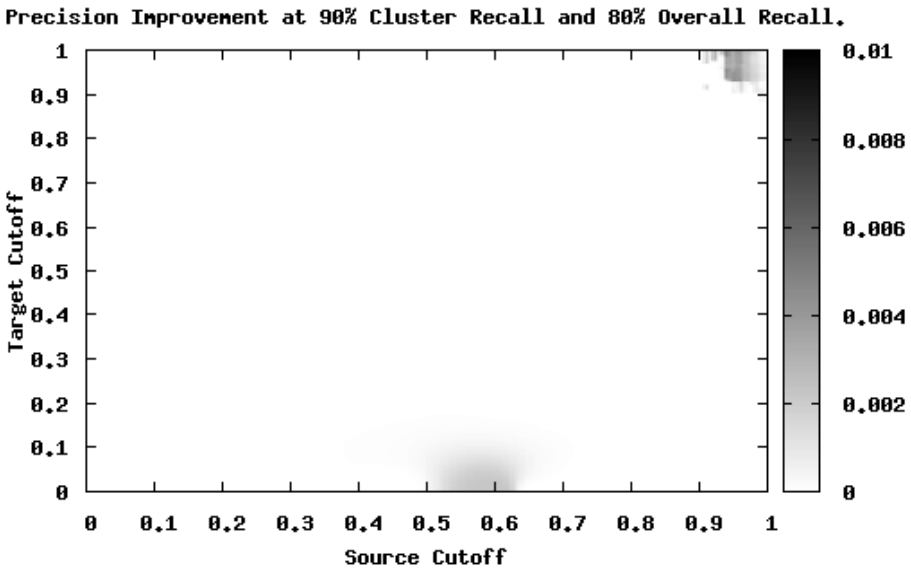

Figure A.16: Clustered Tracing on CM1 dataset - Precision at 90% Cluster Recall and 80% Overall Recall.

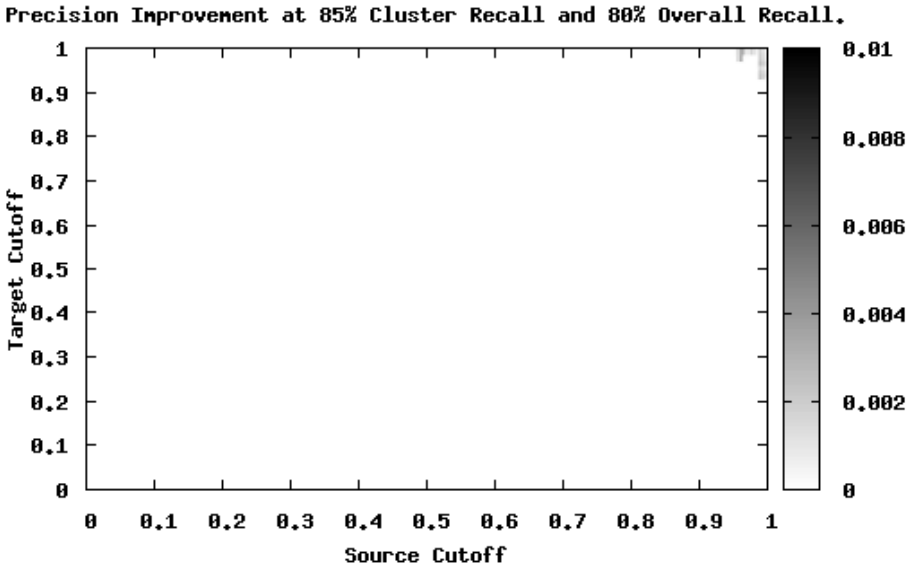

Figure A.17: Clustered Tracing on CM1 dataset - Precision at 85% Cluster Recall and 80% Overall Recall.

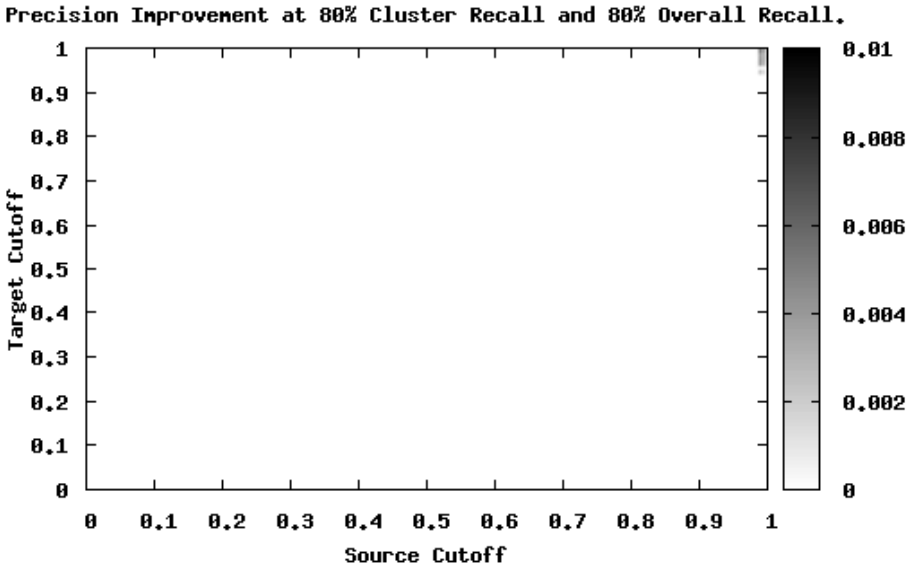

Figure A.18: Clustered Tracing on CM1 dataset - Precision at 80% Cluster Recall and 80% Overall Recall.

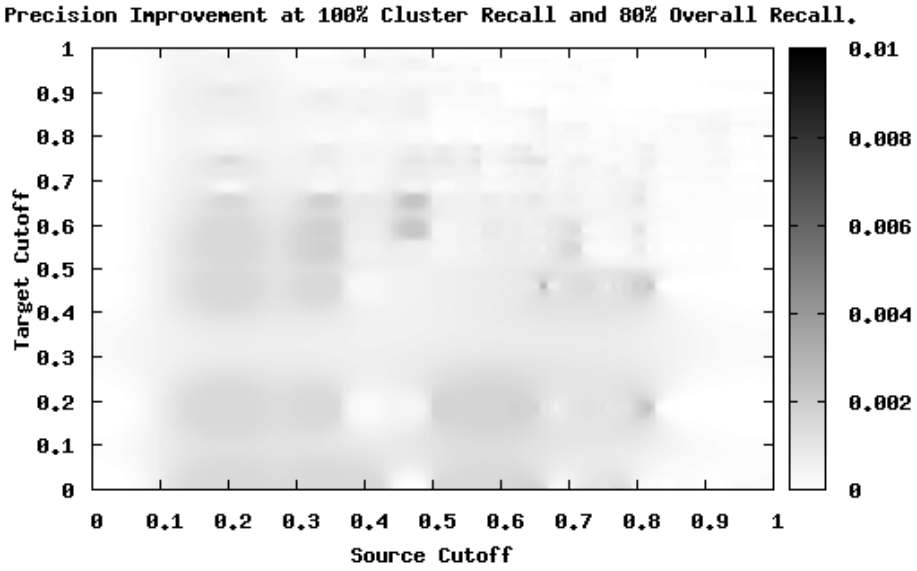

Figure A.19: Clustered Tracing on CCHIT dataset - Precision at 100% Cluster Recall and 80% Overall Recall.

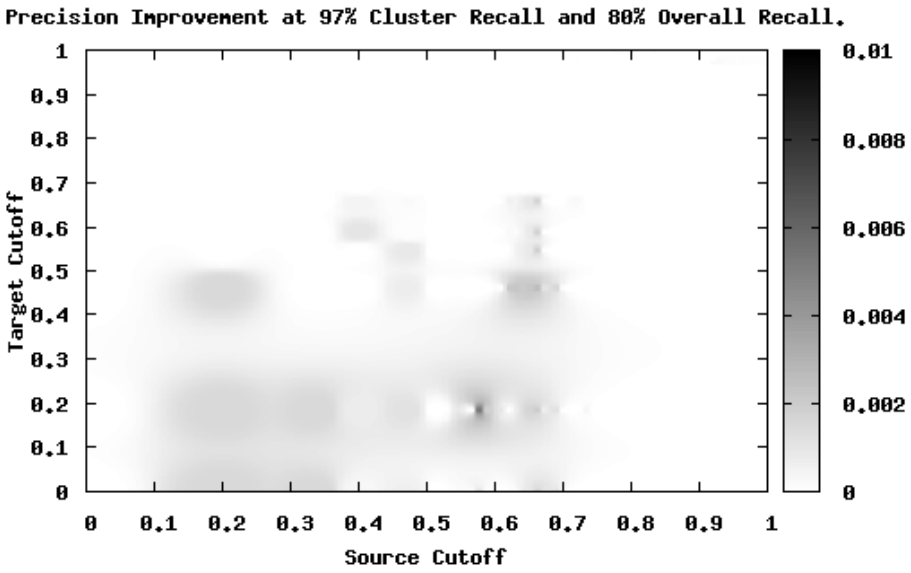

Figure A.20: Clustered Tracing on CCHIT dataset - Precision at 97% Cluster Recall and 80% Overall Recall.

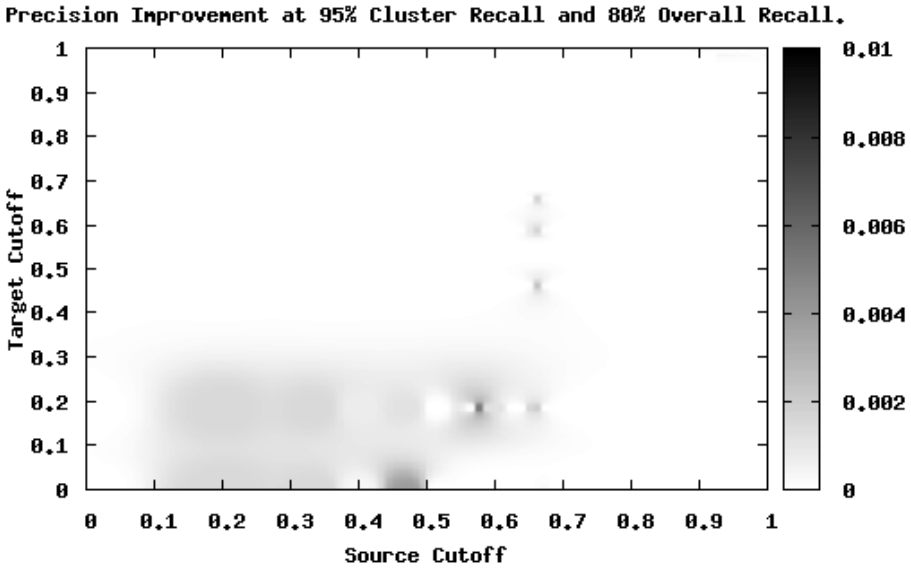

Figure A.21: Clustered Tracing on CCHIT dataset - Precision at 95% Cluster Recall and 80% Overall Recall.

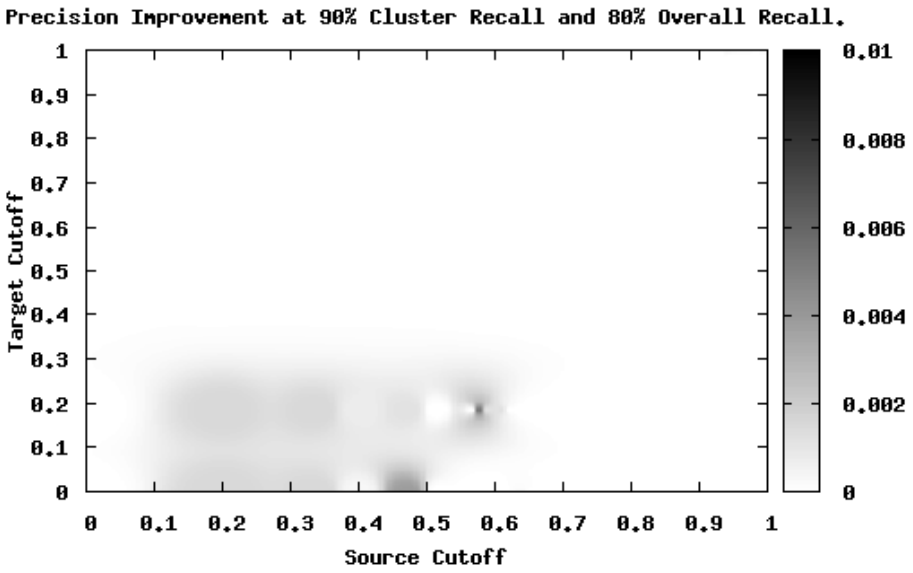

Figure A.22: Clustered Tracing on CCHIT dataset - Precision at 90% Cluster Recall and 80% Overall Recall.

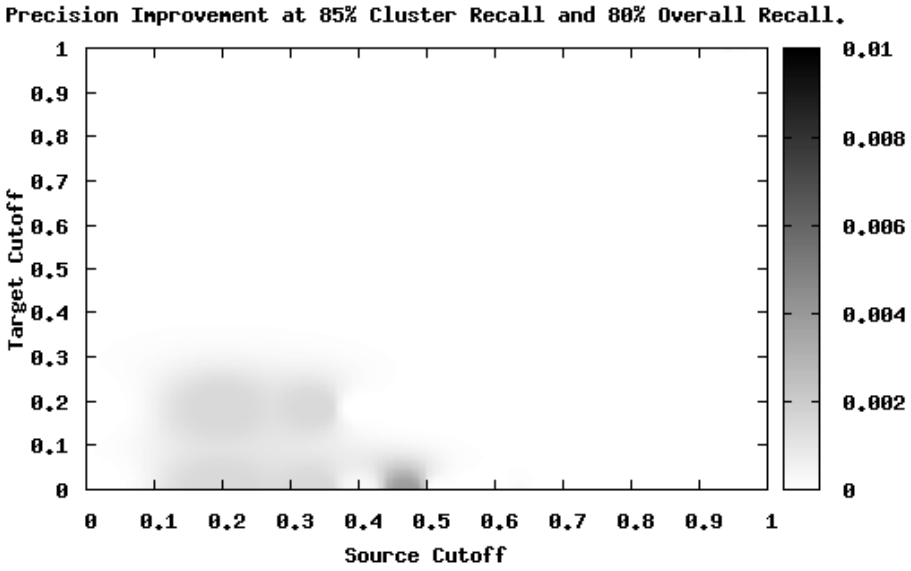

Figure A.23: Clustered Tracing on CCHIT dataset - Precision at 85% Cluster Recall and 80% Overall Recall.

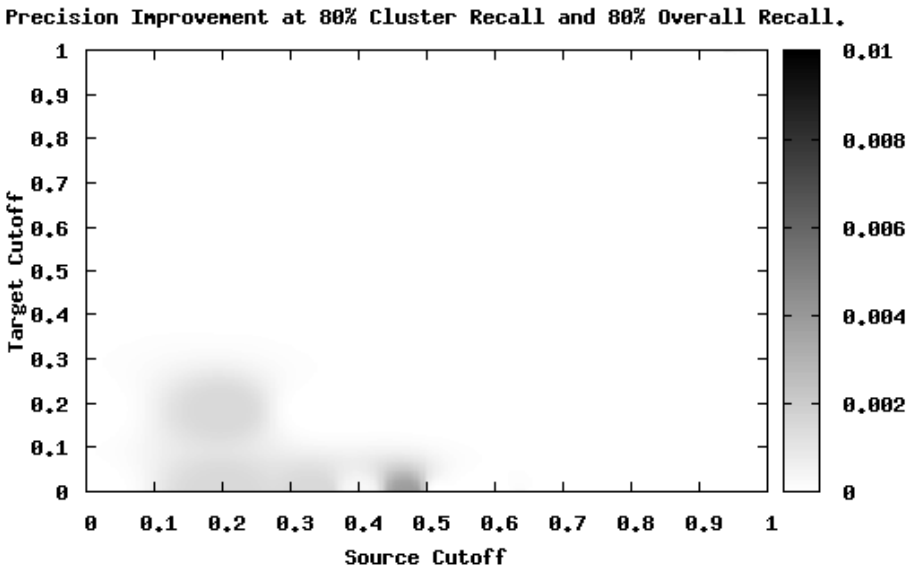

Figure A.24: Clustered Tracing on CCHIT dataset - Precision at 80% Cluster Recall and 80% Overall Recall.