INAUGURAL – DISSERTATION

submitted

to the

Combined Faculty for the Natural Sciences and Mathematics

at Heidelberg University, Germany,

for the degree of

Doctor of Natural Sciences.

Put forward by

M.Sc. Thorsten Merten

Born in Wissen

Oral examination:

IDENTIFICATION OF SOFTWARE FEATURES IN ISSUE TRACKING SYSTEM DATA

THORSTEN MERTEN

2016

Supervisor: Prof. Dr. Barbara Paech
2. Supervisor: Prof. Dr. Kurt Schneider

To Nicole, Annie, and future family members.

ABSTRACT

The knowledge of Software Features (SFs) is vital for software developers and requirements specialists during all software engineering phases: to understand and derive software requirements, to plan and prioritize implementation tasks, to update documentation, or to test whether the final product correctly implements the requested SF. In most software projects, SFs are managed in conjunction with other information such as bug reports, programming tasks, or refactoring tasks with the aid of Issue Tracking Systems (ITSs). Hence ITSs contains a variety of information that is only partly related to SFs.

In practice, however, the usage of ITSs to store SFs comes with two major problems: (1) ITSs are neither designed nor used as documentation systems. Therefore, the data inside an ITS is often uncategorized and SF descriptions are concealed in rather lengthy. (2) Although an SF is often requested in a single sentence, related information can be scattered among many issues. E.g. implementation tasks related to an SF are often reported in additional issues. Hence, the detection of SFs in ITSs is complicated: a manual search for the SFs implies reading, understanding and exploiting the Natural Language (NL) in many issues in detail. This is cumbersome and labor intensive, especially if related information is spread over more than one issue.

This thesis investigates whether SF detection can be supported automatically. First the problem is analyzed: (i) An empirical study shows that requests for important SFs reside in ITSs, making ITSs a good target for SF detection. (ii) A second study identifies characteristics of the information and related NL in issues. These characteristics represent opportunities as well as challenges for the automatic detection of SFs.

Based on these problem studies, the Issue Tracking Software Feature Detection Method (ITSoFD), is proposed. The method has two main components and includes an approach to preprocess issues. Both components address one of the problems associated with storing SFs in ITSs. ITSoFD is validated in three solution studies: (I) An empirical study researches how NL that describes SFs can be detected with techniques from Natural Language Processing (NLP) and Machine Learning. Issues are parsed and different characteristics of the issue and its NL are extracted. These characteristics are used to classify the issue's content and identify SF description candidates, thereby approaching problem (1). (II) An empirical study researches how issues that carry information potentially related to an SF can be detected with techniques from NLP and Information Retrieval. Characteristics of the issue's NL are utilized to create a traceability network

of related issues, thereby approaching problem (2). (III) An empirical study researches how NL data in issues can be preprocessed using heuristics and hierarchical clustering. Code, stack traces, and other technical information is separated from NL. Heuristics are used to identify candidates for technical information and clustering improves the heuristic's results. The technique can be applied to support components, I. and II.

ZUSAMMENFASSUNG

Software Features (SFs) sind zentrale Artefakte für die Softwareentwicklung und das Anforderungsmanagement. SFs werden beispielsweise genutzt, um Anforderungen zu verstehen, abzuleiten oder zu dokumentieren. Oft stützt sich auch die Planung der Entwicklungsarbeiten und die Dokumentation auf SFs. In der Praxis werden SFs meist in Verbindung mit anderen Informationen, wie Fehlerbeschreibungen, Entwicklungs- und Refactoring-Aufgaben in einem Issue Tracking System (ITS) verwaltet. Demnach beinhalten ITSe meist eine Vielzahl von Informationen, die jedoch nur teilweise mit SFs in Zusammenhang stehen.

Die Verwaltung von SFs in ITSen bringt in der Praxis jedoch zwei große Probleme mit sich: (1) ITSe wurden zur Unterstützung der Softwareentwicklung, nicht aber für die Dokumentation erstellt. Daher sind die Daten in ITSen oft falsch kategorisiert und SFs verbergen sich in ausschweifenden Beschreibungen oder Kommentaren. (2) Auch wenn SFs meist mit nur einem Satz beschrieben werden, so befinden sich verwandte Informationen überall im ITS. Beispielsweise werden zugehörige Implementierungsaufgaben oft in einem neuen Issue festgehalten. Somit ist die Erkennung von SFs eine schwierige Aufgabe: Um SFs manuell zu finden, müssen mehrere Issues inklusive der Kommentare im Detail gelesen und bewertet werden. Dies ist sehr aufwändig, insbesondere wenn darüberhinaus noch verwandte Informationen aus mehreren Issues zusammengetragen werden müssen.

Die vorliegende Arbeit untersucht, inwiefern SFs automatisch erkannt werden können und analysiert zunächst das Problem: (i) Eine empirische Studie zeigt, dass wichtige SFs in ITSen gefunden werden können und ITSen dadurch ein gutes Ziel für die automatische Erkennung darstellen. (ii) Eine weitere Studie identifiziert Charakteristiken der Informationen und natürlichsprachlichen Formulierungen in Issues. Diese Charakteristiken wiederum stellen Herausforderungen, aber auch Chancen, für eine automatische Detektion von SFs dar.

Basierend auf der Problem-Analyse wird die Issue Tracking Software Feature Detection Method (ITSoFD), eine Methode zur Detektion von SFs in ITSen, vorgestellt. ITSoFD hat zwei Hauptkomponenten und adressiert die beiden Probleme, die sich durch die Verwaltung von SFs in ITSen ergeben. ITSoFD wird in drei Studien validiert: (I) In einer erstem empirischen Studie wird untersucht, inwiefern SFs mit Techniken aus dem Natural Language Processing (NLP) und dem Machine Learning erkannt werden können. Hierbei werden verschiedene Charakteristiken der Issues und der natürlichen Sprache extrahiert und zur Klassifizierung von Issues genutzt. Diese Studie untersucht Problem (1). (II) In einer zweiten empirischen Studie wird untersucht, inwie-

fern in Beziehung stehende Informationen in verschiedenen Issues durch Techniken des NLP und Information Retrieval zusammengeführt werden können. Es werden verschiedene Charakteristiken der natürlichen Sprache genutzt, um verwandte Issues miteinander zu verlinken. Diese Studie untersucht Problem (2). (III) In einer dritten empirischen Studie wird untersucht, inwiefern technische Informationen wie Code und Stack Traces in Issues von natürlicher Sprache getrennt werden können. Heuristiken werden genutzt, um Kandidaten für technische Informationen zu bestimmen und diese Kandidaten werden durch Clustering zusammengefasst um Falscherkennungen durch die Heuristiken auszugleichen. Diese Technik wird als Vorverarbeitung für obige Komponenten eingesetzt.

ACKNOWLEDGMENTS

This work was partly funded by the Bonn-Rhein-Sieg University of Applied Sciences Graduate Institute. I express my gratitude to Prof. Dr. Ing. Rainer Herpers and Dr. Rita Cornely. Your continuous endeavors push the boundaries of the Graduate Institute and pave the way for many PHD theses at BRSU including mine.

In 1992 my mom and dad unknowingly send me on my computer science journey due to a generous gift: a Commodore 64. This computer exposed me to the world of software creation and revealed how *powerful* and simultaneously *hard* software creation can be by programming my first text adventures. Since then many stations of my life involved mastering computers and eventually I came to understand that the world of software creation is not the world of computers: it is the world of humans. Humans, who make different decisions, humans who follow different paths, and humans who create individual solutions.

Fast forwarding 20 years, Prof. Dr. Barbara Paech and the Software Engineering Group at Heidelberg University gave me the opportunity to investigate both aspects that always fascinated me about software: in this thesis I explore *how powerful software is* by applying of machine learning and information retrieval techniques and I study *how hard software creation is* for humans by exploring the artifacts issued during software creation. I thank Barbara and all the friendly, inspiring, challenging and creative people, at the Software Engineering Group for their continuous support and their collaboration as co-authors, reviewers, and discussion partners. Furthermore, I would like to thank Prof. Dr. Kurt Schneider, whom I first met at a conference workshop where he immediately asked challenging questions about my work, for being the second supervisor of this thesis.

Being located at University of Applied Sciences Bonn-Rhein-Sieg while undertaking the research for this thesis at the Heidelberg University's Combined Faculty for the Natural Sciences and Mathematics gave me the unique opportunity to connect with more magnificent people: I thank Prof. Dr. Simone Bürsner and my colleagues at BRSU for endless discussions about this thesis and science in general. I thank my students for their helpful and inspiring work in seminars, B. Sc., and M. Sc. theses. Special thanks go to the tamers of the GATE: Bastian Mager and Daniel Krämer. I also thank Matúŝ Falis for his support and for introducing the joy of Python programming to my world.

Speaking of GATE and Python: this thesis would not have been possible without the open source movement. I thank the creators of

the countless open software packages that I used in this thesis and the participants of the open source projects that I researched. I try hard to give something back to the community by reporting issues, crating patches, and – of course – by publishing my own source code and data.

Finally, the most important 'thank you' goes to Nicole and Annie. I thank you both for giving me the time that I needed in front of my laptop and together with my peers: for time at the office, time at home, time in the cars passenger seat and sometimes even time during our holidays. Nicole: you questioned, understood and ultimately supported countless decisions in conjunction with this thesis. Your love, your trust and your strengthening helped me to finish every sentence. Annie: your arrival made us a family! Your love and your smile reminds me of the important things in life over and over. Every single day.

CONTENTS

I PRELIMINARIES

1 INTRODUCTION 3
- 1.1 Motivation 3
- 1.2 Description of the Problems 4
- 1.3 Research Questions 5
- 1.4 Scientific Contributions 5
  - 1.4.1 Empirical Findings on SFs in ITSs 5
  - 1.4.2 A Method to Detect Software Features in ITSs 6
  - 1.4.3 An Approach to Separate Technical Data from Natural Language 6
  - 1.4.4 An Approach to Detect SF descriptions in ITSs 6
  - 1.4.5 Empirical Findings on Trace Recovery in ITSs 6
- 1.5 Structure of the Thesis 6
- 1.6 Previous Publications 7

II BACKGROUND AND DEFINITIONS

2 SOFTWARE FEATURE FUNDAMENTALS 13
- 2.1 Definitions 14
  - 2.1.1 ITS Data 14
  - 2.1.2 Software Feature Request 14
- 2.2 Management with Issue Tracking Systems 16

3 TEXT PREPROCESSING FUNDAMENTALS 21
- 3.1 Definitions 21
  - 3.1.1 Language Processing and Engineering 21
  - 3.1.2 Documents and Corpora 22
  - 3.1.3 Preprocessing 23
- 3.2 Preprocessing Techniques 24

4 INFORMATION RETRIEVAL FUNDAMENTALS 29
- 4.1 Definitions 29
  - 4.1.1 Term Frequencies 29
  - 4.1.2 Traceability Matrix 30
  - 4.1.3 Trace Retrieval 30
- 4.2 Information Retrieval Techniques 31

5 TEXT AND DATA MINING FUNDAMENTALS 33
- 5.1 Definitions 33
- 5.2 Text and Data Mining Techniques 35
- 5.3 Machine Learning Techniques and ITS Data 36

6 MEASUREMENT FUNDAMENTALS 39
- 6.1 Precision and Recall in Information Retrieval 39
- 6.2 Precision and Recall in Classification 39
- 6.3 The F Measure 40

6.4 The MaxRP Measure 41

III GENERAL STUDY SETUP

7 RESEARCH METHODOLOGY 45

8 DATA ACQUISITION 49
8.1 Researched Projects 49
8.2 Project Characteristics 51
8.3 Content Analysis and Gold Standard Creation 53

9 TOOL SUPPORT 55
9.1 General Architecture for Text Engineering 55
9.2 The OpenTrace Workbench 56
9.3 Natural Language Toolkit and Scikit-learn 56

IV PROBLEM INVESTIGATION

10 SOFTWARE FEATURES IN ISSUE TRACKING SYSTEMS – AN EMPIRICAL STUDY 61
10.1 Study Design 61
10.1.1 Research Questions 62
10.1.2 Data 62
10.1.3 Analysis Procedures 62
10.2 Results 67
10.2.1 Results for Project Radiant CMS 67
10.2.2 Results for Project Mixxx 71
10.2.3 Results for Project Apache OFBiz 75
10.2.4 Overall Results 78
10.3 Threats to Validity 80
10.4 Related Studies 80
10.5 Implications for Software Feature Detection 81

11 ISSUE TYPES AND INFORMATION TYPES – AN EMPIRICAL STUDY 83
11.1 Study Design 83
11.1.1 Research Questions 83
11.1.2 Data 84
11.1.3 Analysis Procedures 85
11.2 Results 87
11.2.1 Information Types and Issue Types 88
11.2.2 Distribution of Issue and Information Types 90
11.2.3 Project Differences 94
11.3 Threats to Validity 96
11.4 Related Studies 96
11.5 Implications for Software Feature Detection 97

12 SUMMARY OF INVESTIGATION STUDIES 99

V ITSOFD: THE ISSUE TRACKING SOFTWARE FEATURE DETECTION METHOD

13 DESIGNING ITSOFD 103
13.1 Challenges in Software Feature Detection 103

13.2 Solution Design 104
13.2.1 ITSoFD Phase 1: Issue Extraction 105
13.2.2 ITSoFD Phase 2: Issue Preprocessing 105
13.2.3 ITSoFD Phase 3: SFR Detection 113
13.2.4 ITSoFD Phase 4: Related Issue Tracing 117
13.3 Related Work 118
13.3.1 ITS and Task Specific Preprocessing 119
13.3.2 SFR Detection 120
13.3.3 Trace Recovery in ITS Data 123
14 SEPARATING NATURAL LANGUAGE AND TECHNICAL DATA – AN EMPIRICAL STUDY 125
14.1 Study Design 125
14.1.1 Research Question 126
14.1.2 Data 126
14.1.3 Evaluation Procedures 126
14.2 Results 127
14.3 Discussion 129
14.4 Threats to Validity 130
14.5 Conclusion 131
15 DETECTING SOFTWARE FEATURE REQUESTS IN ISSUES – AN EMPIRICAL STUDY 133
15.1 Study Design 133
15.1.1 Research Questions 134
15.1.2 Data 135
15.1.3 Algorithms and Settings 137
15.1.4 Evaluation Procedures 140
15.2 Results 140
15.2.1 Best Preprocessing Techniques 140
15.2.2 MLFs and Detection Levels 143
15.2.3 Cross-training 147
15.3 Discussion 149
15.4 Threats to Validity 151
15.5 Conclusion 152
16 RECOVERING RELATED ISSUES IN ISSUE TRACKING SYSTEMS – AN EMPIRICAL STUDY 153
16.1 Study Design 153
16.1.1 Research Questions 154
16.1.2 Data 154
16.1.3 Algorithms and Settings 157
16.1.4 Evaluation Procedures 158
16.2 Results 158
16.2.1 IR Algorithm Performance on ITS Data 158
16.2.2 ITS-specific Preprocessing and Weighting 160
16.2.3 Trace Types and Issue Types 161
16.2.4 Results per Project 162
16.3 Discussion 165

16.4 Threats to Validity 166
16.5 Conclusion 166

VI CONCLUSION
17 DISCUSSION 171
18 SUMMARY 173
19 FUTURE WORK 175

VII APPENDICES
A CODING GUIDELINES 179
A.1 Coder Distribution and Agreement Factors 179
A.2 Coding Guidelines 179
B DISTRIBUTION OF ISSUE AND INFORMATION TYPES 185
C CLARIFICATION AND SOLUTION DETECTION 187
C.1 Plots for the Clarification label 188
C.2 Plots for the Solution label 190
C.3 Cross Training Impact for Clarification and Solution 192
D REGULAR EXPRESSIONS TO DETECT TECHNICAL INFORMATION 195

BIBLIOGRAPHY 197

LIST OF FIGURES

Figure 2.1 Typical ITS and Issue Structure. 13
Figure 2.2 Typical Issue Tracking System Workflow. 17
Figure 2.3 Excerpts of Two Issues From the Redmine Project. 17
Figure 3.1 Sentence and Word Tokenization Examples. 24
Figure 3.2 Grammatical Tagging Example. 25
Figure 3.3 Grammatical Parsing Example. 26
Figure 9.1 Data Annotation with GATE. 55
Figure 10.1 Feature Graph Definition. 66
Figure 10.2 Radiant Feature Graph. 69
Figure 10.3 Radiant: Feature Abstraction Levels. 70
Figure 10.4 Radiant: Feature Relations. 70
Figure 10.5 Mixxx Feature Graph. 72
Figure 10.6 Mixxx: Feature Abstraction Levels. 74
Figure 10.7 Mixxx: Feature Relations. 75
Figure 10.8 Abstraction Levels for Software Features. 78
Figure 11.1 Issue Sizes. 85
Figure 11.2 Issue Types and Information Types: Research Process Overview. 86
Figure 11.3 Issue Types and Information Types: A Taxonomy. 88
Figure 13.1 ITSoFD Overview (Part 1). 106
Figure 13.2 ITSoFD Overview (Part 2). 107
Figure 13.3 Example for Clustering Steps. 111
Figure 13.4 Bottom-up Clustering. 112
Figure 13.5 ITSoFD Results for Each Phase. 114
Figure 13.6 Text Annotation with GATE. 116
Figure 14.1 Issue Preprocessing: Study Setup. 125
Figure 14.2 Precision and Recall for Different Cluster Similarities. 129
Figure 14.3 Example of Intermingled Code and NL from Apache Thrift. 130
Figure 15.1 Software Feature Detection: Study Setup. 134
Figure 15.2 Redmine Issue Example. 135
Figure 15.3 Detailed Examples for Preprocessing Setting Influences. 143
Figure 15.4 F_1 Scores for Request Functionality Detection. 145
Figure 15.5 $MAX(R), P_{\geqslant 0.2}$ and $MAX(R), P_{\geqslant 0.05}$ Scores for Request Functionality Detection. 146
Figure 16.1 ITS Trace Recovery: Study Setup. 153
Figure 16.2 Gold Standard Creation Tool. 156
Figure 16.3 Best F_1 and F_2 Scores for Every IR Algorithm. 159

Figure 16.4 Best Results With and Without Removing Noise. 160
Figure 16.5 Influence of Term Weighting. 161
Figure C.1 F_1 Scores for Clarification Detection. 188
Figure C.2 $MAX(R), P_{\geqslant 0.2}$ and $MAX(R), P_{\geqslant 0.05}$ Scores for Clarification Detection. 189
Figure C.3 F_1 Scores for *Solution* Detection. 190
Figure C.4 $MAX(R), P_{\geqslant 0.2}$ and $MAX(R), P_{\geqslant 0.05}$ Scores for Solution Detection. 191

LIST OF TABLES

Table 1.1 Structure and Contributions of this Thesis. 8
Table 2.1 ITS Data Fields in Several ITSs. 19
Table 4.1 Example Traceability Matrix. 30
Table 6.1 Trace Retrieval Evaluation Measures. 39
Table 8.1 Mapping Projects to Studies. 50
Table 8.2 Project Characteristics. 52
Table 10.1 Issues and User Documentation. 63
Table 10.2 Rules to Identify Feature Relevant Information. 64
Table 10.3 Examples for Abstraction Levels in Issues. 66
Table 10.4 Radiant Features. 67
Table 10.5 Mixxx Features. 73
Table 10.6 OFBiz: Manufacturing Component Features. 76
Table 10.7 OFBiz: Commonalities and Differences of ITS and UD. 77
Table 11.1 Population, Sample and Coding Sizes. 87
Table 11.2 Distribution of Annotated Issue Types. 91
Table 11.3 Top Combinations of Issue Types. 91
Table 11.4 Most Used Information Types. 93
Table 11.5 Top TF-IDF Scores. 94
Table 13.1 Implications of Investigation Studies for the Solution Design. 103
Table 13.1 Implications of Investigation Studies for the Solution Design (Continued). 104
Table 13.2 Summary of Manual Activities in ITSoFD. 108
Table 14.1 Corpora for NL and Technical Separation. 127
Table 14.2 Results for Line Tokenization. 128
Table 14.3 Results for White Space Tokenization. 128
Table 15.1 Software Feature Detection: Extracted Data and Annotations. 137
Table 15.2 Evaluated Preprocessing Techniques and Linguistic MLF-Sets. 141
Table 15.3 Evaluated Linguistic MLF-sets. 141

Table 15.4 Average and Maximum F_1 Scores for Preprocessing. 142
Table 15.5 Machine Learning Feature Sets. 144
Table 15.6 Best F_1 and $MAX(R)$, $P_{\geqslant p}$ Scores. 147
Table 15.7 Best Cross-Training F_1 and $MAX(R)$, $P_{\geqslant p}$ Scores. 148
Table 16.1 Extracted Traces Versus Gold Standard. 157
Table 16.2 Algorithms and Preprocessing Settings. 157
Table 16.3 Data Fields Weights. 158
Table 16.4 Best Results for Different Issue Types. 163
Table 16.5 Best Results for Different Trace Types. 164
Table 16.6 Best Results per Project. 164
Table A.1 Coder Distribution per Project. 179
Table A.2 Coder Agreement Factors. 180
Table B.1 Complete Information Type Count. 186
Table C.1 Reference Machine Learning Feature Sets. 187
Table C.2 Clarification: Best Cross-Training F_1 and $MAX(R)$, $P_{\geqslant}$ Scores. 192
Table C.3 Solution: Best Cross-Training F_1 and $MAX(R)$, $P_{\geqslant}$ Scores. 193

ACRONYMS

API Application Programming Interface

BOW Bag of Words

BM25 Okapi Best Matching 25

CL Computational Linguistics

DM Data Mining

DT Decision Tree

DTM Developer Trace Matrix

FP False Positive

FN False Negative

GATE General Architecture for Text Engineering

GUI Graphical User Interface

GSTM Gold Standard Trace Matrix

IEEE Institute of Electrical and Electronics Engineers

IREB International Requirements Engineering Board e.V.

ITS Issue Tracking System

ITSoFD Issue Tracking Software Feature Detection Method

IR Information Retrieval

LE Language Engineering

LR Logistic Regression

LOC Lines of Code

LSA Latent Semantic Analysis

ML Machine Learning

MLF Machine Learning Feature

MNB Multinomial Naive Bayes

MSR Mining Software Repositories

NB Naive Bayes

NL Natural Language

NLP Natural Language Processing

NLTK Natural Language Toolkit

O Domain Object

OSS Open Source Software

RA Requirements Artifact

RE Requirements Engineering

RF Random Forrest

RQ Research Question

SE Software Engineering

SGD Stochastic Gradient Descent

SF Software Feature

SQ Software Quality

SFR Software Feature Request

SRS Software Requirements Specification

SVM Linear Support Vector Machine

TF-IDF Term Frequency - Inverse Document Frequency

TP True Positive

TN True Negative

UC Use Case

UD User Documentation

UI User Interface

URL Uniform Resource Locator

VSM Vector Space Model

XML Extensible Markup Language

Part I

PRELIMINARIES

In this part software feature detection is introduced. A motivation for software feature detection is laid out and the three main problems that are investigated in this thesis are described. For these problems four overall research questions are derived to guide the problem investigation systematically. The research questions are further refined and answered throughout the thesis. Finally, an overview of the thesis is given and previous publications of the author containing relevant ideas and concepts are outlined.

1 INTRODUCTION

1.1 MOTIVATION

Software Features (SFs) are heavily used in Requirements Engineering (RE) and the whole Software Engineering (SE) process. Hence the knowledge of SFs is helpful for many SE tasks, such as understanding and deriving Requirements Artifacts (RAs), checking which SFs are considered during implementation, updating documentation, release planning in software product management (Fricker and Schumacher, 2012), or software product line engineering (Kang et al., 1990).

Approaches that support these SE tasks typically assume a dedicated representation of SFs. However, in industrial and Open Source Software (OSS) projects SFs are often managed implicitly with the aid of an Issue Tracking System (ITS)[1]. Moreover, many development projects typically do not have explicit or up to date requirement documents or SF descriptions (Alspaugh and Scacchi, 2013), especially if RE is practiced ad hoc (Ernst and Murphy, 2012).

Advantageously, most SE projects employ ITSs to aid their software development efforts (Skerrett and The Eclipse Foundation, 2011). The ITS is usually the most critical tool to capture development tasks (Ernst and Murphy, 2012) and to describe and discuss SFs. Moreover, the ITS usually includes a broad range of SFs information on different abstraction levels (Paech, Hübner, and Merten, 2014) and it is the place where the views of users and developers are tied together (Bertram et al., 2010; Nguyen Duc et al., 2011) and where knowledge about SFs is build (Hemetsberger, 2006). However, ITS are neither used nor designed as documentation systems and the SFs are typically unstructured and thus hard to find. Although feature and bug related issues are usually categorized by metadata, this metadata is often inaccurate (Herzig, Just, and Zeller, 2013) or missing (Merten et al., 2015) and cannot be used to detect SFs descriptions reliably. In addition, related information with respect to an SF is generally not found in the same issue. E.g. if a bug connected to a certain SF is found, the bug is usually reported in a new issue (Merten et al., 2016a). Hence, the information about an SF in a single issue is not necessarily complete.

Overall, SFs are vastly important for SE, but detecting SF descriptions in ITS is a manual, labor intensive, and tedious task. Further-

1 ITSs are also known as bug trackers, ticket systems, help desk systems, or project management systems. In the following "Issue Tracking System" (ITS) is used as an umbrella term for all these systems.

more, this task is supported by rather basic search functionalities of the ITS only (Tran et al., 2009). Hence, this thesis researches a method to detect SFs related information automatically ITSs.

One driver for this thesis was my investigation into an industrial software development project. In the project environment about 100 issues are collected every week. Usually, these issues are categorized as "SF" or "bug report" respectively[2]. However, in the comments to some of the bug reports SFs descriptions could be found. Often these SFs emerged during a discussion of the customer representative and developers and the developers implemented them during bug resolution. Such "hidden" requests for SFs were a major problem for two reasons:

1. the company could not update the software documentation accordingly, as only issues categorized as SFs were picked up by the documentation team and
2. the company could not bill their customers accordingly, as issues categorized as "bug report" could not be billed.

Although the outcome of this thesis cannot be applied directly to their environment, the thesis represents a first step towards a reliable SF detection within ITSs.

1.2 DESCRIPTION OF THE PROBLEMS

To detect SFs in ITSs this thesis addresses the following three problems P1 to P3:

P1 UNDERSTANDING SFS IN ITS DATA Although researchers analyze the contents, especially the Natural Language (NL) data of ITSs (Bertram, 2009; Herzig, Just, and Zeller, 2013), research does not focus on SFs descriptions and their structure. This results in a limited understanding of many aspects, e.g. whether project relevant SFs are actually recorded in the ITS and if so, how those SFs are described. Little is also known on how SFs are complemented by metadata, or which information besides the SFs resides in an ITS[3]. However, no empirical assessment of these aspects has been made.

P2 DETECTING SF RELATED INFORMATION A manual search for SF descriptions in the ITS is no trivial task in most software projects. Automation is needed to ease this task. Research has shown that SFs

2 Among other categories that are not relevant for this example.

3 Whereas some aspects of NL in ITSs have been researched. E.g. Bertram, 2009 describes social aspects of issue tracking, Ko and Chilana, 2011 research discussions in issues and Zimmermann et al., 2009 and Lotufo, Passos, and Czarnecki, 2012 suggest to improve ITSs from different points of view.

can be detected from requirement documentation or product descriptions (Bakar, Kasirun, and Salleh, 2015) or in many cases from source code (Dit et al., 2013). However, no means to detect SF descriptions in ITSs exist.

P3 IDENTIFYING INFORMATION RELATED TO SFS SFs are often accompanied by other information like related software bugs (Bertram et al., 2010). Traceability research has shown that such related information can be revealed for structured RAs (Borg, Runeson, and Ardö, 2014; Gotel et al., 2012). However, it is unclear whether such relations can be found in ITSs[4].

1.3 RESEARCH QUESTIONS

The following overall Research Questions (RQs) are defined to investigate P1 to P3, where RQ 1 and RQ 2 are related to P1, RQ 3 is related to P2, and RQ 4 is related to P3:

RQ 1 What information about SFs can be found in an ITS of a software product, and how well is this information suited to derive a feature representation of the software?

RQ 2 How is NL information categorized, described, and distributed in an ITS?

RQ 3 Can SFs descriptions be detected automatically in ITS NL data?

RQ 4 Do trace retrieval algorithms perform effectively on ITS data?

These RQs are further refined in respective studies: RQ 1 is refined in three detailed questions in Chapter 10, RQ 2 is refined in three detailed questions in Chapter 11, RQ 3 is refined in three detailed questions in Chapter 15, and finally RQ 4 is refined in four detailed questions in Chapter 16. An overview of the studies and the RQs is given in Table 1.1.

1.4 SCIENTIFIC CONTRIBUTIONS

1.4.1 *Empirical Findings on SFs in ITSs*

Part iv studies the challenges related to SF detection. Chapters 10 and 11 present two empirical studies on ITS data showing that (1) many important SFs can be found in an ITS, hence an ITS is a good target for SF detection, and (2) how these SF are formulated and which related information can be found in issues. Both studies tackle problem P1 defined above.

4 Although promising results for duplicate issue detection have been published, e.g. by Kaushik and Tahvildari, 2012, Tian, Sun, and Lo, 2012, or Borg et al., 2014.

1.4.2 *A Method to Detect Software Features in ITSs*

Part V presents solutions related to SF detection. Based on the findings of the empirical studies describes in the previous sections, the challenges in SF detection are derived and implications for technical solutions are introduced. Then, the Issue Tracking Software Feature Detection Method (ITSoFD) is introduced. ITSoFD is an overall solution design for SF detection. All major components of ITSoFD are validated in three empirical studies, outlined in the following Sections 1.4.3, 1.4.4, and 1.4.5.

1.4.3 *An Approach to Separate Technical Data from Natural Language*

In Chapter 14 an approach to separate technical data form NL is presented and empirically evaluated. In contrast to related work, this approach is based on simple heuristics and clustering and can be applied without an explicit training phase. The approach represents a data preprocessing step to tackle the problems P2 and P3.

1.4.4 *An Approach to Detect SF descriptions in ITSs*

In Chapter 15 an approach to detect SFs descriptions in issues based on Machine Learning (ML) techniques is presented and empirically evaluated. As to the author's knowledge this is the first publication that evaluates ML techniques for SFs detection in ITS data. The approach tackles problem P2.

1.4.5 *Empirical Findings on Trace Recovery in ITSs*

In Chapter 16 an empirical evaluation of Information Retrieval (IR) techniques to detect links between issues is presented. Although related work shows promising results for link detection between duplicated issues or structured and well written RAs, this thesis shows that these approaches cannot be transferred to most ITS data. However, Chapter 16 presents promising enhancements to previous research that can improve IR techniques on ITS data. The empirical evaluation investigates problem P3.

1.5 STRUCTURE OF THE THESIS

In general this thesis is based on two pillars. The first pillar in Part IV investigates in the problems that come with software feature detection and the second pillar in Part V describes solution studies that tackle the identified problems.

Beforehand, Part ii defines important terms and gives background information on SFs and SF requests, ML, and IR.

Part iii describes the data acquisition, the applied research methodologies, and the tools that were used to implement the experiments presented in Chapters 8, 7, and 9, respectively.

The first pillar in Part iv includes two studies investigating whether SFs can be found in ITSs (Chapter 10) and how issues are structured (Chapter 11). Finally, Chapter 12 discusses the overall results of the problem investigation.

The second pillar in Part v includes two approaches investigating whether ITS data can be processed for an automatic SF detection. The preprocessing method in Chapter 14 is an approach to separate technical data from NL and is evaluated in an empirical study. Chapter 15 describes an approach to detect SFs descriptions based on ML techniques and Chapter 16 describes ITS specific advances for trace retrieval techniques to find related issues in ITSs. Both, the approach and the advances are evaluated in an empirical study.

In Part vi the results and findings of the thesis are discussed (Chapter 17), summarized (Chapter 18), and finally implications of the thesis for future work are presented (Chapter 19).

Table 1.1 shows the structure of the thesis, summarizes the conducted experiments, and names the results of every experiment. In addition, Table 1.1 attributes the conducted experiments to the problems described in Section 1.2 and to the RQs stated in Section 1.3.

1.6 PREVIOUS PUBLICATIONS

"Knowledge sharing and collaboration are a key foundation of research and development." – Jonathan Spira.

Parts of the ideas, concepts, algorithms, and evaluation results presented in this thesis have already been published in scientific proceedings. I took the role of the main author in the publications presented below, conducted most of the research, and authored the papers[5]. An exception is (Paech, Hübner, and Merten, 2014): here the second author analyzed SFs in software user documentation and I analyzed SFs in ITSs. All three authors shared the responsibilities for cross-cutting aspects and writing. Altogether, the following publications preceded this thesis:

First ideas with respect to ITS analysis were published as short paper and poster. Due to feedback in the poster presentation, an initially broad range of ideas was focused on SFs detection during the preparation of this thesis.

> Thorsten Merten and Barbara Paech (2013). "Research Preview: Automatic Data Categorization in Issue Tracking

5 As a matter of course, corrections and smaller text passages as provided by my co-authors were incorporated in the publications. Furthermore, my research was often build on initial implementations by my students, who are listed as co-authors.

		DISCUSSES	
	CONTENTS AND RESULTS	PROBLEMS	RQS
Part i	Introduction		
Sec. 1.2	*Description of the Problems*	P1, P2, P3	
Sec. 1.3	*Definition of the Overall RQs*		RQ 1 - 4
Part ii	Fundamentals		
Part iii	Methodologies & Tools		
Part iv	Problem Investigation		
Ch. 10	On Understanding SFs in ITSs *Result: empirical findings on SFs in ITSs*	P1	RQ 1
Ch. 11	On Understanding the Structure of Issues, the Information in ITSs, and the NL Usage in ITSs *Result: empirical findings on issue contents*	P1	RQ 2
Ch. 12	Overall Results: problem Investigation	P1	
Part v	SF request Detection		
Ch. 14	On Preprocessing Issues *Results: a clustering-based method to preprocess issues* and an empirical evaluation of this method*		[RQ 3, RQ 4]
Ch. 15	On Detecting SFs requests *Results: MLF sets to detect SFs descriptions and an empirical evaluation of seven ML algorithms using these MLFs sets.*	P2	RQ 3
Ch. 16	On Finding Related Issues *Results: enhancements to IR-based methods to retrieve traces in ITSs and an empirical evluation of five IR-based methods used with and without these enhancements.*	P3	RQ 4
Part vi	Conclusion & Future Work	P1, P2, P3	

* Results can be applied as preprocessing step in the following studies.

Table 1.1: Structure and Contributions of this Thesis.

Systems." In: *43. Jahrestagung der Gesellschaft für Informatik e.V. (INFORMATIK 2013)*. Koblenz, Germany: Lecture Notes in Informatics, pp. 128–130

Empirical findings about SFs, abstraction levels of SFs descriptions and findings on ITS NL contents were published in:

Barbara Paech, Paul Hübner, and Thorsten Merten (2014). "What are the Features of this Software?" In: *9th International Conference on Software Engineering Advances (ICSEA 2014)*. Nice, France: IARIA XPS Press, pp. 97–106

ITS NL contents were then analyzed in depth. Empirical findings how issues of different types (e.g. feature requests and bug reports) and especially the NL is composed, were published in:

Thorsten Merten et al. (2015). "Requirements Communication in Issue Tracking Systems in Four Open-Source Projects." In: *6th Workshop on Requirements Prioritization and Communication (RePriCo 2015)*. Essen, Germany: CEUR Workshop Proceedings, pp. 114–125

An approach to automatically separate NL and technical information, was published in:

Thorsten Merten et al. (2014). "Classifying Unstructured Data into Natural Language Text and Technical Information." In: *11th Working Conference on Mining Software Repositories (MSR 2014)*. Hyderabad, India: ACM Press, pp. 300–303

An empirical evaluation of IR algorithms for automatic trace retrieval in ITSs and improvements with respect to data preprocessing and weighting was published in:

Thorsten Merten et al. (2016a). "Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data?" In: *22nd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016)*. Vol. 9619. Gothenburg, Sweden: Springer, pp. 45–62

An ML based approach to automatically detect software feature descriptions in ITSs on different levels of detail was published in:

Thorsten Merten et al. (2016b). "Software Feature Request Detection in Issue Tracking Systems." In: *24th IEEE International Requirements Engineering Conference (RE 2016)*. Beijing, China: IEEE, pp. 166–175

Part II

BACKGROUND AND DEFINITIONS

In this part relevant background knowledge is introduced. First, software features and important concepts related to issue tracking systems are defined. Then an overview of the technologies and techniques used in the thesis is given; including text preprocessing, information retrieval, and machine learning. Finally, measures used in experiment validation are described and a new measure to evaluate software feature detection is introduced.

2 SOFTWARE FEATURE FUNDAMENTALS

This paragraph gives a brief overview on the way data in is organized in ITSs. With this background different types of ITS data are defined as they are referred to throughout the thesis. Then, Section 2.2 discusses how ITSs are used to manage SFs in detail.

Figure 2.1 shows the typical structure of an ITS. The ITS is represented by the gray area in figure Figure 2.1. Every solid box in the ITS represents an issue i_n and the dotted and dashed boxes shown in issue i_0 are examples for the typical data fields of an issue. A description of the data fields is shown on the left hand side and related example data is given on the right hand side. An issue is usually comprised of a title, a description, comments, and meta-data. Meta-data includes but is not limited to information such as the user who created the issue, the date and time of issue creation, or users who commented on the issue. For this thesis the most important data is the NL text in an issue's title, description and its comments[1].

Data Field	**Example Data**
ITS	Radiant CMS using GitHub ITS
Issue (i_0)	`https://github.com/radiant/radiant/issues/25`
Metadata	ID, time, date, reporter, # of comments, issue type, ...
Title	>Internationalization<
Description	>Provide support for internationalization of the admin interface for Radiant and extensions<
Comment ($c_0 \in i_0$)	
Metadata	time, date, user name, ...
Description	>come on! a cms in version 0.9 without any internationalization its just useless... no?<
Comment ($\cdots \in i_0$)	...
Comment ($c_n \in i_0$)	c_{24}
Issue (i_1)	...
Issue (i_m)	i_{320}

Figure 2.1: Typical ITS and Issue Structure and Example Data.

1 Some ITSs allow to add additional data fields. However, none of the projects that were researched in this thesis structures issues on this level.

2.1 DEFINITIONS

2.1.1 *ITS Data*

The data and metadata in an ITS is referred to as follows:

DEFINITION: ITS DATA FIELD

An ITS data field an collective term for an issues title, description and its comment(s).

DEFINITION: ITS METADATA

ITS metadata is an collective term for structured metadata related to an issue. An example is the author of the issue.

DEFINITION: DATA FIELD METADATA

Data field metadata is an collective term for structured metadata related to an ITS data field. An example is the author of a comment.

2.1.2 *Software Feature Request*

The *Shorter Oxford English Dictionary* (Trumble, 2007) defines

SOFTWARE as "the programs and other operating information used by a computer[...]" and

FEATURE as "a distinctive or characteristic part of a thing [software]; a part that arrests attention by its prominence etc. [...]"[2].

The Institute of Electrical and Electronics Engineers (IEEE) consequently merges these definitions and adds examples for such characteristics. IEEE specifies that a

SOFTWARE FEATURE is a "distinguishing characteristic of a software item (e.g., performance, portability, or functionality)" (ANSI/IEEE, 1998).

However, the above definition does not explain how an SF can be described; e.g. in terms of size or in terms of abstraction. Gorschek and Wohlin define SFs via their abstraction level

2 The term "feature" is also used in the Data Mining, Text Mining, and Machine Learning contexts (Domingos, 2012; Han, Kamber, and Pei, 2012; Hausser, 1999; Manning, Raghavan, and Schütze, 2008; Manning and Schütze, 1999; Witten, Frank, and Hall, 2011). It denotes "an individual measurable property of a phenomenon being observed" (Bishop, 2009). This thesis uses the term Software Feature (SF) to distinguish software functionality from such a measurable property.

FEATURE LEVEL REQUIREMENTS "should not offer details as to what functions are needed in order for the product to support a [software] feature; rather the requirements should be an abstract description of the [software] feature itself." (Gorschek and Wohlin, 2006)

In contrast the International Requirements Engineering Board e.V. (IREB) defines an SF with a higher level of abstraction: they say that a

SOFTWARE FEATURE is "a delimitable characteristic of a system that provides value for stakeholders. Normally comprises several requirements and is used for communicating with stakeholders on a higher level of abstraction and for expressing variable or optional characteristics." (Glinz, 2012)

It is noteworthy that the IREB definition adds the fact that an SF should provide value for stakeholders in contrast to the other definitions above.

In Part v, this thesis focuses on the detection of Software Feature Requests (SFRs). At the same time the different SF definitions above allow a broad range of formulations to request an SF with respect to abstraction level, wording and size. In the thesis a definition that does not restrict the size or abstraction level of an SFR is used, since multiple stakeholders tend to use different approaches to describe or request an SF[3]. Thus no restrictions with respect to size and abstraction level apply in the following definition of an SFR as in the definition of the IEEE. However, the value aspect of the IREB definition is added to distinguish an SFR from other requests that add value mainly to the developers. Examples for requests that do not provide a direct value for users of the software are refactoring or technical documentation tasks. Overall an SFR is defined as follows.

DEFINITION: SOFTWARE FEATURE REQUEST

A Software Feature Request (SFR) is text that requests for a distinguishing characteristic of a software item (e.g. a quality or functionality) that provides value for users of the software.

It should be noted, that SFs are often described implicitly as collection of software requirements or RA. Hence, many other well known publications do not explicitly define SFs. Examples are the popular textbook "*Software Engineering*" by Sommerville or the 2000 pages comprising "*Encyclopedia of Software Engineering*" by Marciniak from 2001.

3 This is discussed in detail in Part iv of this thesis.

2.2 MANAGEMENT WITH ISSUE TRACKING SYSTEMS

ITSs were invented to report and manage software bugs historically. However, issues can be used to track all kinds of work through information systems (Kunz and Rittel, 1970). Thus the functionality of ITSs increased early and they were used to manage and discuss multiple software development tasks.

Modern ITSs are known as project management systems. They allow to track additional data such as work time and they can be customized according to complex development workflows. A stakeholder authors an issue and categorizes whether this issue is a bug report, describes an SFR or another software development task (e.g. documentation or refactoring). This category is subsequently referred to as *issue type*.

DEFINITION: ISSUE TYPE

An issue type is an attribute that relate an issue to a disjoint set of issues. Examples for issue types are "feature", "bug", "refactoring", or "documentation".

The definition explicitly states a disjoint set of issues, since ITSs historically enforce assigning exactly one issue type per issue. Another approach is to tag issues with multiple issue types. This technique is referred to as *issue tags*.

DEFINITION: ISSUE TAGS

Issue tags are attributes that relate an issue to one or many (possibly intersecting) sets of issues. Examples for issue tags are "feature", "bug", "frontend", "database", or user names of users that are involved in solving the issue.

When the issue is submitted, the ITS automatically adds related ITS metadata. It assigns a unique ID, stores the author (or reporter) of the issue, and records the creation time. Then the issue is usually assigned to a software developer and is either implemented right away or further discussed (e.g. to elicit more information).

If the SFR is discussed further and users or developers add comments to the issue, the ITS records related data field metadata, such as the authors of the comments or the creation time of the comments. Finally, the issue is marked as "closed", which means that no more work with regard to the issue needs to be done. Occasionally, an issue is reopened in case it was closed before the work was completed.

Such an ITS workflow can be arbitrarily complex. An issue can be assigned to different groups like requirements engineers, or testers. It can have additional states like "rejected", if it will never be scheduled for development, or "duplicate", if it was already reported before.

Figure 2.2 shows a typical ITS workflow that is representative for the projects researched in this thesis[4].

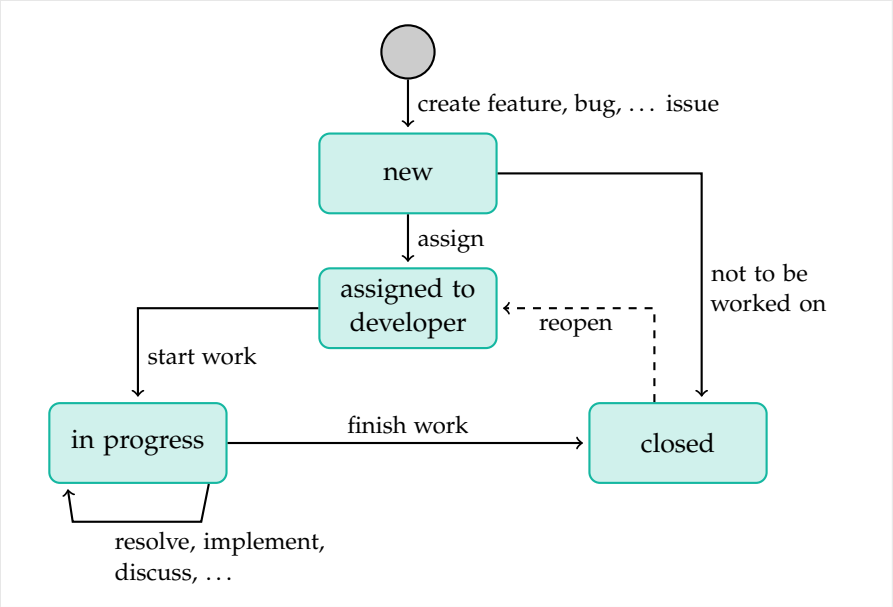


Figure 2.2: Typical Issue Tracking System Workflow.

ITS DATA FIELD	ISSUE #1910	ISSUE #12700
Title	Delete/close created forum entry	Let messages have a "solved" flag
Description	I suggest a feature under the forums where the user can close or delete the topic he/she started. This way, other users will not get confused if the problem is already resolved.	It would be easier to go through the messages in the forums if there was a "solved" flag users could set to show that their questions have been answered. • A filter could then be used to only show "open" messages. [...]
Comments	[none]	[none]

Figure 2.3: Excerpts of Two Issues From the Redmine Project.

Figure 2.3 shows the NL content of two example issues. Often, this NL text suffers from inconsistencies like typos or ambiguity (Ko and Chilana, 2011). Scacchi, 2009 note that SE artifacts in OSS projects are written in an informal way, which they call software informalisms: "in OSS development projects, software informalisms are the preferred

4 See Chapter 8 for the studied projects. Furthermore, the Jira ITS documentation provides additional examples for typical workflows: `https://confluence.atlassian.com/jira`.

scheme for describing or representing OSS requirements. There is no explicit objective or effort to treat these informalisms as 'informal software requirements' that should be refined into formal requirements within any of these communities". Thus SFRs are seldom refined or restructured in contrast to other RAs, which are often subject to specific quality assurances[5]. In addition the NL in ITSs carries a multitude of noise[6]. E.g. a bug report may contain:

1) NL to describe the bug,
2) a log file excerpt to support the bug description,
3) an NL question that asks when exactly the bug occurs,
4) an NL response from the user that includes,
5) a stack trace to show the actual impact on the code,
6) multiple NL comments to discuss a bug-fix including and/or
7) a code snippet (a patch) that fixes the bug.

In addition, two or more issues are often interconnected. Links between issues can usually be established by a simple domain-specific language. E.g. the text `#42` creates a trace link to an issue with ID 42 when written in an ITS data field. In some ITSs the semantic of such links can be specified (e.g. to distinguish duplicated from otherwise related issues). These semantically enriched links will be referred to as *trace types*.

DEFINITION: TRACE TYPE

A trace type is a link between two issues with a particular semantic meaning.

The issues in Figure 2.3 are marked as *related issues* in the according ITS of the Redmine project but they can be interpreted as duplicated issues judging from their content, too. Besides expressing a general relation or marking duplicated issues, there are more reasons to create links between issues. These reasons include but are not limited to:

- Expressing that a bug is related to a feature issue.
- Dividing (larger) issues in child-issues (e.g. for organizational purposes).
- Expressing that a certain issue blocks the resolution of another issue.

5 Natt och Dag and Gervasi, 2005 surveyed automated approaches to improve the NL quality of RAs.

6 NL and noise in issues is investigated in Part iv of this thesis.

Most ITS implementations store the same information in slightly different ways. Table 2.1 summarizes the data fields of four modern issue tracking systems. The information in Table 2.1 was derived from the data structures and user interfaces of the Bugzilla, Jira, Redmine, and GitHub ITSs[7], which are among the most commonly used ITSs according to the *The Eclipse Community Survey 2011* (Skerrett and The Eclipse Foundation, 2011). *Italic text* denotes that the data field contains user defined text. Besides different naming conventions, each of the data fields is employed in the default installation of all five ITSs. The first column of the table represents the naming conventions used in this thesis.

	ITS DATA FIELD NAME IN			
ITS DATA FIELD	BUGZILLA	JIRA	REDMINE	GITHUB
ID	bug ID	project-ID, ID	ID	ID
title	summary	summary	subject	title
description	description	description	description	comment
start timestamp	reported	created	start date	commented
close timestamp	closed	resolved	closed	closed
version	software version	software version	milestone	milestone
reporter	reported	reporter	added by	opened
assignee	assigned to	assignee	assignee	assignee
comments[1]	additional comment	comment	comment	comment
status changes[1]	modified	updated	updated by	various
file attachments[1]	attachment	attachment	files	embedded or as link
issue type	is handled via custom values, e.g. bug, feature, defect, enhancement ...			
issue status	is handled via custom values, e.g. new, accepted, assigned, closed ...			
scheduling	is handled via priority or severity fields in combination with milestones			
timestamps[1]	are recorded implicitly with each new entry, e.g. status change or comment creation			
usernames[1]	are recorded implicitly with each new entry, e.g. status change or comment creation			
custom fields[1]	all ITS include options to add custom fields			
trace types[1]	are stored in a database or plain text (e.g. the text "#7" references the issue with ID 7)			

[1] The plural form denotes that multiple instances can be associated to one issue.

Table 2.1: ITS Data Fields in Several ITSs.

7 See `https://www.bugzilla.org`, `https://www.atlassian.com/software/jira`, `https://github.com`, and `http://redmine.org`

3 TEXT PREPROCESSING FUNDAMENTALS

To work with raw text occurring in ITS data fields, proper preprocessing is important for IR as well as ML techniques, since "real-world databases are highly susceptible to noisy, missing, and inconsistent data" (Han, Kamber, and Pei, 2012, p. 83). Furthermore, "text mining [ML and IR] is arguably so dependent on the various preprocessing techniques [...] that one might even say text mining is to a degree defined by these elaborate preparatory techniques" (Feldman and Sanger, 2006, p. 57). At the same preprocessing techniques need to be chosen carefully: "[...] the appropriate preprocessing combinations depending on the domain and language may improve the accuracy considerably. In the meantime, inappropriate preprocessing combinations may degrade the accuracy significantly as well." (Uysal and Gunal, 2014)

Finally, preprocessing can largely reduce the processing time of most algorithms if the preprocessing technique comprises a data reduction (Manning, Raghavan, and Schütze, 2008).

3.1 DEFINITIONS

3.1.1 *Language Processing and Language Engineering*

In his paper "A definition and short history of Language Engineering" Cunningham, 1999 amply discusses the relatedness of three fields: (1) Natural Language Processing (NLP), (2) Language Engineering (LE), and (3) Computational Linguistics (CL). His discussion is based on a multitude of dictionaries and scientific publications. Cunningham states that:

COMPUTATIONAL LINGUISTICS is "that part of the science of human language that uses computers to aid observation of, or experiment with, language."

NATURAL LANGUAGE PROCESSING is "a term used in a variety of ways in different contexts [...] NLP is a branch of computer science that studies computer systems for processing natural languages. It includes the development of algorithms for parsing, generation, and acquisition of linguistic knowledge; the investigation of the time and space complexity of such algorithms; the design of computationally useful formal languages [...] for encoding linguistic knowledge [...]" and so on. Finally,

LANGUAGE ENGINEERING is "the discipline or act of engineering software systems that perform tasks involving processing human language. Both the construction process and its outputs are measurable and predictable. The literature of the field relates to both application of relevant scientific results and a body of practice."

From these statements it can be concluded that CL focuses on understanding language with the aid of computers, NLP focuses on the study of computers and algorithms with respect to the processing of languages, and LE focuses on the engineering part: e.g. the application of (NLP based) methods to achieve a certain goal. In this thesis only LE methods based on rules are discussed. To distinguish these from ML and IR based methods as used in Part V, LE is defined as follows:

DEFINITION: LANUGAGE ENGINEERING

Language Engineering is the creation of rule based methods to process or classify natural language.

Rule-based methods are used regularly in LE or as preprocessing methods for NLP tasks. For example to tag parts of speech (Brill, 1992)[1], to extract information such as names or places (Cunningham et al., 2016, pp. 113), or, to a certain extend, even to detect software requirements (Vlas and Robinson, 2013).

3.1.2 *Documents and Corpora*

Feldman and Sanger, 2006 state that "a document can be very informally defined as a unit of discrete textual data within a collection that usually, but not necessarily, correlates with some real-world document such as a business report, legal memorandum, e-mail, research paper, manuscript, article, press release, or news story". They use the term "document collection", which is often called a *corpus*, too. Documents considered in this thesis are either issues extracted form ITSs or emails extracted from development mailing lists. Hence, corpus and document is referred to as follows.

DEFINITION: CORPUS

A corpus is a dataset of issues extracted from one or more ITSs or a dataset of emails extracted from one or more mailing list archives.

1 Part of speech tagging is introduced in detail in the paragraph "Grammatical Tagging".

DEFINITION: DOCUMENT

A document is a unit of discrete textual data including metadata within a corpus (e.g. an issue or an email).

3.1.3 *Preprocessing*

Data preprocessing is often defined by its goals (Han, Kamber, and Pei, 2012, p. 83):

DATA CLEANING can be applied to remove noise and correct inconsistencies in data.

DATA REDUCTION can reduce data size by, for instance, aggregating or clustering.

DATA TRANSFORMATIONS (e.g., normalization) may be applied, where data are scaled to fall within a smaller range like 0.0 to 1.0.

DATA INTEGRATION merges data from multiple sources into a coherent data store such as a data warehouse.

The task of data integration is necessary whenever data is processed form different sources such as different ITSs and/or mailing list. The data integration process is described whenever data is extracted from more than one source. However, in this thesis the term preprocessing is not used for data integration, since the data integration part does not influence the experiment results as much as other preprocessing techniques. The preprocessing steps that have the highest impact on the experiments are *data cleaning* and *data reduction*[2]. Hence preprocessing will be used as follows.

DEFINITION: PREPROCESSING

Preprocessing is the uniform transformation of all documents in a corpus to remove noise (clean the data) or to reduce the overall amount of data.

DEFINITION: NOISE

Noise is any unwanted data within a corpus with respect to a particular data processing task.

Intentionally, this definition does not denominate "unwanted" as it depends on the data processing task. For example in Part V non-NL artifacts are removed in a preprocessing step. However, this does not always have positive impacts on the results and thus non-NL artifacts are cannot be considered noise per se.

2 Data transformation is an important step since IR and ML algorithms usually work with numerical data. However, this transformation is handled implicitly by the applied approaches and will not be elaborated.

3.2 PREPROCESSING TECHNIQUES

The following preprocessing techniques are suggested by text books (Feldman and Sanger, 2006, pp. 57; Han, Kamber, and Pei, 2012, pp. 84; Manning and Schütze, 1999, pp. 124; Witten, Frank, and Hall, 2011, pp. 349) as well as by related tool documentation (Bird, Klein, and Loper, 2009, pp. 87; Cunningham et al., 2016, pp. 119) and are introduced in order of their usual processing sequence.

LEXICAL ANALYSIS AND LOWERCASING To make text computer processable, the first step is to divide the text into smaller units called tokens. This process is known as lexical analysis or tokenization. A token can either be a word, a sentence, a punctuation sign, a number, or any unit of text. In the following sentence and word segmentation are distinguished. It is, however, notable that "the question of what counts as a word [or a sentence] is a vexed one in linguistics, and often linguists end up suggesting that there are words [or sentences] at various levels [...] which need not to be the same" (Manning and Schütze, 1999, p. 125). In contrast to tokenization, lowercasing is a simple process. It equalizes different words by converting every character to lowercase.

Example text before tokenization:

This is an example for text preprocessing.
It is very basic. See `http://www.nltk.org` for implementations.

Example sentences after sentence tokenization:

This is an example for text preprocessing.

It is very basic. | See `http://www.nltk.org` for implementations.

Example sentences after word tokenization and lowercasing:

this | is | an | example | for | text | preprocessing | . | \n

It | is | very | basic. | . | see | http://www.nltk.org | for | implementations | .

Figure 3.1: Sentence and Word Tokenization Examples.

Figure 3.1 shows an example for sentence and word tokenization and lowercasing, as it is applied in the experiments in Parts iv and v. Sentence tokenization delimits text in sentences whenever it finds dots followed by a whitespace character, dots followed by a line break, or multiple line breaks. Word tokenization creates a new token at every occurrence of whitespace. It also separates interpunctuation characters followed by a whitespace character (colored ◯ in Figure 3.1). This way information such as an Uniform Resource Locator (URL) is preserved as one token, but other interpunctuation is separated. On the contrary, this procedure has the disadvantage that sentences or

words separated by interpunctuation without a whitespace character are not divided correctly[3]. The latter exemplifies the statement of a "vexed question" by Manning and Schütze.

GRAMMATICAL TAGGING AND PARSING Grammatical Tagging, also called part of speech tagging, is the process of determining which grammatical form a token has. Usually the grammatical form of each word in a sentence is represented by the annotation system of the Penn Treebank Project (Santorini, 1990). Penn Treebank tags are a two or three letter encodings, each of which represents a grammatical form. Many different techniques for grammatical tagging exist. Basic techniques use a rules or a grammar to determine the grammatical form of a word (Brill, 1992). More advanced techniques use knowledge derived from statistical models[4]. One of the most popular approaches is the Stanford Tagger (Toutanova et al., 2003).

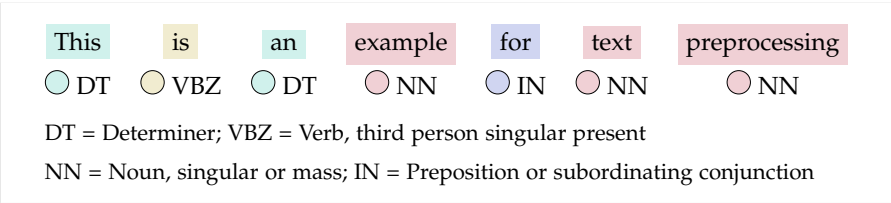


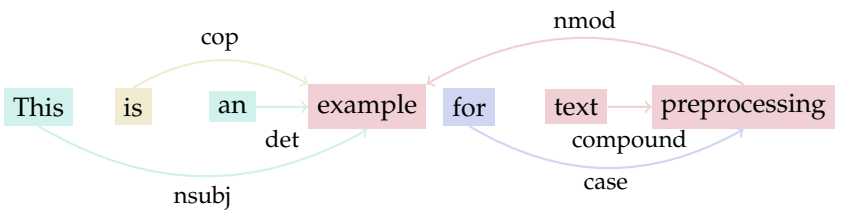
Figure 3.2: Grammatical Tagging Example.

The approach by Chen and Manning, 2014 detects the references between words in addition to grammatical tagging. This is called grammatical parsing. Besides an implementation of the approach itself, Chen and Manning provide a trained instance of this implementation, that can be used on English texts. The output of this trained instance is shown in Figure 3.3. Both the implementation and the trained instance are utilized in this thesis.

STEMMING AND LEMMATIZATION Stemming as well as lemmatization are techniques to reduce words to a common form. Stemming reduces words to their word stem (Hull, 1996): e.g. the words "developer", "developing", and "development" are all reduced to "develop". Although stemming is appropriate to equalize most word forms in the English language, it cannot equalize all forms. E.g. the words "be", "was" and "were" do not have the same stem although they originate from the irregular verb "to be". Lemmatization employs various techniques to find the infinitive form of a verb (Hausser, 1999, pp. 258). However, in this thesis stemming is applied as (1) the

3 More advanced tokenization methods use statistical models, e.g. NLTK's Punkt Sentence Tokenizer (see `http://www.nltk.org`). However, such techniques need to be trained on a large number of similar documents to work efficiently.

4 E.g. by Hidden Markow Models, Maximum Entropy or Conditional Markow Models, Chapter 5 describes the operating mode of such statistical models in more detail.



case	The case relation is used for any preposition in English.
compound	The compound relation is used for noun compounds.
cop	A copula is the relation between the complement of a copular verb and the copular verb. Copular heads are avoided when possible.
det	A determiner is the relation between the head of an noun phrase and its determiner.
nmod	The nmod relation is used for nominal modifiers of nouns or clausal predicates.
nsubj	The nsubj relation is used for a nominal subject. This is a noun phrase which is the syntactic subject of a clause.

Abbreviations and descriptions as in Nivre et al., 2015.

Figure 3.3: Grammatical Parsing Example.

computational overhead of lemmatization techniques is significant and (2) different times are seldom used in SFRs (Merten et al., 2015). In this thesis a slightly enhanced implementation of the Porter algorithm (Porter, 1980) is used as provided by the NLTK (Bird, Klein, and Loper, 2009). After stemming the example from Figure 3.1 yields

> this is an exampl for text preprocessing. it is veri basic. see http://www.nltk.org for detail.

STOP WORD REMOVAL According to Feldman and Sanger, 2006, p. 68 stopwords are "the function words and in general the common words of the language that usually do not contribute to the semantics of the documents and have no real added value". However, there is no definition or agreed upon list of English stop words. Hence, the stopword list from the Natural Language Toolkit (NLTK) (Bird, Klein, and Loper, 2009) is employed in this thesis as it has proven serviceable in many applications. After stopword removal the example from Figure 3.1 yields

> example text preprocessing. basic. See http://www.nltk.org implementations.[5]

5 As stop word removal destroys the sentence structure, it is important not to apply stop word removal before grammatical tagging or parsing.

GENERAL CLEANUP In addition to the techniques presented above, other cleanup tasks are often applied. For example hyperlinks or other non-NL artifacts can be removed, as they are often considered noise. Words that are expressed in `CamelCase` or named according to the `under_score` convention are often separated in the context of ITS data (Borg, Runeson, and Ardö, 2014). Finally, issues often contain a mix of technical data like source code, log files, and stack traces and NL. Chapter 14 presents a preprocessing technique to separate NL from technical information. The details of cleanup tasks are described whenever they were applied in this thesis. In general, preprocessing tasks largely depend on the input corpora and the data processing tasks and can therefore not be considered beneficial per se. Hence most experiments in this thesis are conducted with different preprocessing techniques to show the influence of the techniques on the data processing task. Finally, multiple preprocessing techniques are often concatenated in order to improve the data or reduce the dataset. E.g. Gervasi and Zowghi, 2014 consider only nouns, adjectives, adverbs, and verbs after grammatical tagging for further processing in their experiment.

4 INFORMATION RETRIEVAL FUNDAMENTALS

The goal of IR is "to retrieve all the documents that are relevant to a user query while retrieving as few non-relevant documents as possible" (Baeza-Yates and Ribeiro-Neto, 2011, p. 4). Usually this is achieved by computing the textual similarity of a query q to a document d in the corpus:

$$\textit{similarity}(q, d) \rightarrow R, \text{ with } R = \{r \in \mathbb{R} | 0 \leqslant r \leqslant 1\}. \tag{4.1}$$

If, however, q is replaced by another document d' IR techniques compute the similarity between the two documents d and d', which is the way IR is applied in this thesis.

4.1 DEFINITIONS

4.1.1 *Term Frequencies*

DEFINITION: TERM FREQUENCY

If w is a word and d is a document, then the number of occurrences of w in d is

$$\text{TermFreq}(w, d), \tag{4.2}$$

or the term frequency of w in d.

To normalize the term frequency within a corpus, the Term Frequency - Inverse Document Frequency (TF-IDF) can be computed.

DEFINITION: TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY

TF-IDF is defined as

$$\text{TF-IDF}(w, d) = \text{TermFreq}(w, d) \cdot \log\left(\frac{n}{\text{DocFreq}(w)}\right), \tag{4.3}$$

where n is the number of documents and $\text{DocFreq}(w)$ the number of documents that contain the word w.

4.1.2 *Traceability Matrix*

A link between two software artifacts[1] is also called a trace link. Traceability is the field of creating and maintaining such trace links. Usually one differentiates between horizontal and vertical traceability (Pohl, 2010) as well as pre- and post-traceability (Gotel and Finkelstein, 1994). In this thesis the focus is on horizontal and vertical post-traceability: e.g. the trace links between different issues (horizontal post-traceability) during all SE phases (vertical post-traceability) without focusing on why and by whom these issues were created (pre-traceability).

	I_1	I_2	I_3	I_4	I_5
I_1	-	✓			
I_2		-			
I_3			-		
I_4	✓			-	
I_5			✓		-

Table 4.1: Example Traceability Matrix.

Trace links between issues can be expressed in a so called traceability matrix. A traceability matrix represents all trace links between a set of issues. Table 4.1 shows an example for a traceability matrix. It is read from column to rows and shows trace links from issue I_1 to I_4, I_2 to I_1, and I_3 to I_5. The matrix in Table 4.1 is directed (from issue I_a to I_b, but not necessarily from I_b to I_a). An undirected traceability matrix is simply mirrored at its diagonal (indicated by "-" in Table 4.1).

4.1.3 *Trace Retrieval*

Whenever trace links are not created explicitly, it is often desired to retrieve trace links automatically or semi-automatically from given artifacts. This process is called *trace retrieval*[2]. In the context of this thesis, automatically retrieved trace links by IR algorithms without human intervention are considered[3]. Trace retrieval is defined as follows.

1 A "software artifact is any piece of information, a final or intermediate work product, which is produced and maintained during software development" (Kruchten, 2003).
2 Sometimes referred to as trace recovery.
3 However, other approaches exist, e.g. to find duplicate crash reports (Dang et al., 2012). Such approaches make use of specific structured data, which is not present in the context of the researched projects.

DEFINITION: TRACE RETRIEVAL

Trace retrieval is the automatic generation of trace links between issues.

4.2 INFORMATION RETRIEVAL TECHNIQUES

Equation 4.1 on page 29 can be used to retrieve trace links when combined with a threshold: if I is the set of all issues, the similarity (S) of two issues is computed by

$$\textit{similarity:}\ I \times I \rightarrow R, \text{ with } R = \{r \in \mathbb{R} | 0 \leqslant r \leqslant 1\}. \tag{4.4}$$

A trace link has only two states: it is either present or it is not. In contrast, S is a real number. Hence, a trace link is created only if the similarity exceeds a certain threshold t, so that $\textit{trace}_t : I \times I \rightarrow \{\mathtt{true}, \mathtt{false}\}$ with

$$\textit{trace}_t(i, i') = \begin{cases} \mathtt{true} & : \textit{similarity}(i, i') \geqslant t \\ \mathtt{false} & : \textit{similarity}(i, i') < t \end{cases} \tag{4.5}$$

defines whether a trace between issue i and i' exists. A traceability matrix of size $|I| \times |I|$ with elements $a_{i,j} = \mathtt{trace}_t(i, j)$ can now be created by computing the similarity between all issues in the corpus. For a corpus with n issues this is computed in $O(n^2)$. The threshold t, is usually derived empirically by comparing the results of $\textit{trace}_t$ with different t to a ground truth.

As IR algorithms compute the text similarity between documents, spelling errors, hastily written notes that leave out information, or noise have a negative impact on the algorithm performance. In addition, the performance is influenced by source code as well as stack traces, which often contain a considerable amount of the same terms (e.g. Java package names). Therefore, an algorithm might compute a high similarity between two issues that refer to different topics if they both contain a similar stack trace.

In this thesis the following IR algorithms were used to calculate a traceability using Equation 4.5 with different thresholds t:

VECTOR SPACE MODEL (VSM) The VSM maps the tokens of a document to vectors. By using a distance metric such as TF-IDF, the similarity of two document can be computed. One of the main problems in VSM is exactly this dependency on each term and each term's spelling. For example VSM does not consider synonyms in its computation. Therefore, the VSM approach may compute a high similarity between issues with equal terms that may have different meanings due to their context (Salton, Wong, and Yang, 1975).

LATENT SEMANTIC ANALYSIS (LSA) The LSA copes with the mentioned problems of VSM. Instead of computing S between terms, LSA computes S between concepts of documents. Concepts are an abstraction of multiple terms and represent the "topics" of a document. LSA creates those concepts using singular value decomposition (Baeza-Yates and Ribeiro-Neto, 2011, p. 101) which also reduces the search space. In this thesis LSA is applied using the cosine measure as a distance measure (Furnas et al., 1988).

OKAPI BEST MATCH 25 (BM25) In contrast to the above, BM25 is a probabilistic approach to calculate S. It relies on the assumption that an ideal set of related documents can be found and computes the probability of each other document to be in this set. The BM25 extensions BM25L and BM25+ both try to compensate problematic behavior of BM25 on long documents (Lv and Zhai, 2011b). In this thesis the acronym BM25 is used for all three algorithms: the original BM25 (Robertson et al., 1992) as well as its variants BM25L (Lv and Zhai, 2011b), and BM25+ (Lv and Zhai, 2011a).

In summary all approaches depend on the following properties of an issue i in the corpus of issues I:

- the actual terms in i compared to another issue i',
- the number of terms (term frequency) in i, and
- the number of terms in i that and how often they occur in I (TF-IDF).

These properties are influenced by text preprocessing (as presented in Section 3.2) and due to this influence it cannot be determined how well algorithms perform on a certain corpus without experimenting. However, BM25 is often used as a baseline to evaluate the performance of new algorithms for classic IR applications such as search engines (Baeza-Yates and Ribeiro-Neto, 2011, p. 107). More information and details on algorithms, their advantages, and their disadvantages can be found in the IR literature (Baeza-Yates and Ribeiro-Neto, 2011; Manning, Raghavan, and Schütze, 2008).

5 TEXT AND DATA MINING FUNDAMENTALS

"Data mining is the process of discovering interesting patterns and knowledge from large amounts of data"
- Han, Kamber, and Pei, 2012

In general "data mining is the process of discovering interesting patterns and knowledge from large amounts of data" (Han, Kamber, and Pei, 2012, p. 8). Implicitly, this statement assumes that those large amounts of data are structured, e.g. in data bases or data warehouses. In the context of this thesis the discovery of patterns that represent an SFRs is of interest. In ITSs SFRs are described using NL and the only structured information can be found in the ITS metadata. Hence, the ITS metadata alone is not sufficient to discover patterns that represent an SFR. Instead the ITS data fields need to be considered to search for patterns, which is where text mining comes into play. "In a manner analogous to data mining, text mining seeks to extract useful information [...]. In the case of text mining, however, the data sources are document collections, and interesting patterns are found [...] in the unstructured textual data in the documents in these collections" (Feldman and Sanger, 2006, p.1).

The border between data and text mining is very thin. In this thesis text mining is applied in general, but the ITS meta data is considered too, shifting towards the more general area of data mining. However, data mining and text mining share the same workflow (Feldman and Sanger, 2006; Han, Kamber, and Pei, 2012, cf.):

- Data needs to be integrated and selected (e.g. where to look for patterns).
- Data needs to be cleaned (e.g. data preprocessing).
- Patterns need to be found (e.g. the actual process of data or text mining).
- Results need to be represented.

5.1 DEFINITIONS

In terms of the application of data- and text mining, this thesis is about data classification. To be exact, a piece of data (e.g. an ITS data field) is classified whether is represents an SFR or not. Such a classification is implemented by supervised ML algorithms in both data and text mining. Supervised means that the ML algorithm needs to learn before it can be applied (Han, Kamber, and Pei, 2012; Witten, Frank, and Hall, 2011). Similar to human learning, a ML algorithm learns from examples (also called the training data), but in contrast to human intuition an ML algorithm calculates a statistical model that

classifies a piece of data. This statistical model is also called a prediction model, or simply 'model' in Data Mining (DM) literature (Han, Kamber, and Pei, 2012; Witten, Frank, and Hall, 2011).

DEFINITION: MACHINE LEARNING ALGORITHM

A ML algorithm refers to the implementation of a supervised method to generate a prediction model for classification.

DEFINITION: CLASS

A class is a category that should be predicted by an ML algorithm, e. g. whether an issue is a "Feature" or "Bug".

DEFINITION: TRAINING DATA

Training data are the example documents in a corpus $T \in C$ that are used to train an ML algorithm.

DEFINITION: TEST DATA

Test data are all other documents $d \in C, d \notin T$ that are used to evaluate an ML model.

DEFINITION: CLASSIFIER/MODEL

A classifier, also known as a prediction model, ML model, or simply model is a synonym for the statistical prediction model that is generated by training an ML algorithm. In other words a classifier is a trained ML algorithm that can assign classes to documents.

DEFINITION: MACHINE LEARNING FEATURE

An MLF is a property derived from the ITS meta-data or the NL text in an ITS data field that is used for ML.

In general, a prediction model is more precise, the more training data is provided. In addition to the amount of training data, the selection of appropriate MLFs[1] is very important (Domingos, 2012; Guyon and Elisseeff, 2003). For example, an ML algorithm can be trained to predict whether an issue should be classified as *feature* or *bug* using *all the words that occur in the issue* as MLF. In this case *feature* and *bug* are the classes and *all the words that occur in the issue* is the MLF. This par-

1 In ML objects are usually represented with n-dimensional vectors of numerical values, called *feature vectors*. A single value within these feature vectors is called a feature. Since this thesis heavily relies on the distinction of Software Feature (SF) and Machine Learning Feature (MLF) the term MLF is used for Machine Learning Feature (MLF).

ticular MLF is also called *Bag of Words (BOW)*[2]. Now assume that the ML algorithm from the example above is not trained with the BOW, but with the number of comments of each issue. It would be difficult for the ML algorithm to compute a precise prediction model (unless every *bug* issue has more comments than a *feature* issue or vice versa). However, the combination of both BOW and number of comments may improve the results.

In the above examples two types of MLFs are discussed. Namely, binary and ordinal MLFs. The BOW is a binary MLF as a word has only two states: It can be present in the bag or not. The number of comments of an issue is an ordinal MLF as the number is represented within an ordered series (i.e. the number of comments). Finally, there are nominal features with no inherent order or sequence. An example for the latter is the issue type. An issue type can be bug, feature, or refactoring but the order is undefined. Nominal features can be transformed into binary or ordinal features by discretization. For example one can divide the number of comments in an issue in three classes: issues with short, medium, or long discussions. The classifiers used in this thesis are trained with binary and ordinal features, only. Accordingly, discretization is applied to nominal features.

5.2 TEXT AND DATA MINING TECHNIQUES

Different algorithms can be employed for classification tasks. In this thesis the following seven algorithms are used to detect SFRs, being among the most heavily used algorithms in classification tasks (Han, Kamber, and Pei, 2012; Pedregosa et al., 2011; Witten, Frank, and Hall, 2011):

NAIVE BAYES CLASSIFIER (NB) Studies that compare classification algorithms have found the naive bayes classifier to be comparable in performance with most of the classifiers described next as well as selected neural network classifiers (Han, Kamber, and Pei, 2012, p. 350). Also, Bayesian classifiers are known to be accurately and fast when applied to large databases. NB is traditionally used to distinguish desired emails from spam emails.

MULTINOMIAL NAÍVE BAYES CLASSIFIER (MNB) MNB is, similar to NB, very popular for text classification tasks, especially to learn models based on the frequencies of words (Witten, Frank, and Hall, 2011, p. 97).

LINEAR SUPPORT VECTOR CLASSIFIER (SVM) Linear support vector classifiers, also called SVMs, are known to be highly accurate. In contrast to the above, SVM select boundary instances of each

2 More MLFs are introduced in Chapter 15.

class called support vectors. Then they build a linear discriminant function that separates those support vectors and transitively the classes. As SVMs are often used with higher-ordered MLF sets, this function is usually a hyperplane in an appropriate space. Hence training an SVM is usually processing intensive.

LOGISTIC REGRESSION (LR) The LR model is used in many text classification tasks, too. LR is a modified regression technique so that it can better be applied to classification tasks and produces a proper output for the classified instances to either belong to the class (1) or not (0). Usually regression techniques deliver continuous values[3]. However, as LR needs to perform probability estimations to predict the class labels, it is an computationally intense approach.

STOCHASTIC GRADIENT DESCENT (SGD) The SGD learner uses a linear model to classify. SGD has been around in the ML community for quite a while, but it had a revival in 2004 when it was used for large scale classification tasks (Zhang, 2004). As text classification with many different words can become a large scale problem rather quickly, SGD is very popular for text classification, too.

DECISION TREE (DT) The DT can be defined as "a flowchart-like tree structure, where each internal node (non-leaf node) denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (or terminal node) holds a class label." (Han, Kamber, and Pei, 2012, p. 330) Hence the classification decisions made by a DT are comprehensible and, most importantly, traceable. This makes DTs very popular in classification tasks.

RANDOM FOREST (RF) The RF classifier learns multiple DTs from randomized sub-samples of the dataset and averages[4] the results.

In all experiments the default settings for the above classifiers, as provided by the NLTK (Bird, Klein, and Loper, 2009) and the Scikit-learn (Pedregosa et al., 2011) Application Programming Interfaces (APIs), are used. More details on ML algorithms can be found in Bird, Klein, and Loper, 2009; Bishop, 2009; Han, Kamber, and Pei, 2012; Witten, Frank, and Hall, 2011.

5.3 MACHINE LEARNING TECHNIQUES AND ITS DATA

Issues contain data fields, data fields contain NL text, and NL text is composed of sentences[5]. Hence, the detection of SFRs can be ap-

3 Witten, Frank, and Hall, 2011, pp. 125 give a detailed explanation of the advantages and disadvantages of using regression techniques in classification tasks.

4 Some RF implementations use another DT for their final decision.

5 Sentence is used as synonym for phrases, bullet points, and so on.

proached on different levels of detail. Detection on *issue* level reveals, whether the issue contains at least one SFR. Detection on *data field* level reveals, whether the title, the description, or a certain comment to the issue contains at least one SFs. Detection on *sentence* level reveals, whether a certain sentence in the NL describes an SFR. In figures and tables, these levels will be referred to as *I*, *DF*, and *Se*.

The level of detail influences the SFR detection rate. In general terms, the higher the level of detail, the harder is a detection due to the following circumstances:

- On a lower level, more MLFs can be employed to train the model. E.g. on issue level, the words in the title, description and comments can be used to create the BOW MLF. In contrast, on sentence level only the words of that sentence form the BOW.
- More objects need to be classified on a higher level with the same amount of training data. E.g. if a dataset includes > 200 SFR annotations. These are used for a detection within a low number of issues and a high number of sentences, respectively[6].

For every level, some MLFs can be derived directly (e.g. issue meta-data MLFs on issue level) without the need for further transformation. To gather the MLFs from other levels, the concepts of *inheritance* and *aggregation* are employed. E.g. if the detection is done on sentence level, every sentence can inherit the meta-data MLFs from the comprising data field and issue. If the detection is done on issue level, every issue accumulates the MLFs from its embedded data fields and, transitively, the MLFs derived from their textual contents. On data field level both concepts are used. This allows to compare the same combinations of MLFs on different levels and thus to evaluate which detection level yields the best results.

6 E.g. 599 issues compared to 11149 sentences in the evaluation in Chapter 15.

6 MEASUREMENT FUNDAMENTALS

For IR and classification tasks the performance of an approach needs to be evaluated. In IR the goal is to measure whether relevant documents were retrieved and in classification the task is to evaluate how good the classification is done.

For both tasks the measures of precision and recall can be used. However, the exact definition of those measures depends on their context of use. This leads to slightly different definitions in IR and ML.

6.1 PRECISION AND RECALL IN INFORMATION RETRIEVAL

In IR, automated approaches are evaluated with respect to a ground truth, which often needs to be created manually. In the context of trace retrieval, automated approaches create a trace matrix. The results are evaluated with respect to a reference trace matrix. Recall (R) measures the retrieved relevant links and Precision (P) the correctly retrieved links:

$$\text{Recall} = \frac{\text{Correct Links} \cap \text{Retrieved Links}}{\text{Correct Links}} \tag{6.1}$$

$$\text{Precision} = \frac{\text{Correct Links} \cap \text{Retrieved Links}}{\text{Retrieved Links}} \tag{6.2}$$

Huffman Hayes, Dekhtyar, and Sundaram, 2006 define *acceptable*, *good* and *excellent* P and R ranges for the task of trace retrieval as shown in Table 6.1.

ACCEPTABLE	GOOD	EXCELLENT
$0.6 \leqslant R < 0.7$	$0.7 \leqslant R < 0.8$	$R \geqslant 0.8$
$0.2 \leqslant P < 0.3$	$0.3 \leqslant P < 0.4$	$P \geqslant 0.4$

Table 6.1: Trace Retrieval Evaluation Measures as in Huffman Hayes, Dekhtyar, and Sundaram, 2006.

6.2 PRECISION AND RECALL IN CLASSIFICATION

As in IR, ML performance is typically measured by comparing the results of a classifier to a manually created ground truth.

In the case of SFR detection the algorithm determines whether a piece of data (e.g. an ITS data field) corresponds a class or not (e.g. either the ITS data field contains an SFR or not). Such a classification can either be correct or not. Thus, classification results end up in four different measures of relevance:

1. The piece of data is correctly classified to belong to the class. It is a True Positive (TP).
2. The piece of data is correctly identified not to belong to the class. It is True Negative (TN).
3. The piece of data is wrongly classified to belong to the class. It is a False Positive (FP).
4. The piece of data is wrongly identified not to belong to the class. It is a False Negative (FN).

Based on these four measures of relevance, R and P are defined as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{6.3}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{6.4}$$

In simple terms, it can be summarized that R measures how many relevant SFRs are found and P measures how many relevant SFRs are found correctly in the context of SFR detection.

6.3 THE F MEASURE

The following problem is inherent to the precision and recall measures: R can be maximized by retrieving all documents in IR or by classifying all documents as the desired class in ML. This results in a low P in both cases. On the other hand P can be maximized by retrieving only one correct document in IR or by classifying only one document correctly in ML, which results in a low R in both cases. This problem is picked up by the F_β-Measure, as the harmonic mean of P and R. The F_β-Measure balances P and R so that both, a low precision or a low recall, results in a lower F_β-Measure. The β weights precision and recall, where $\beta < 1$ favors precision and $\beta > 1$ favors recall. Hence a general range of good or bad F_β-Measures cannot be given and depends largely on the use case[1].

1 The importance of recall in favor of precision is discussed in the next section.

In the thesis P, R and the F_1 and F_2 measures are reported. The F_1 measure balances P and R equally and the F_2 measure gives a higher emphasis on recall.

$$F_\beta = \frac{(1+\beta^2) \times P \times R}{(\beta^2 \times P) + R}, \text{ where } \beta^2 = \frac{1-\alpha}{\alpha} \tag{6.5}$$

with $\alpha \in [0, 1]$ and thus $\beta \in [0, \infty]$ (adapted from Manning, Raghavan, and Schütze, 2008, p. 156).

6.4 THE MAXRP MEASURE

In addition to the above another measure is defined for this thesis.

DEFINITION: MAXRP MEASURE

$$\text{MAX}(R), P_{\geqslant p}, \text{ with } 0 \leqslant p \leqslant 1 \tag{6.6}$$

is the best achievable recall with a precision higher than p%.

It is often desired to maximize recall in favor of precision as argued e.g. by Berry et al., 2012. The $\text{MAX}(R), P_{\geqslant p}$ measure satisfies this by defining the minimum viable precision to be at least p and measures the best achievable recall, which should ideally be close to 1.

The following example illustrates the break even point of the precision in the $\text{MAX}(R), P_{\geqslant p}$ measure with respect to manual classification. It uses a dataset size as in the related study in Chapter 15: with 76 true positives in a dataset, a $\text{MAX}(R), P_{\geqslant 0.05}$ yields $76 \times 20 = 1,444$ hits assuming the worst allowed precision of 5%. This implies that $1,444 - 76 = 1,364$ false positives have to be winnowed manually, to correctly identify the true positives excluding any false positives. Or differently put: this is the same as boosting the precision to 1 manually. The additional manual work is amortized, as soon as the number of objects to be classified is $> 1,364$ for 76 true positives.

In the dataset in Chapter 15 this is the case on the data field level, which consists of 599 titles +599 descriptions +3519 comments = 4717 data fields, or on the sentence level, which consists of 11149 sentences, but not on the issue level, which consists of 599 issues. $\text{MAX}(R), P_{\geqslant p}$ will be reported for $p = 0.05$ on the sentence and data field levels, and $p = 0.2$ on the issue level. With these settings for p sound values for R can be achieved in Chapter 15. If p, is e.g. doubled ($p \in \{10, 20\}$), the results for R become significantly worse.

Part III

GENERAL STUDY SETUP

In this part the overall study setup is described. First, the data used in the experiments and how it was acquired is explained. Then the overall research methodology used in all experiments is presented. Finally, the tools and software support that represent the foundation of the experiments are introduced.

7 RESEARCH METHODOLOGY

"When we undertake a large project we can easily be overwhelmed by the enormity of the task. We will be unsure about what to do first. A good understanding of the ideal process will help us to know how to proceed." – Parnas and Clements, 1986.

To select a research method the types of RQs should be understood (Easterbrook et al., 2008). At this point the overall RQs for this thesis as stated in Section 1.3 are revised by their type to identify appropriate research methods. According to Easterbrook et al., 2008 those research questions can be categorized as follows:

RQ 1 is a mixture of an existence questions of the form, "Does X exist?" as well as an descriptive-comparative questions of the form, "How does X differ from Y?":
What information about SFs can be found in an ITS of a software product, and how well is this information suited to derive a feature representation of the software?

RQ 2 is an description and classification questions such as, "What is X like?":
How is NL information categorized, described, and distributed in an ITS?

RQ 3 is a design questions of the form, "What is an effective way to achieve X?":
Can SFs descriptions be detected automatically in ITS NL data?

RQ 4 is a mixture of a causality question of the form, "What effect does X have on Y?" and a design question[1]:
Do trace retrieval algorithms perform effectively on ITS data?

All of these questions can be investigated using case study research. It has often been argued that case studies are a dominant and effective tool in software engineering research (Demeyer, 2011). A case study is defined as "an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between the phenomenon and context are not clearly evident" (Yin, 2013, p. 13). Research distinguishes between exploratory and confirmatory case studies where the former is an initial investigation of some phenomena to derive new hypotheses, and the latter is used to test existing theories (Easterbrook et al., 2008; Kitchenham et al., 2002; Wohlin et al., 2012). Wieringa, 2014 too stresses that there are two types of research problems in his work on design science. He names these problems knowledge problems and design problems respectively.

1 In addition to an empirical evaluation of existing IR methods on issues, improvements (e.g. new designs) to these methods are evaluated.

In this thesis five studies are presented. Two of these studies, presented in Part iv, are exploratory case studies that will be referred to as *investigation studies* as they investigate different parts of the SFR detection problem. In Part v ITSoFD, a solution design for SF detection, is presented and then validated in three studies. These studies will be referred to as *solution studies*. In summary this thesis distinguishes between:

INVESTIGATION STUDIES Investigation studies manually investigate the data and derive an answer to predefined RQs. These studies are implemented in the thesis as empirical case studies on written artifacts of OSS SE projects. The thesis contains two investigation studies in Chapters 10 and 11.

SOLUTION STUDIES Solution studies implement and validate a solution that fits a previously defined goal (e.g. a design). In this thesis designs are automated approaches that solve problems on OSS ITS data and every design is validated by measuring the automatically created results with respect to a manually created ground truth. The thesis evaluates the three main components of the ITSoFD solution design, which is described in Chapter 13 and in the three solution studies in Chapters 14, 15, and 16.

As for any empirical research, the work presented in this thesis has limitations to its internal and external validity. Although known threats were considered and mitigated throughout the experiments, a perfect validity can never be guaranteed. Threats to validity are discussed as proposed by Runeson and Höst, 2009, Wohlin et al., 2012, or Yin, 2013:

CONSTRUCT VALIDITY Construct validity discusses whether used measurements reflect, whatever is investigated in the RQs. In solution studies for example, the F_1 Score can be used to measure experiment results. It might, however, be more appropriate to discuss the results with another measure that better reflects the actual impact on the SE task (Berry et al., 2012).

INTERNAL VALIDITY Internal validity discusses whether different assumptions have been made between the individuals involved in the study. An important example for this thesis is that whenever a ground truth is manually created, humans tend to interpret the data differently or can simply make errors during creation.

EXTERNAL VALIDITY External validity discusses to what extent results can be generalized or applied to another case. Even though a single case study usually cannot draw a sample that is statistically representative, "the intention is to enable analytical generalization where the results are extended to cases which have

common characteristics and hence for which the findings are relevant, i.e. defining a theory." (Runeson and Höst, 2009)

RELIABILITY Reliability discusses whether the study can be replicated reliably, e.g. with the same results.

Whenever threats of validity are discussed in Parts iv and v, the focus is on internal and external validity. Concerning the *internal validity*, there is always the risk of human errors when data is coded (e.g. annotated, or labeled). If such errors lead to a wrong ground truth of the experiment, results are rendered essentially worthless. Concerning the *external validity*, experiments can almost never be generalized without limitations. It is discussed to what extend experiments can be applied to other data or different areas and which constraints need to be considered.

Construct validity is addressed using the standard measures applied in the field in every experiment of this thesis. However, even those standard measures are criticized (Berry et al., 2012), and their applicability is discussed wherever appropriate[2]. To ensure *Reliablility*, the data that is considered ground truth, the source code, and the results are distributed along with this thesis. It can be downloaded via the following DOI from heiDATA Dataverse Network[3]:

`http://dx.doi.org/10.11588/data/10089`.

2 Especially for the solution studies in Section 15.4 and Section 16.4.
3 `http://heidata.uni-heidelberg.de/dvn`.

8 DATA ACQUISITION

The data for the studies described in Parts iv and v was acquired from ITSs and mailing list archives of OSS projects. This kind of data is freely available, archived, and accessible. Usually large volumes of data and communication activities are stored in ITSs if people work in a distributed manner (Fitzgerald, Letier, and Finkelstein, 2012), which is the case in most OSS projects. Furthermore, ITSs and mailing lists foster communication between project members as well as users in distributed OSS projects (Bertram, 2009; Ernst and Murphy, 2012). Thus little offline communication takes place, which in turn allows to sample comprehensive datasets.

8.1 RESEARCHED PROJECTS

C:GEO c:geo is an Android application that supports playing the real world treasure hunting game geocaching. Users of g:geo use the app to search for the location of geocaches, e.g. "small treasures", that are revealed by finding the correct geographic coordinates. c:geo has between $1,000,000$ and $5,000,000$ million downloads according to the Google Play Store, which indicates that it is widely used.

LIGHTTPD lighttpd is a lightweight from server that promises a small memory footprint compared to other HTTP server implementations. In general users of lighttpd configure the product so that is can serve dynamically generated web pages for their environment. lighttpd is packaged with all major Linux and Unix distributions, which indicates that it is widely used.

MIXXX Mixxx is a music player intended to be used by disk jockeys. Users of Mixxx play sequences of musical tracks that are typically mixed together in a way that they appear to be one continuous track. Although Mixxx is intended for a very specific user group, it is starred over 500 times and forked over 300 times on GitHub, which indicates a wide user and developer range.

OFBIZ Apache OFBiz (Apache Open For Business) is a software product to automate enterprise processes. It integrates enterprise resource planning, customer relationship management, supply chain management and other business applications. Users of OFBiz automate various parts of their enterprise processes, typically by configuring OFBiz to reflect their business processes. OFBiz features success

stories from major companies on its website, which indicates multiple installations in complex environments.

Due to the size and complexity of OFBiz, only the manufacturing component is studied in this thesis.

RADIANT Radiant is a modular content management system. Users of Radiant manage the content of dynamic web pages[1], such as blogs, company profiles, or eCommerce platforms. Radiant is starred over 1,600 times on GitHub, which indicates a large number of installations and users.

REDMINE Redmine is a project management application and in its core an ITS. Users of Redmine manage software development projects and track and discuss SFRs, bugs, or other implementation tasks. Besides this, Redmine features a wiki, a forum, and other additional components. Redmine is used by major companies as well as other major OSS projects such as the Ruby programming language. This and the fact that Redmine is starred over 2000 times in GitHub indicates a wide range of installations and users.

OTHER PROJECTS In addition to the above, the evaluation in Chapter 14 is based on issues and archived emails from nine OSS projects. The dataset contains issues and emails that mix NL and many different programming languages as well as other technical data. The details on these projects are not relevant at this point and will be presented in the according chapter.

PROJECT	part iv, ch. 10	part iv, ch. 11	part iv, ch. 14	part v, ch. 15	part v, ch. 16
c:geo		✓		✓	✓
lighttpd		✓		✓	✓
Radiant	✓	✓		✓	✓
Redmine		✓		✓	✓
Mixxx	✓				
OfBiz	✓				
Other projects			✓		

Table 8.1: Mapping Projects to Studies.

MAPPING PROJECTS TO STUDIES Table 8.1 relates the projects to the empirical studies described in Part iv and v. A red tickmark (✓) is used if the project is studied to understand the problem of SF detection better and a green tickmark (✓) is used if the data of the

1 Served by a web server such as lighttpd.

project is employed to validate the design of a solution. It can be seen from Table 8.1 that Mixxx and OfBiz are only used to understand the problem. Investigating in these projects showed that both projects use various means to discuss SFs besides the ITS. This in turn makes them a weak candidate for SFR detection from ITS data.

Finally, the data of the 'other projects' is used in one experiment, only. This experiment is about data preprocessing and a dataset that consists of a mixture of multiple programming languages and natural language is necessary for validation and to ensure generalizability. Such a dataset could not be derived from the other projects.

8.2 PROJECT CHARACTERISTICS

Table 8.2 gives an overview of the characteristics of the projects from the previous section. These characteristics differ with respect to software type, intended audience, programming languages, project size, as well as the used ITS, to ensure that the sample includes realistic data from various project types. In addition, projects that use their ITS very systematically as well as projects that use their ITS in an ad-hoc fashion are included. It is important to include poorly structured or unstructured data to test whether our methods and improvements can be applied to real life ITSs. E.g. it has been shown that searching in ITSs can be improved if the systems are very well structured (Tran et al., 2009). However, many real life projects use only the predetermined structure in the ITS and we cannot assume a better structure when we design our methods.

Besides a broad range of project characteristics, the following rationals were relevant for selection: c:geo was chosen because the ITS contains more consumer requests than the other projects. Furthermore the software needs to integrate tightly with Android sensors on a technical level so that various levels of abstraction are needed to describe SFRs. lighttpd was chosen because its technical audience includes much technical data, such as excerpts from configuration files, in the ITS and SFRs are usually described on a very low abstraction level. Radiant was chosen because none of its issues are categorized as SFR or bug and the ITS contains fewer issues than the ITSs of the other projects. Redmine was chosen because the ITS is assumably used in a very structured way in comparison to the other projects as it is itself an ITS. Mixxx was chosen since it employs so called *blueprints*[2]., which are comparable to traditional requirement artifacts, in addition to standard issues. Apache OfBiz was chosen because it is rather large in comparison to the other projects in terms of its code base. Furthermore, OFBiz is often employed in mission critical environments such as manufacturing. Both factors might impact the ITS usage and how SFRs are formulated.

2 Offered by the Launchpad ITS: `http://www.launchpad.net`

	C:GEO	LIGHTTPD	MIXXX	OFBIZ	RADIANT	REDMINE	OTHER†
Software Type	Android app	HTTP server	DJ SW	ERP	CMS	ITS	various
Audience	consumer	technician	consumer	consumer	consumer developer	developer	consumer developer
Programming Language	Java	C		Java	Ruby	Ruby	various
ITS	GitHub	Redmine	Launchpad	Jira	GitHub	Redmine	mostly Jira
ITS Usage	ad-hoc	structured	structured	structured	ad-hoc	structured	mixed
Categorizes Issues by	tagging	bug, feature	bug, blueprint	six classes, i.a. bug and feature	tagging	bug, feature, patch	not relevant
ITS Size (in # of issues)	3,850	2,900	2,200	120*	320	19,000	n/a
Open Issues	450	500	1,100	20*	50	4,500	n/a
Closed Issues	3,400	2,400	1,100	100*	270	14,500	n/a
Project Size (in LOC)	130,000	41,000	94,117	n/a*	33,000	150,000	n/a

* With respect to the manufacturing component.

† Issues and emails from 9 different projects. Details are presented in Chapter 14.

Table 8.2: Project Characteristics.

8.3 CONTENT ANALYSIS AND GOLD STANDARD CREATION

To perform the problem investigation studies, datasets and corpora need to be understood in order to describe results clearly and to derive statistical data with respect to the projects. To evaluate the solution designs, a gold standard (a ground truth) is necessary. In this thesis thematic coding (Robson and McCartan, 2015) is applied to annotate the data for the problem studies and to create the gold standards. Essentially, thematic coding can be seen as the assignment of topics to data.

Both, the annotations for the investigation and the ground truth should be as error free as possible and thus it is often suggested that at least two people should agree on a coded dataset (Neuendorf, 2002). However, this involves at least twice the work, which is arguably not feasible for every case study. This section describes how gold standards are developed in the thesis and argues, that the effort to create a gold standard should be determined according to (1) the complexity of the task and (2) the type of evaluation for which the gold standard is used.

For the study in Chapter 10 the ITSs and User Documentation (UD) of three OSS projects was annotated by two coder, respectively. The selected issues and the sections of the UD were read and text that describes or mentions SFs was annotated. During annotation the two coders synchronized the used labels at the end of every coding, so that related software features can easily be identified.

For the study in Chapter 11 four annotators coded every sentence of the drawn issues manually. During this process, every coder developed his own coding schema. Then the four schemata were consolidated into one large schema and all synonyms were merged. Finally, consolidated schema was discussed again, to ensure that every coder was satisfied and all annotations are represented.

For the study in Chapter 14, the gold standard comprises documents with NL, code, stack traces and log file excerpts. Arguably it is possible for a trained human experts to distinguish NL from technical artifacts without the need for another expert to check the gold standard. Nevertheless, more than 10% of the documents in the gold standard were checked at random and none of the checks revealed any error.

For the study in Chapter 15 the gold standard are annotations on the sentence level on an issue corpus. These annotations were also used for the problem study in Chapter 11. As the corpus was used in two studies and the annotation on a sentence level cannot be considered a simple task, we applied best practices for content annotation (Neuendorf, 2002) to create the gold standard: (1) to generate a common understanding of the content and the potential annotations, a coding guidebook was created, (2) to validate the common

understanding, all coders coded a smaller test-corpus of five issues and discussed the results. During discussion, the coding guidebook was revised as appropriate, (3) to ensure the common understanding, two coders annotated each document and the agreement was measured. (4) Finally, the inter-coder agreement was measured. The four annotators agreed on average with a Cohens kappa (Cohen, 1960) of 0.91 on the labels for issue titles and 0.88 on the labels for issue descriptions, which is a rather high agreement. Due to limitations in the used annotation tool, the exact kappa for the issue comments cannot be reported. However, compared to title and description, a significantly lower agreement was observed in random samples of comments. Assumably, this happened due to increasing tiredness or even inadvertence, since annotating up to 50 comments in a single issue is an arduous task.

For the study in Chapter 16 the gold standard is a manually created traceability matrix. It has been shown that even professional human analysts do not achieve perfect results in this task (Cuddeback, Dekhtyar, and Hayes, 2010). Furthermore, it is often a subjective decision, whether two issues are actually duplicates or whether they include related content. Due to the high amount of manual effort necessary for gold standard creation[3], the gold standards for each project in this experiment are created by a different person. However, it is necessary that every coder has the same understanding of a trace. To limit the effects of subjectiveness, the creators first discussed the meanings of each trace type and agreed upon when a trace should be created. Then borderline cases were were discussed, whenever a trace was unclear.

3 $(n \times n)/2$ manual comparisons for n issues.

9 TOOL SUPPORT

In this section the high level tools and frameworks used to implement the experiments in Parts iv and v are introduced.

9.1 GENERAL ARCHITECTURE FOR TEXT ENGINEERING

General Architecture for Text Engineering (GATE) (Cunningham et al., 2016) is used to implement the experiments described in Chapters 14 and 16[1]. GATE is a Java-based LE framework using the pipes and filters architecture pattern (Hohpe and Woolf, 2003, pp. 70). GATE offers many features for text processing such as lexical analysis, stemming, grammatical tagging and so forth. Most features are assembled in GATE by interfacing to other Java implementations such as the popular Stanford Parser (Chen and Manning, 2014). This makes GATE a flexible tool to analyze and exploit text.

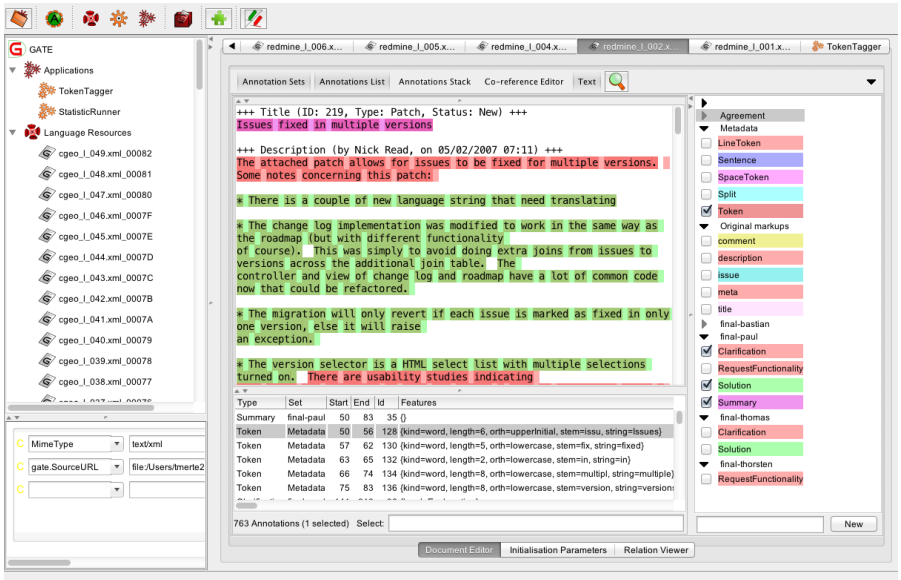


Figure 9.1: Data Annotation with GATE.

For interaction, GATE provides a Graphical User Interface (GUI) that gives immediate feedback and allows to create text processing pipelines. In addition, this interface can be used for manual text annotation as shown in Figure 9.1. In this thesis GATE is used to get a better understanding of the data and to create experimental solutions as well as

1 Together with OpenTrace, see Section 9.2.

to create the annotations for the study in Chapter 11 and the gold standards for the experiments in Chapters 14 and 15.

Furthermore, GATE provides the 'GATE embedded' API. Therewith GATE can be used programmatically without interfering with a GUI. This is especially useful to implement a larger experiment setup in an reproducible manner. E.g. the final experiment setup in Chapter 16 is implemented with GATE embedded due to the experiment size.

9.2 THE OPENTRACE WORKBENCH

The OpenTrace workbench (Angius and Witte, 2012) and the CorpusTools, an extension to OpenTrace for easier experiment conduction (Krämer, 2014), is used to implement the experiments in Chapter 16[2]. OpenTrace is a Java-based tool designed for trace retrieval between NL RAs and includes means to evaluate results with respect to a reference matrix.

To calculate a traceability matrix OpenTrace utilizes IR implementations from the Apache Lucene[3] project, which provides Java-based indexing and search technologies. OpenTrace is implemented as an extension to the GATE framework and it utilizes GATE's features for basic text preprocessing and processing tasks, e.g. for lexical analysis or stemming. For the experiments in this thesis some changes and enhancements were made to OpenTrace:

1. OpenTrace was refactored to be compatible with the current GATE version[4],
2. OpenTrace was enhanced in order to process ITS data, and
3. OpenTrace was enhanced in order to support different variants of the BM25 algorithm.

9.3 NATURAL LANGUAGE TOOLKIT AND SCIKIT-LEARN

The Natural Language Toolkit (NLTK) (Bird, Klein, and Loper, 2009) and the Scikit-learn (Pedregosa et al., 2011) ML tools are used to implement the experiments presented in Chapter 15[5]. These Python-based packages support implementing NLP and ML applications.

2 Cleland-Huang, Czauderna, and Hayes, 2013 presented another tool to conduct trace retrieval experiments. This was not used for two reasons: (1) In contrast to TraceLab (Cleland-Huang, Czauderna, and Hayes, 2013) OpenTrace could be put to work rather quickly and (2) OpenTrace is based on GATE, which the author of this thesis was already familiar with, thus speeding up the development.

3 `https://lucene.apache.org`.

4 At the time of writing, the current GATE version is 8.1.

5 Both tools make extensive use of the famous NumPy (`http://www.numpy.org`) and SciPy (`https://www.scipy.org`) packages. However, introducing these packages is out of the scope of this thesis.

NLTK offers a variety of preprocessing methods like stemming, stopword-removal, and so on. It includes many language processing algorithms, such as grammatical tagging, n-gram extraction, clustering, or ML. In addition to its own functionality, NLTK interfaces with other applications, such as the Stanford Parse (Chen and Manning, 2014) or Scikit-learn for extended ML functionality.

Scikit-learn includes ML tools for classification, clustering, dimensionality reduction, preprocessing, or regression tasks. In this thesis Scikit-learn's ML algorithms[6] are used for classification tasks.

6 Related ML algorithms are introduced in Section 5.2.

Part IV

PROBLEM INVESTIGATION

In this part the problem of software feature detection is analyzed in depth. First, a study compares software features found in an issue tracking system to software features from user documentation and to feature lists. The study proofs that issue tracking systems are a good target for software feature detection. Thereupon another study dives deeper into the language used to describe software feature requests. This study shows how issues are composed and identifies challenges for an automatic software feature detection. Finally, the outcome of all studies is recapitulated and discussed.

10

SOFTWARE FEATURES IN ISSUE TRACKING SYSTEMS – AN EMPIRICAL STUDY

"The real purpose of the scientific method is to make sure nature hasn't misled you into thinking you know something you actually don't know."
- Robert M. Pirsig, Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values.

This chapter describes and discusses an empirical study investigating the first of four RQs defined in Section 1.3:

> RQ 1: What information about SFs can be found in an ITS of a software product, and how well is this information suited to derive a feature representation of the software?

In other words, this research question investigates whether ITSs are actually a fruitful source to retrieve SFs. The study compares the ITS to another source for SFs: the UD. The UD has already been recommended as a substitute for a Software Requirements Specification (SRS) by Berry et al., 2004. It has often been argued that good UD should be complete and that it should contain all important features. Parnas, 2010 stresses the importance of good documentation in SE and argues that documentation, such as UD, should be developed during software design to gain a higher completeness and make the act of documenting less tedious for the software developers. However, it will be shown in this study that UD is not always as complete or up-to-date as it should be and especially the transition from one documentation system to another[1] can deteriorate the UD significantly. Thus UD cannot be seen as a good source for SFs per se.

Additionally, the study investigates different aspects of SFs in ITSs such as the levels of abstraction that are used to formulate SFs. Parts of this study were previously published in (Paech, Hübner, and Merten, 2014).

In the next section the study setup is described and RQ 1, as stated above, is refined in three fine grained RQs. Thereafter Section 10.2 presents the results for each RQ split by the studied projects. Section 10.3 discusses the treats to validity of the study and Section 10.4 shows related work with respect to the investigation in this chapter. Finally, Section 10.5 discusses the implications of SF information in ITSs and UDs for the proposed solution presented and evaluated in Part V.

10.1 STUDY DESIGN

Case study research is applied, as this is an exploratory study trying to understand real-world phenomena. In particular the ITSs, UD,

1 In case of this study, the transition from a single document to a documentation wiki in Apache OFBiz.

and SF lists of tree OSS software projects are analyzed to answer the research questions stated in the next section.

10.1.1 *Research Questions*

The main question stated in the beginning of this chapter is refined in the following three research questions:

RQ 1.1 What feature information can manually be derived from the ITS and the UD?

RQ 1.2 What are the commonalities and differences of feature information from UD and ITS and how well does the information fit to the feature list provided by the developers themselves?

RQ 1.3 How could this information be derived semi-automatically?

RQ 1.1 and RQ 1.2 try to understand whether the ITS is a good source for SFs compared to another promising candidate, the UD. In prospect of Part V, RQ 1.3 then asks for first indications how SFs could be derived semi-automatically.

10.1.2 *Data*

The study was conducted on the ITS data from the *Mixxx*, *OFBiz*, and *Radiant* projects as introduced in Section 8.1. Details of the projects and the ITS data are summarized in Table 8.2 on page 52.

As this particular study compares the UD and the ITS, Table 10.1 additionally compiles quantitative information on the ITS and the UD for every researched project. The Mixxx project employs the concept of blueprints in addition to issues. Blueprints integrate or abstract a number of issues and are usually higher level requirements. Thus for Mixxx all 113 available blueprints and randomly sampled issues (to identify the quality of links between issues and blueprints) were analyzed. For Radiant all 348 available issues were analyzed. In Radiant it was manually identified whether the feature-relevant issues are implemented, since the ITS does not provide according information. Finally, for OFBiz only the manufacturing component and one feature was studied due to the size of the project. The OFBiz ITS contains 5567 issues, 120 of these issues are related to the manufacturing component, which was determined by filtering the ITS. Those issues were studied in depth.

The data sources were analyzed in February 2014 and stored locally to ensure a reliable reproduction of the results.

10.1.3 *Analysis Procedures*

ITS AND UD INFORMATION	MIXXX	OFBIZ	RADIANT
# features in list	22	1	10
Size (in LOC)	94117	n/a	33887
Programming language	C++	Java	Ruby
# issues	2211 + 113 blueprints + 138 user questions	120	348
# issues implemented	1239 + 59 blueprints	94	See text
# issues analyzed	50 + 113 blueprints	all	all
# analyzed issues with SF information	22 + 53 blueprints	19	50
# issues implemented and analyzed with SF info.	22 + 53 blueprints	16	43
# subdivisions UD	14 chapters consisting of 69 sections	343	120 pages
# subdivisions UD analyzed	all	36	all
# subdivisions with feature information	62	34	64
# provided features identified in ITS	21	7	12
# provided features identified in UD	24	12	12

Table 10.1: Issues and User Documentation.

FEATURE INDICATORS Issues that describe a new SF or a Software Quality (SQ) potentially include features. Those issues that containing already implemented features that mention an SF, an SQ or a component of an SF or an SQ are included in the analysis. It is not always easy to determine the implementation status of an issue. E.g. in the Radiant ITS, the implementation status is not managed explicitly. Thus, the implementation status needs to be revealed by analyzing the comments of an issue and the associated commits. For Mixxx and OFBiz the issues with the status *Implemented*, or *Patch Available* are taken into consideration. During data analysis no indication could be found that the implementation status was assigned wrongly.

In the UD those section and page titles that contain SF and SQ information were analyzed, similar to the issues in the ITS. In case of the UD, however, it can be assumed that the SFs and SQs described in the UD are actually implemented. This assumption was affirmed during data analysis.

To derive the SF related information systematically, the indicators in Table 10.2 were used. Below the feature indicators, the table shows relevant examples.

FEATURE ABSTRACTION LEVELS After SF identification, the information was classified according to its abstraction levels. It is well-known that requirements and features are typically described on dif-

ITS	UD
The issue is implemented AND mentions functionality or quality X *AND is not related to a bug AND is not only related to refactoring AND the term* X *or a component* X_i *of* X *is explicitly or implicitly mentioned.*	*The item describes functionality or quality* X *AND not operation (such as installing or getting help) AND the term* X *or a component* X_i *of* X *is explicitly or implicitly mentioned.*
j Radiant (Quality Performance, Component "Radius Parser" of "Radius Template Language"): *"Speed up Radius parser"*	Radiant Page Titles (Quality Performance and Caching) *"Disable caching in a radiant system"*
Radiant (Functionality Asset Management): *"Integrate an asset management solution"*	Radiant Page Title (Functionality Admin UI) *"Altering Tabs in the Admin UI"*
Radiant Implementation Status by comment: *"Seeing as there's a setting for this now, this issue can be closed?"*	Mixxx (Quality was not mentioned)
Mixxx (Quality User Experience, Functionality Vinyl Control): *"Improvements to the overall vinyl control user experience"*	Mixxx Section (Functionality Broadcast): *"Live Broadcasting Preferences"*
Mixxx (Functionality, Component Crates and Playlist): *"Currently Mixxx does not support hierarchies for crates and playlists. This, however, is possible"*	OFBiz (Functionality Routing Task): *"Find Routing Task"*
OFBiz: (Functionality Production Machines) *"cover the case in which many machines are used to complete a production task"*	

Table 10.2: Rules to Identify Feature Relevant Information.

ferent abstraction levels. Based on the work by Gorschek and Wohlin, 2006, three abstraction levels of features are distinguished:

REQUIREMENTS LEVEL The mentioned SF comprises several functions or the SQ affects several functions[2].

FUNCTION LEVEL SF or SQ refer to a single function, which a user can perform. Implementation details are not mentioned.

CODE LEVEL SF or SQ refer to a single function, which a user can perform. Implementation details are mentioned (similar to the component level used in Gorschek and Wohlin, 2006, it focuses on "how something is implemented").

Table 10.3 exemplifies issue texts written in different levels of abstraction. Those levels can be mapped to the UD structure: page or section titles usually refer to requirements and sub-pages and subsections usually refer to functions. Code details were only mentioned in the UD of Radiant, as the user is required to change classes to set certain functionalities in this content management system up.

In addition to the abstraction levels, the distinction by Berry et al., 2004 for typical section types of an UD is incorporated: the abstractions of the domain, also called the Domain Objects (Os) and the Use Cases (UCs). O is used when the feature is an object that can be identified as a direct part of the GUI or the software. UC is used when the feature requires some kind of dialogue between the user and the system similar to the UCs described by Cockburn, 2001.

Finally, the relations between features were identified. The relations are reflected particularly in visualizations, such as Figure 10.2 and Figure 10.5. Figure 10.1 introduces a legend how relations are visualized particularly. Based on the information available, the following relations between the identified features could be determined.

IDENTICAL features of different sources

PART OF features of different and same sources

OVERLAPPING features of different source)

As this chapter is based on (Paech, Hübner, and Merten, 2014), the second author of this publication researched the UD, the thesis' author and third author of the publication researched the ITS, and first author acted as a reviewer.

RQ 1.1 was answered using thematic coding. The UD pages, UD sections, and the ITS issues were coded manually. Then a set of codes characterizing the features was derived. Each coding resulted in one feature set. Those feature sets were compared to feature lists provided

2 Also called feature level in (Gorschek and Wohlin, 2006)

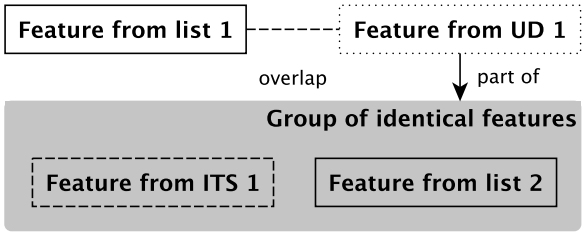


Figure 10.1: Feature Graph Definition.

	FUNCTION	QUALITY
Requirements	*Radiant:* "Break Radiant into several different extensions" *OFBiz:* cf. Table 10.2, example bottom left.	*Radiant:* Internationalization
Function	*Radiant:* "Errors when changing your password should be shown" *Mixxx:* "Implementation of a traktor library feature to allow professional DJs the smooth migration [...]." *OFBiz:* "Improve mrp to support to products which have no orders against them"	*Radiant:* "Make it so that pages are only cached for GETs" *Mixxx:* "Smooth Wave-forms" (relates to a less stuttering display for track visualization). *OFBiz:* "There is a need to be able to block viewing info except that info that may pertain to that login"
Code	*Radiant:* "Javascript to stop you from navigating away from a page with changes" *Mixxx:* "It would be nice to be able to specify multiple <option>s for MIDI controls in XML mapping files." *OFBiz:* "[...] accepts the partyId as a parameter, but has been commented [...] [however, the] functionality is vital for determining which employees are responsible for rejects"	*Radiant:* "[Add] Ruby 1.9.x compatibility" *Mixxx:* "Distribute Launchpad translations with Mixxx Releases"

Table 10.3: Examples for Abstraction Levels in Issues.

on the project's website and with each other. As these feature lists are placed prominently on the websites, it can be stipulated that they are most marketing relevant for the project members.

RQ 1.2 was answered by comparing the two feature sets from UD and ITS in depth and the answer of RQ 1.3 is based on the experience of the two coders.

10.2 RESULTS

In the following, the research questions are answered for each project individually. The last section summarizes the insights over all projects.

10.2.1 *Results for Project Radiant CMS*

PROVIDED FEATURE LIST The feature overview[3] on the Radiant website depicts ten SFs using a name and a short one- or two-sentence description. Table 10.4 shows these features and our classification as SF or SQ and O or UC. The table points out whether the feature was identified in the ITS or UD. Brackets indicate that the related ITS or UD features are formulated slightly different from the feature list.

PROVIDED FEATURE LIST	IDENTIFIED IN
Built with Ruby on Rails (SQ,O)	ITS
Custom Text Filters (SF,UC)	-
Flexible Site Structure (SQ,O)	-
Intelligent Page Caching (SQ,O)	ITS, UD
Layouts (SF,O)	(UD)
Licensed under the MIT License (SQ,O)	-
Pages (SF,O)	ITS
Radius Template Language* (SF,O)	ITS, UD
Simple Admin Interface (SF,UC)	ITS, UD
Snippets (SF,O)	(ITS, UD)

* a special macro language (similar to HTML and Ruby).

Table 10.4: Radiant Features.

RQ1: FEATURE INFORMATION FROM UD AND ITS The UD is organized in a wiki. The starting page of this wiki is a global table of contents and the table of contents is divided into eleven chapters. Eight of which deal with administrative issues.

Three chapters could be identified as primarily relevant for further analysis: *The Basics*, *How Tos*, and *Extensions*.

3 `http://radiantcms.org/overview`, accessed on August 8, 2014.

The Basics contains seven links to top level UD pages. Except for the links to *FAQs* and *Getting Started*, the links point to pages describing Radiant features as mentioned in the feature list (Pages, Layouts, Snippets, Radius Tags, Customizing the Admin GUI). In addition, there are six links to details of the Radius Tag feature and two links to details of the admin GUI feature. Each top level UD page contains the intent and summary of the feature, a screenshot of the GUI, and descriptions how to use the feature.

The *How Tos* chapter contains 29 links to top level UD pages. As visible by the titles, those links point to tutorials describing advanced features. The tutorials include usage examples and reference the basic feature pages. The Radius Template Language is referenced from almost all pages.

The *Extensions* chapter starts with six pages describing the concept and usage of Radiant extensions, followed by a list of 27 common extensions, and 11 pages that describe how to develop an extension for Radiant. According to the indicators of Table 10.2 64 pages are relevant to SFs in total.

Figure 10.2 shows the relations between the radiant features at a glance and brackets indicate the different abstraction levels. The boxes marked with *UD* on the right side show the 13 features identified from the UD. *Content delivery* refers to different content representation methods such as HTML or RSS. *Content location* refers to the search functionality. Most pages are on the function level and many describe layout and the pages count a feature is described with does not signify the importance of that feature.

Radiant uses GitHub for different aspects such as SFRs, bug reports, discussions of the development process, discussions about refactorings, user problems, or discussions about documentation. However, GitHubs optional issue tags are rarely used, so that issues related to features, bugs, or other aspects of SE are not categorized by any means. Therefore, all of the 348 issues were manually analyzed and their category was derived from the descriptions and comments.

The boxes marked as ITS on the left side in Figure 10.2 refer to the eleven SFs identified from the ITS. *Asset management* refers to content such as image or PDF files. *Development* comprises support for website developers. *Frontend* refers to usability. A lot of issues deal with the *Simple Admin User Interface (UI)*. Although the admin UI is an important Radiant feature, the number of issues does not signify the importance of the feature in general. In addition some issues contain many comments, but at the same time they have a very short implementation in terms of LOC.

RQ2: COMMONALITIES AND DIFFERENCES OF UD, ITS AND PROVIDED FEATURE LIST Figure 10.3 shows the abstraction levels of the identified features. As it could be expected, the feature descrip-

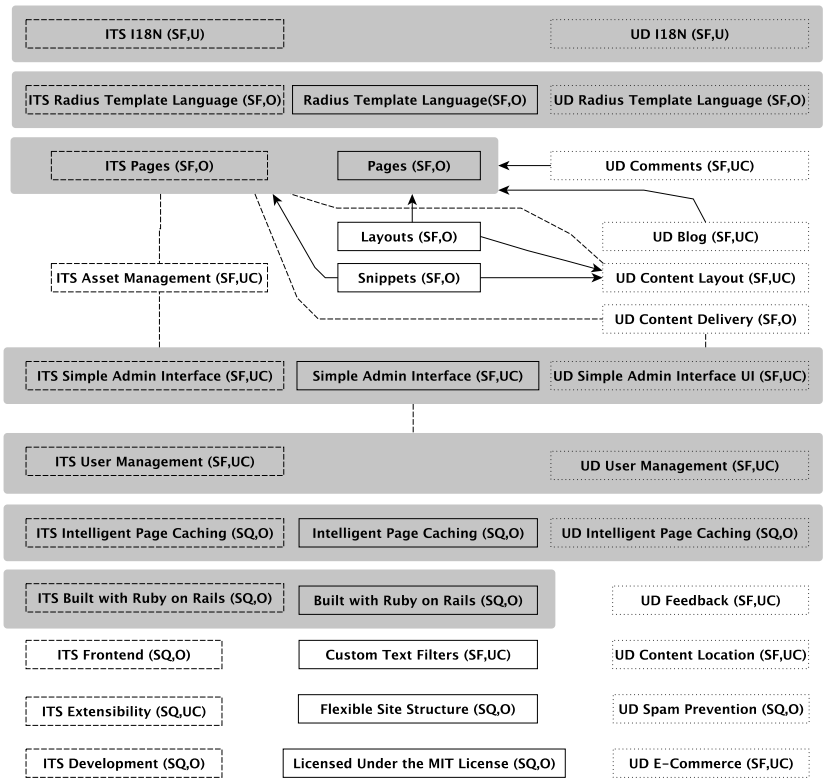


Figure 10.2: Radiant Feature Graph (No Transitive Relations Shown).

tions in the ITS are quite often on the code level, while the UD features are described on all three levels. Almost a third is on the code level which is rather unusual for an UD. However, website developers are the users of Radiant and thus special domain specific languages are employed for some functionalities in Radiant.

Figure 10.4 shows commonalities as well as differences between the different feature sources. Almost half of the ITS features (45%) are identical to the provided feature list, while only a third of the UD features (31%) is identical.

The 18% provided features not identified from the ITS may be due to the fact that the ITS was not used from the beginning of the development. The basic functionality of Radiant was implemented before the ITS was used. For ITS features not in the provided list (31%), the content could be a reason. While *Internalization* and *Extensibility* seem relevant as prominent features, some website *Development* features might be too low-level. Interestingly all ITS features that do not have a relation to either the list or the UD are quality-related. So in the case of Radiant the ITS is the best source for quality related SFs.

All features identified from the UD seem relevant, although 31% of them are not in the feature list. The part-of relations between *Pages*, *Layouts* and more fine-grained features, such as *Blog* or *Comments* in Figure 10.2 show that granularity is a challenge since only UD features

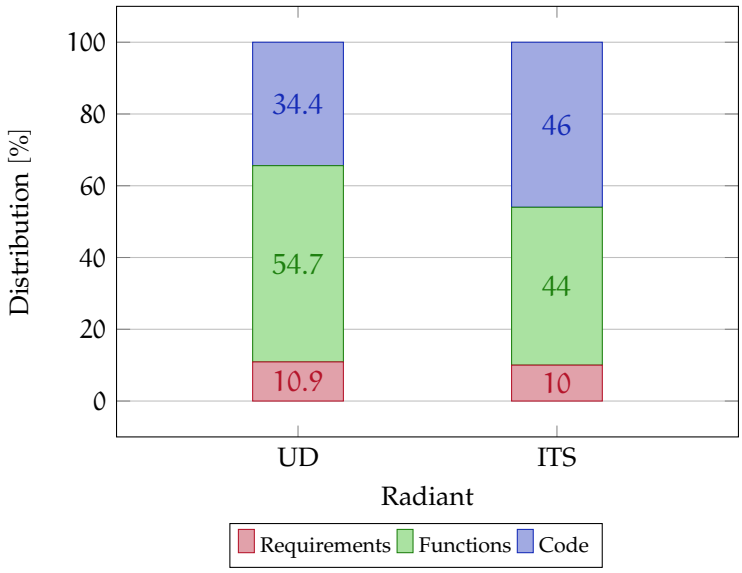


Figure 10.3: Radiant: Feature Abstraction Levels.

are organized using part-of relations. Finally, UD features are more closely related to ITS features (45% are identical) than to the feature list.

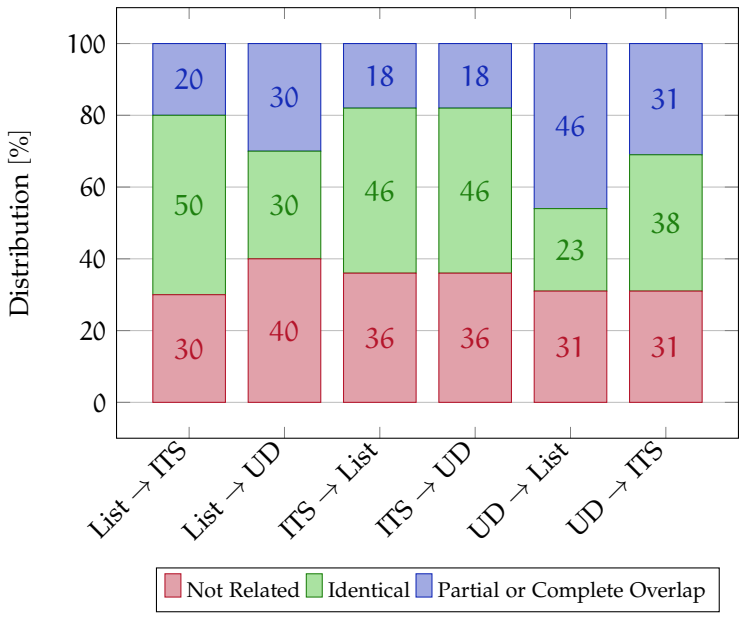


Figure 10.4: Radiant: Feature Relations.

RQ3: AUTOMATIC IDENTIFICATION OF FEATURE INFORMATION
Most feature-related information can be identified in older issues. 34 SFs could be found in issues #1 to #68. 16 SFs in #71 to #202 and no SFs was found in #203-#384. This suggests that a) older issues should be available for automatic feature extraction and b) it might be best if the ITS is used from the beginning of the development (e.g., design

and prototyping phases). In Radiant however, the ITS was used after a prototype of the software had already been built.

27 of the 43 issues with SF related information contain the text '*should*be*', '*add*', '*would*be*' and '*allow*user*' in the issue title or description. However searching for these terms includes about same amount of refactoring- and bug-related issues. Hence, the precision of a keyword based approach is relatively low. However, depending on the usage of an ITS, it might be possible to extract more precise search terms. A pitfall in automatic analysis are the tags of the Radiant ITS. Although tags like *bug*, *design*, or *javascript* are introduced in Radiant, they are not used consistently. Since tags are optional, most issues are not tagged at all. The *bug* tag is used for only a single issue, which is not reliable for any automatic approach. Further efforts, for example using ML models, are necessary to identify feature labels.

For the automatic identification of SFs from the UD, the relevant pages need to be identified in a first step. In the Radiant UD the relevant pages can be found in three chapters. These chapters can be identified more efficiently manually than automatically as the semantic structure of the UD needs to be understood. Some pages are only related to system operation and do not contain any SFs. To a certain extent, these pages could be identified by searching for operation-related terms such as 'installation' in order to discard non-feature-relevant pages. Another input for the identification of relevant pages is the linkage structure of the wiki. Basically, the most frequently referenced pages most likely are feature-relevant. Thus a high in-degree in degree with respect to hyperlinks is an indicator for an important SF.

10.2.2 *Results for Project Mixxx*

PROVIDED FEATURE LIST Similarly to Radiant, the list of provided features including short marketing descriptions was taken from the website[4]. The feature list contained 20 functional features as shown in Table 10.5. The table lists SFs which are part of a more general feature (such as *Powerful Library* or *Decks*) in one row.

RQ1: FEATURE INFORMATION FROM UD AND ITS The Mixxx UD is part of a general documentation wiki. The wiki combined developer documentation and documentation for Mixxx users. This study focuses on the user manual. The manual contains 14 chapters, nine of which are feature relevant. These nine chapters contain 69 sections. According to the introductory text of the chapters requirements level SFs could found in eight of them. The 54 sections which satisfied the feature indicators of Table 10.2 are all on the function level. Finally the UD describes no SFs on the code level.

4 `http://mixxx.org`, accessed on August 8, 2014.

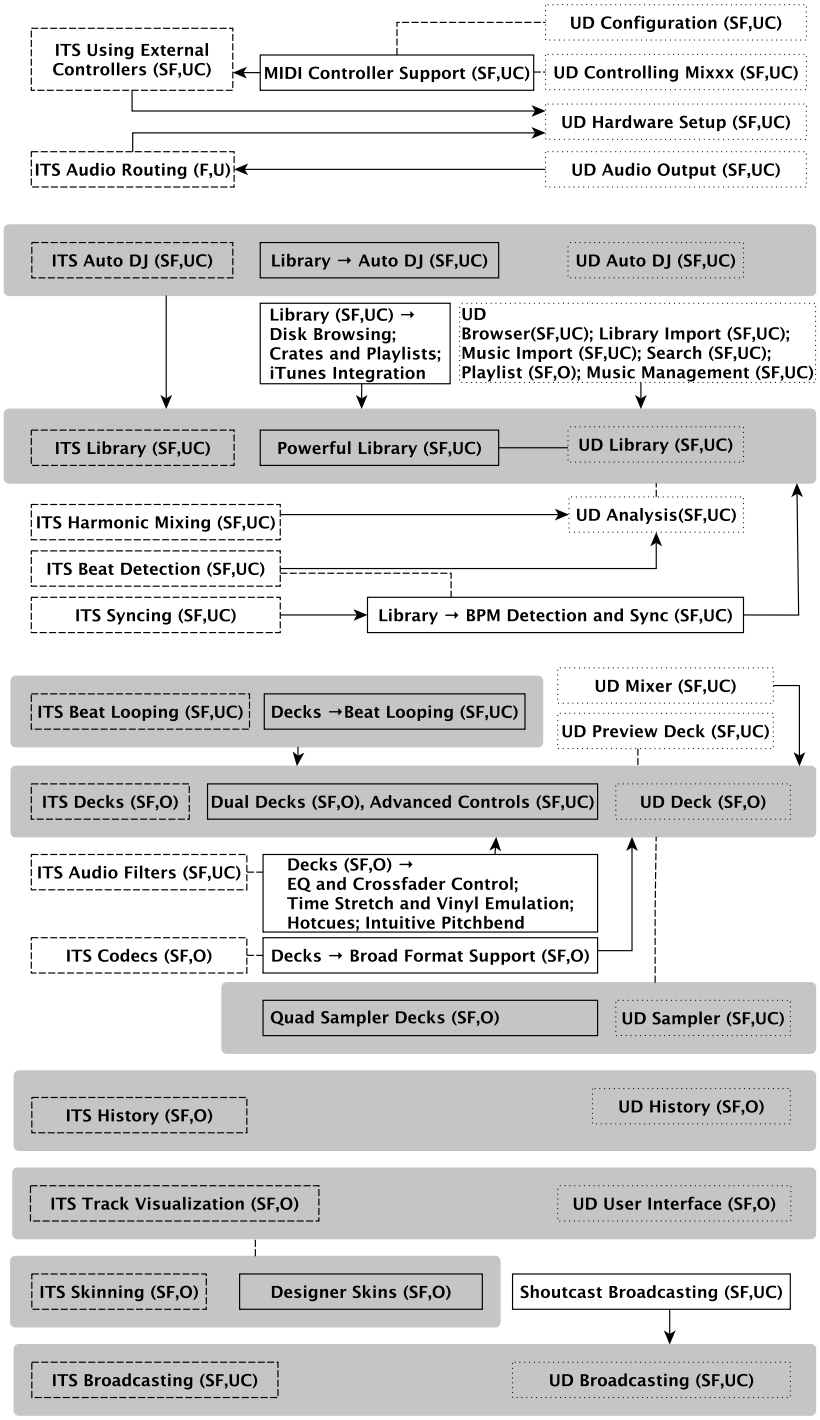


Figure 10.5: Mixxx Feature Graph (No Identical, Unrelated Features, or Transitive Relations Shown).

PROVIDED FEATURES	IDENTIFIED IN
Advanced Controls (F,U), Dual Decks (F,O)	ITS,UD
Decks: Beat Looping. Broad Format Support, Hotcues, Intuitive Pitchbends, EQ and Crossfader Control, Time Stretch and Vinyl Emulation (F,O)	(ITS, UD)
Designer Skins (F,O)	ITS
Free Timecode Vinyl Control (F,U)	ITS,UD
Microphone Input (F,U)	ITS,UD
MIDI Controller Support (F,U)	(ITS,UD)
Powerful Library: Auto DJ, BPM Detection and Sync, Crates and Playlists, Disk Browsing, iTunes Integration (F,U)	ITS,UD
Quad Sampler Decks (F,O)	UD
Recording (F,U)	ITS,UD
ReplayGain Normalization (F,U)	-
Shoutcast Broadcasting (F,U)	(ITS,UD)

Table 10.5: Mixxx Features.

The boxes marked with *UD* on the right side in Figure 10.5 show 20 of the 24 identified features and their classification (the features *DJing (SF,UC), Microphone (SF,UC), Recording (SF, UC)* and *Vinyl Control (SF, UC)* are not linked to the provided features and thus have been omitted in the Figure). The feature *Analysis* refers to the preparation of harmonic mixing[5], *Controlling Mixxx* allows setting device specific options, and *Vinyl control* allows to use records to control the digital playback from the computer with special time-code vinyl or CDs. All of the identified features describe a functionality. The number of sections or chapters corresponds to the complexity of the features. *Music Management* is the only feature described in detail, which is not directly related to a UD chapter.

The Mixxx ITS contains 2211 issues (including bugs and feature requests), 113 blueprints and 138 user questions. The issue classification in bugs and features as made by the developers is very reliable for the issues that were analyzed. The blueprints describe refactorings and higher level requirements for features. Blueprints and issues are often linked together and issues are often linked with the code. However, cases could be identified where a link to to code would be appropriate but not existent. Since blueprints contain more feature-relevant information than issues in the way the ITS is used in the project, all 113 blueprints were analyzed for SF information. 59 of the 113 blueprints were already implemented.

5 Mixing songs together that have the same or a harmonically compatible minor or major key.

The boxes marked with ITS on the left side in Figure 10.5 refer to 15 of the 21 features identified from the ITS (*Development (SQ,O)*, *Internationalization (SF,O)*, *Microphone Usage (SF, UC)*, *Playback (SF, UC)*, *Recording (SF, UC)* and *Vinyl Control (SF,UC)* are not linked to the provided features and thus have been omitted in Figure 10.5). All of the identified features describe functionality. *Beat Detection* analyzes the speed of a track. *Beat looping* repeats a short part of the track. *Codecs* allow the use of different digital formats. As for Radiant, *Development* describes support for the developers. *Skinning* refers to different GUI looks that can be applied in Mixxx. *Syncing* automatically matches the speed of different songs for the mix. The blueprints are generally very organized and use higher abstraction levels. Only few blueprints are described on the code level as shown in Figure 10.6.

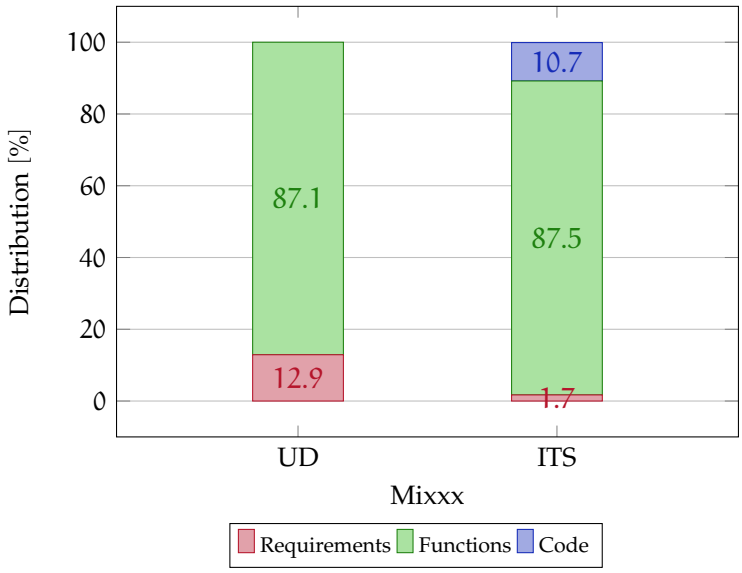


Figure 10.6: Mixxx: Feature Abstraction Levels.

RQ2: COMMONALITIES AND DIFFERENCES OF UD, ITS AND PROVIDED FEATURE LIST Figure 10.7 illustrates the commonalities and differences of the UD, ITS, and the provided feature lists.

There are fewer provided features compared to Radiant which cannot be found in the UD or the ITS. This was expected from the fact that the ITS blueprints and the UD seem to be well maintained. Similar to Radiant, more features from the provided list were directly identified from the ITS (40%) compared to the 30% of the provided features identified from the UD.

Furthermore, there are more ITS features (24%) which are not related to the feature list and many identical features between ITS and UD (between 30 and 40%) can be found. However, there are also many part-of relations between ITS features and either UD (48%) or the provided features (38%). And there are even more part-of relations from UD features to either the ITS (58%) or the provided features (63%).

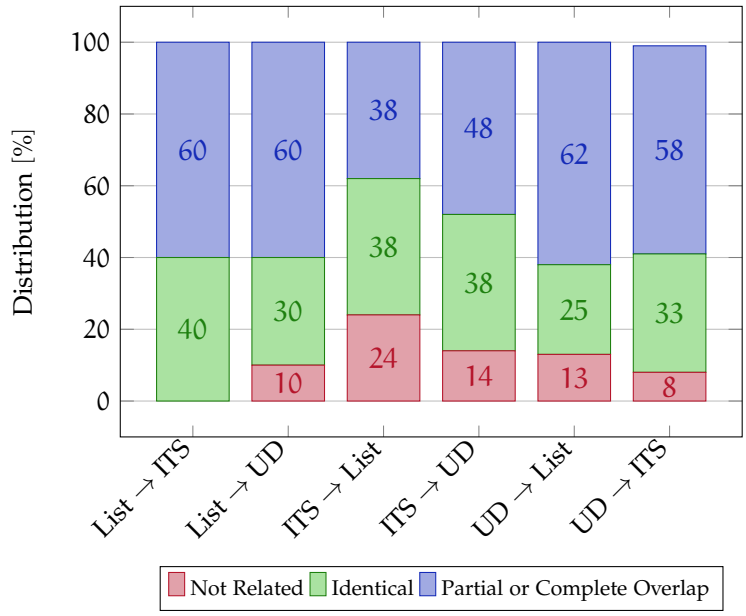


Figure 10.7: Mixxx: Feature Relations.

This indicates that the granularity and abstraction level in the different sources is different and multiple sources complement each other. The distinction between SF und SQ is not relevant as no SQ features were found.

RQ3: AUTOMATIC IDENTIFICATION OF FEATURE INFORMATION

As the Mixxx ITS is maintained very systematically, the possibilities to categorize the status (e.g. implemented, draft, in progress) as well as the issue (e.g. bug, wishlist, blueprint) can be used to find feature relevant information. The identification of the feature description remains a problem though as the categorized are not available on the level of data fields or even paragraphs.

The Mixxx UD is a well-structured document separated into chapters and two section levels. Nine of the 14 chapters are feature-relevant, compared to three out of 11 for Radiant. Often the chapter and section titles directly contain feature-relevant terms. As for Radiant the relevant chapters can be identified just by manually looking at the chapter titles. For the identification of features on the requirement abstraction level, the chapter titles can be used. The section titles on the first section level can be used for feature identification on the function abstraction level. Overall the Mixxx UD structure can decently be mapped to SFs and their abstraction levels and such a mapping would not yield many errors.

10.2.3 *Results for Project Apache OFBiz*

PROVIDED FEATURE LIST The OFBiz project was studied only partially. Because the project is very large, a complete analysis was not feasible and it was decided to research the *Manufacturing Management* feature with the related UD and issues. The manufacturing component can be found on the OFBiz feature page[6], which lists all the SFs on a requirements level.

RQ1: IDENTIFICATION OF FEATURE INFORMATION FROM UD AND ITS The UD for OFBiz is organized as a wiki. However, the wiki contains only more or less empty pages and basic structures without content (e.g. sections for role specific documentation). Because of this the wiki could not be analyzed. Instead the outdated *Manager Reference* was used for the UD analysis (last updated in 2004, uploaded to the wiki as PDF attachment between Dezember 2006 and January 2007). Based on the experience gained from the previously analyzed projects, the chapter and the section headings were analyzed in depth.

FEATURE UD	FEATURE ITS
Bill of Materials (SF, UC)	Data Security (SQ,O)
Bill of Mat. Simulation (SF, UC)	Internationalization (SF,O)
Calendar (SF, UC)	Manage orders (SF, UC)
Job Shop (SF, UC)	Manage Products (SF, UC)
Manufact. Resource Planning (SF, UC)	Manage Production Machines (SF,O)
Manufacturing Rules (SF, UC)	Manage Production Runs (SF, UC)
Production Run (SF, UC)	Resource Planning (SF, UC)
Reports (SF,O)	
Requirement Verification (SF, UC)	
Routing (SF, UC)	
Routing Task (SF, UC)	
Shipment Plans (SF, UC)	
Status Report (SF, UC)	

Table 10.6: OFBiz: Manufacturing Component Features.

The UD 343 subdivisions organized into four hierarchy levels. The sections on the 3rd and 4th hierarchy level could be removed since they refer to single attributes, e.g. particular values for input forms. Eight chapters and 28 sections remain.

From this, 13 distinct features could be identified (confer the left column of Table 10.6). Eight of the 13 features are on the requirement abstraction level. The other five features are mostly finer grained aspects of the requirement abstraction level features. As expected for a manager reference, no code level details were mentioned. Also, quality features were not mentioned and only domain object could be found.

6 `http://OFBiz.apache.org`, accessed on August 8, 2014.

The OFBiz project uses the Jira ITS. All 120 issues of the manufacturing component were analyzed and seven features could be identified (cf. Table 10.6 right column). *Security* was the only SQ aspect, but again this quality could be found only in the ITS and not in the UD.

Issue feature descriptions are generally longer than in the other projects. Requirements, for example, are described in detail and often include multiple solution ideas such as in the following issue excerpt: "...this can be implemented in many ways: a) expanding the concept of Fixed Asset groups ... b) (more complex) add new association entities to link a task ...". Although this diligence suggests a very accurate handling of the ITS, multiple miss classifications could be found (e.g. bugs classified as improvements).

Overall, the ITS meta data is not maintained as well as the NL descriptions. In addition, some SFs are distributed over many issues. E.g. I18N[7] included multiple issues for every single language and a main issue describing the I18N feature itself. The feature is another example for unmaintained meta data as none of these issues were linked together.

FEATURES ITS (7)	FEATURES UD (13)	MAP
Data Security (SQ,O), Internationalization (SF,O)		
Manage Orders (SF, UC)	Bill of Materials (SF, UC), Bill of Materials Simulation (SF, UC), Calendar (SF, UC). Shipment Plans (SF, UC)	(O)
Manage Products (SF, UC), Manage Production Machines (SF, O)	Manufacturing Rules (SF, UC)	(O)
Manage Production Runs (SF, UC)	Production Run (SF, UC)	I
Ressource Planning (SF, UC)	Manufacturing Resource Planing (SF, UC)	I
	Job Shop (SF, UC), Reports (SF, O), Requ. Verification (SF, UC), Routing (SF, UC), Routing Task (SF, UC) Status Report (SF, UC)	

Table 10.7: OFBiz: Commonalities and Differences of ITS and UD.

RQ2: COMMONALITIES AND DIFFERENCES OF UD, ITS AND PROVIDED FEATURE LIST As the descriptions of the UD are coarse and mainly the headings were analyzed, a full mapping could not be derived for this project. Table 10.7 shows a rough mapping of the features of UD and ITS. Two features are identical and few have overlaps. However, almost half of the UD features were not mentioned in the issues. This can be explained by the fact that the project switched to

7 Internationalization.

a different ITS (e.g. to Jira) and older issues related to the UD features were likely not transferred.

RQ3: AUTOMATIC IDENTIFICATION OF FEATURE INFORMATION The OFBiz analysis did not reveal any new insights with respect to automatic identification. For the UD only the chapter and section titles could be the basis for an automatic identification. As for Radiant, most feature related information was identified in older issues. In the ITS, the feature to bug ratio was ten to 18% between 2006 and 2009 and only about three to five percent between 2009 and 2013.

10.2.4 *Overall Results*

RQ1: IDENTIFICATION OF FEATURE INFORMATION FROM UD AND ITS ITS as well as UD can serve to identify features. The features, however, are described on different abstraction levels (cf. Figure 10.8). For both, Mixxx, and OFBiz the UD does not contain features on code level and > 75% on function level. In contrast to the other projects, the UD of Radiant mainly contains feature information on the code and function levels. In the ITS feature information is found on all three levels, but, similar to the UD, there are only few features on the requirements level, and the distribution of abstraction levels in the ITS differs noticeably for each of the projects as shown in Figure 10.8.

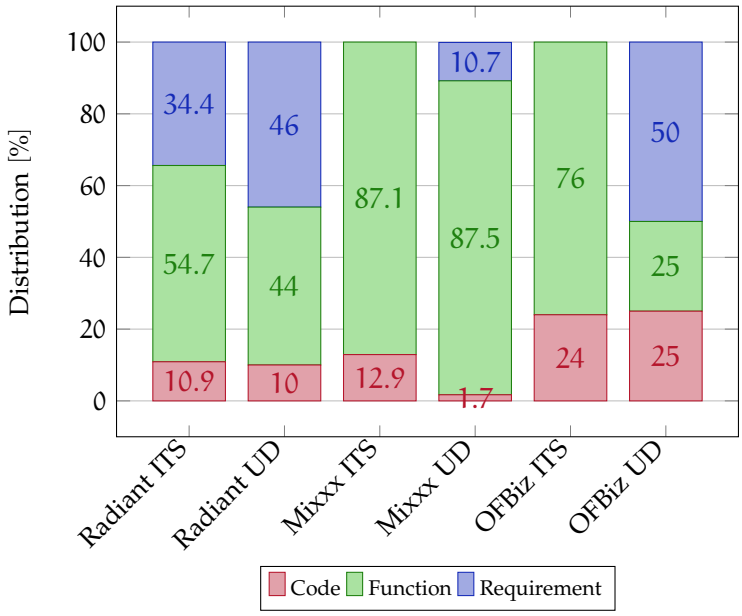


Figure 10.8: Abstraction Levels for Software Features.

Quality features are typically not mentioned in the UD. Radiant and OFBiz UD's mention few quality features and Mixxx's UD none. Most quality features were found in the ITS. Overall neither ITS nor UD were a perfect source for SF detection in the analyzed projects.

However, the ITSs were not used consequently from the beginning. Hence a higher amount of SFs in the ITS is likely in other projects. Finally, it seems likely that multiple sources, e.g. the ITS and the UD, must be searched and combined to yield the complete feature set. Of course other sources or other means of feature location (Dit et al., 2013) could also be added as potential sources.

RQ2: COMMONALITIES AND DIFFERENCES OF UD, ITS AND PROVIDED FEATURE LISTS There is a noticeable overlap between the feature information in the ITS, the UD, and the provided feature list. In the Radiant project, roughly a third of the listed features could not be identified by UD or ITS, in the Mixxx project only a tenth could not be identified. Similarly, for Radiant only a third and for Mixxx only a tenth of the features was not related between ITS and UD. This indicates that, both ITS and UD can be used to record feature information systematically.

As the ITS is mainly important for the developers and the UD is targeted to the users, it seemed more likely that the UD records the listed features better, as argued e.g. by Berry et al., 2004. However, it turned out that UD and ITS record the listed features equally well. It is interesting that in both projects 30-50% of the features were identical (between list and UF, list and ITS, and UD and ITS). In case of Mixxx, there is a high percentage of overlapping features, while in the case of Radiant there are few overlapping features. Thus, even for a systematically documented project like Mixxx, a feature representation generated from UD or ITS are different from the marketing feature list.

RQ3: AUTOMATIC IDENTIFICATION OF FEATURE INFORMATION This study reveals preliminary insights for automatic identification. It seems feasible to manually delimit the relevant pages of the UD and to focus on page or section titles to identify SFs. Also for the ITS simple keywords can be used as a first idea to categorize the relevant issues, but recall and precision are far from perfect. Thus, the case of ITS SF detection does require more sophisticated techniques.

Although best practices in RE suggest to describe new features as as-is and to-be situations, only one issue could be found that mentions both situations. Generally, the to-be situation is described and the as-is situation is implicit. Furthermore, the quality and use of language, the quality of descriptions as well as categories and links differ from project to project. The NL is the most reliable source to detect SFs. Furthermore, an automatic identification needs to be "tuned" for each project. E.g. supervised learning could be a good choice if a model is trained for each of the projects individually.

From the coders experience, the following hints for using ITS and UD to record features instead of setting-up a separate feature documentation can be seen:

- The Mixxx project (as many other projects) shows that feature information can be explicitly managed within an ITS, if it is separated from (but linked to) the usual stream of bugs and change requests. Furthermore, it seems likely that issues in ITS could profit from an abstraction classification or traces to more abstract information.
- Berry et al., 2004 recommend structuring an UD into objects, use cases and advanced features. The UDs of the projects have some similarities to this structure but not enough for automatic approaches.
- Blueprints and issues are much simpler to allocate to software components as both have a technical nature. Without detailed knowledge of the software architecture, this is almost impossible for UD. If relations between features and software components are important, ITS should be used as SF source.

Further implications for an automatic extraction of SFs are discussed in Section 10.5.

10.3 THREATS TO VALIDITY

CONSTRUCT VALIDITY The authors have not been involved in the development of the sources. Thus, our view of what constitutes a feature of the software is clearly an external one, which might be different from what developers consider a feature of their software. To mitigate this threat, the feature list provided by the developers was used for comparison.

EXTERNAL VALIDITY The results are not representative for application software in general, as only three projects were researched. However, for this exploratory study very different projects were chosen to boost the possible insights.

RELIABILITY As only one researcher coded the ITS and, respectively, the UD information, it cannot be claimed that other researchers would reproduce the coding. However, explicit coding indicators were used and explicitly discussed to minimize the bias of the individual coder. Moreover, the performed approach can be adapted to any software development project which provides the required data (e.g. ITS, UD and feature List).

10.4 RELATED STUDIES

In this section related work that derives feature information from diverse sources manually or semi-automatically, is discussed.

Berry et al., 2004 suggest to use user's manuals as the SRS. They validate their suggestion in three case studies and find that UD can be used to document requirements in some cases, at least if the UD is well written. In contrast this chapter examines existing UD to search for SFs. It was found that some UD are so outdated that they cannot be used as a substitute for an SRS.

Ghazarian, 2012 identifies generic classes of software functionality from 15 different requirement specifications in the domain of web-based enterprise systems. The identified classes such as data input or user interface navigation could potentially be useful as indicators of feature information. He also describes that much feature-related information could be found and categorized analyzing only a small amount of issues, respectively only section and paragraph names in the UD. His classification, however, is very technical and uses low abstraction levels. In contrast, this work classifies SFs for different abstraction levels.

Noll and Liu, 2010 analyze an OSS project to identify by whom and where requirements are proposed. They select 13 given features and then trace them. In contrast this study first looks at data sources to manually identify features and then compares them with the given feature list. Thus, this study gathers more data about how SFs are described in detail.

Alspaugh and Scacchi, 2013 discuss how ongoing software development can be done without the need for classical requirements. They too found that SF descriptions are the most prominent requirement-like artifacts in OSS projects. In addition they found that SFs are often written as attributes to existing software versions, competing products or a prototypical implementation. In combination with this chapter their findings suggest that SFs need to be analyzed further, especially with respect to the components and NL that comprise an SF.

10.5 IMPLICATIONS FOR SOFTWARE FEATURE DETECTION

The exploratory study of the OSS projects has shown commonalities and differences of SF information in UD and ITS and given feature lists. The results are promising in the sense that ITS and UD both include relevant features with respect to the projects. The results also show that deriving a complete feature set semi-automatically will be very difficult and as both sources are incomplete in all studied projects.

Some of the feature descriptions formed patterns (e.g. headings in the UD often denoted features). However, most of these patterns were not transferable to other projects.

During the research for this chapter, it could be found that feature information is likely contained in only few issues of an ITS. Due to an ITS's nature, other issues, such as bugs or refactoring tasks are also tracked. Although many ITS provide the option to categorize is-

sues manually, the quality of such manual categories depends largely on the project. Therefore, an analysis of the NL is needed to identify feature information. Furthermore, the feature information can be scattered all over the issue and can be found in title, description or even comments (although title and description are most common). Therefore, only a very small part of the NL in an issue contains the SF request. The rest is information like rationales, solution ideas, social interaction, and so on. These exact contents of an ITS are studied in the next chapter in detail.

For the UD, the starting point to detect and extract feature data semi-automatically is the structure of the respective documents. The analysis of the projects in this study shows that different structure levels in UD map to different feature abstraction levels quite reliably within every project. Moreover, certain UD parts like administrative instructions can be omitted for feature derivation since they do not contain feature-relevant data.

Whenever the ITS and UD are maintained systematically, the metadata is most helpful. However, contemporary ITSs do not provide any means to indicate feature relevant information besides categories or tags. They are assignable on the issue level, only and seldom used systematically. For UD no metadata or categorization is used at all, although some wikis provide such features. Here only manual mappings or LE methods can be applied to find SF related information.

Although this study finds that complete SF information can hardly be derived from a single source the thesis still focuses on the ITS for SF detection. Even if an UD is created for a software project, which is not always the case, the UD is often out of date and does not provide technical information that may be important for developers. E.g. in this study the UD of the OFBiz project was unmaintained for years. The wiki that is used as UD neither contains relevant content nor the content of the old UD has been transferred. In contrast, the ITS information is rather complete. Although incomplete information in ITSs was found in this study, this incompleteness was related to features implemented before the ITS was introduced in the project or because the ITS was switched. However, it is likely that such early SFs are very basic and well known by the developers, so that automatic means to extract those SFs are not necessary. If the ITS is switched, both ITSs can be analyzed or the old data can be transferred to the new system.

11

ISSUE TYPES AND INFORMATION TYPES – AN EMPIRICAL STUDY

"You look at where you're going and where you are and it never makes sense, but then you look back at where you've been and a pattern seems to emerge."
- Robert M. Pirsig, Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values.

The empirical study presented in this chapter investigates the second of four RQs defined in Section 1.3 in depth:

> RQ 2: How is NL information categorized, described, and distributed in an ITS?

The previous chapter illustrates that ITS are a fruitful and thorough source for SFs. The previous chapter also identifies that SFs cannot reliably be detected simply by using the issue's category, as this categories and tags are often wrong or not assigned at all. This suggests that the NL data in issues needs to be analyzed more thoroughly. This chapter investigates the NL information in an ITS and how this information is distributed. In particular, the chapter explores (1) what *additional* information besides SFs and SFRs reside in an ITS. This information is necessary to distinguish information that is related to an SF from information that is not related to an SF and it is studied (2) how this information is actually formulated. The latter information is necessary to identify potential patterns or characteristics which can be used for an automatic detection. Fine grained RQs for both aspects are defined in the next section.

In addition, this particular study investigates how information in ITSs is distributed. E.g. which issue types typically contain which information or whether multiple issue categories can occur together in a single issue. Parts of this study were previously published in Merten et al., 2015.

In the next section the study setup is described and RQ 2 as stated above is refined in three fine grained RQs. Section 11.2 presents the results for every RQ. Section 11.3 discusses the treats to validity of and Section 11.4 introduces related work with respect to the investigation in this chapter. Finally, Section 11.5 discusses the implications of categories, descriptions and the distribution of SF information for SF detection.

11.1 STUDY DESIGN

11.1.1 *Research Questions*

The following research questions employ the concepts of *information type* and *issue type*. Issue type is used in this study as introduced in

Section 2.2. An information type describes the information that is carried by one or more sentences in ITS NL data[1]. The term *information type*, as opposed to the frequently used term *knowledge type*[2], is used, because "information is well-formed data with a meaning [, whereas] knowledge resides in the mind" (Schneider, 2009, pp. 11). Hence, categorized ITS NL data should be considered information, whereas the answers to the RQs in this chapter can be considered knowledge with respect to this information.

An issue *issue type* can be considered a context or frame for one or more information types in this particular study (see the leftmost nodes in Figure 11.3), in addition to a concrete annotation to an issue. E.g. the information type *request* can be used in different issue types, such as *feature request*, *request for fix*ing a bug or *request for refactoring*.

With the concepts of issue and information type, the RQ stated in the beginning of this chapter can be refined in the following three research questions:

RQ 2.1 What *issue types* and *information types* are captured in ITS NL data?

RQ 2.2 What is the distribution of different *issue types* and *information types*?

RQ 2.3 Are *issue types* and *information types* used differently in different projects?

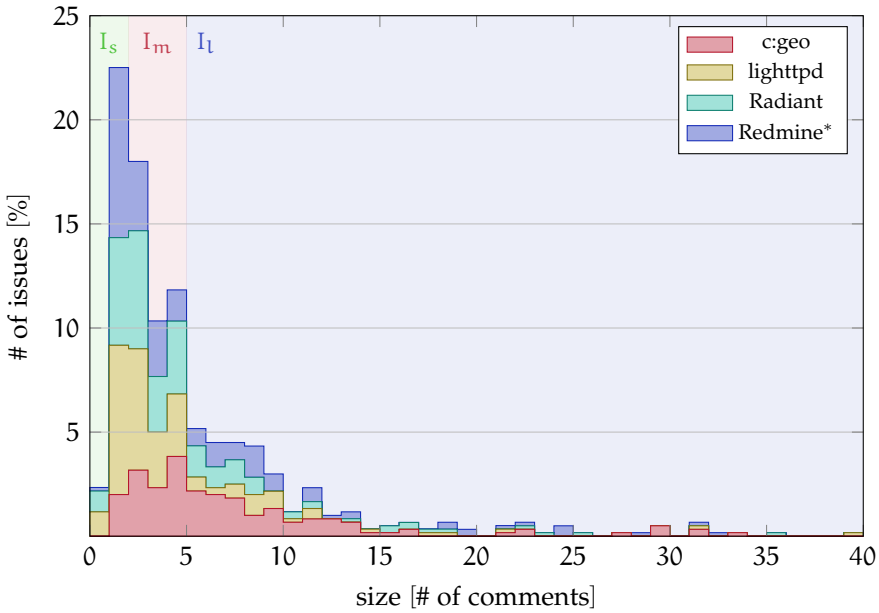
11.1.2 *Data*

The study was conducted on the ITS data from the c:geo, lighttpd, Radiant and Redmine projects, as introduced in Section 8.1. Details of the projects and the ITS data are summarized in Table 8.2. Those projects were chosen for this particular study in view of RQ 2.3: every project has different characteristics with respect to the users of the software, the project type and so on. It will be shown in this chapter, that such characteristics influence the NL data in the issues. E.g. in the lighttpd ITS, issues contain more detailed and technical information than in the c:geo ITS.

All the ITS NL data and metadata that can be retrieved from the respective ITS APIs of every project was extracted, converted into XML, and annotated using GATE for the investigation reported in this chapter. Attachments, as the only exception, were not extracted since attachments (1) are often used for technical information like code snippets, log files, or stack traces and (2) typically use various formats (e.g. JPEG images or Microsoft Word documents), which cannot be converted to plain text easily .

1 See the rightmost nodes in Figure 11.3.

2 E.g. Maalej and Robillard, 2013 use the term "knowledge type" in their taxonomy of API content.



* Figure truncated at 40 comments. 0.8% of the analyzed Redmine issues are longer than 40 comments. The longest issue consists of 173 comments.

Figure 11.1: Issue Sizes.

As suggested in Chapter 10, most requirement-related information can be found in early issues. Thus, the datasets for this study were extracted from the first 1000 issues of every project. Thereafter, the datasets were divided in three sets per project: I_s includes all issues with less than the median number of comments per project (one or two), I_m includes all issues with the median to the mean number of comments (three to five), and I_l includes all issues with more than the mean number of comments (six to 173). Most issues fall in the classes I_s and I_m. Figure 11.1 aggregates the sizes of the extracted issues for every project. The background shading of Figure 11.1 indicates how many issues fall in the $I_{s/m/l}$ classes.

11.1.3 *Analysis Procedures*

This section describes the details of the analysis process. An overview of the process is given in Figure 11.2.

A TAXONOMY OF INFORMATION TYPES: One of the major challenges in this study is to identify the issue types and information types used in the ITS NL data. Although earlier studies analyzed ITS NL data, the authors focused on specific information types, like discussions (Fitzgerald, Letier, and Finkelstein, 2012; Ko and Chilana, 2011) or exclusively on issue types (Herzig, Just, and Zeller, 2013).

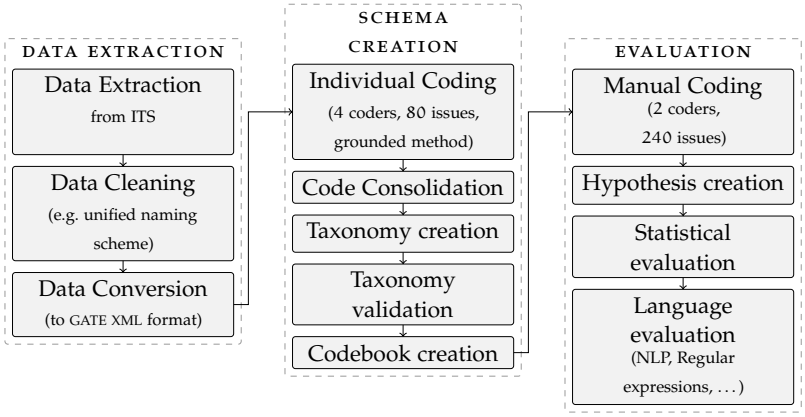


Figure 11.2: Research Process Overview.

In contrast, this particular study provides a broad taxonomy of issue and information types.

To develop the taxonomy, 80 issues were drawn randomly from the I_m and I_s sets. Four authors of the original publication (Merten et al., 2015) coded every sentence of these 80 issues manually using GATE (20 issues per coder). During this process, every coder developed his own taxonomy. The only requirements for the taxonomy development were, that

1. issue and information types should be distinguished, and
2. every sentence should be coded.

Intentionally, every coder used a two-phase schema: the first phase represents the *issue type* (e.g. feature-, bug-, or software development process-related information) and the second the *information type* (e.g. functionality or quality request, clarifying question, or *as-is* descriptions of the situation). To consolidate the individual schemata, a large schema out of all issue and information types from every coder was created. This schema includes 14 issue types from coders c_i ($5_{c1} + 3_{c2} + 3_{c3} + 3_{c4}$) and 127 information types ($45_{c1} + 25_{c2} + 36_{c3} + 21_{c4}$). Then, all synonyms were merged (e.g. one coder named a sentence *suggested solution*, another *solution*, and a third *potential solution*). The remaining information types were discussed. During discussions the coders found that information types are sometimes bound to an issue type and are sometimes neutral, e.g. they can be found in different issue types.

An example is the *as-is* information type. It describes the current status of the software and can be used for feature-related issue types to describe the context of a new SF, or for bug-related issue types to describe the problematic behavior as it is in a certain software version. These neutral information types were added to all relevant issue types. So the *as-is* information type is present in feature- as well

as bug-related issues as shown in the final taxonomy in Figure 11.3 in Section 11.2.

FURTHER ANALYSIS BASED ON THE TAXONOMY After the final taxonomy was consolidated, a coding guideline based on the final taxonomy was created and discussed by all authors. The coding guideline was tested on another 4 issues from the $I_{m/l}$ sets by all coders with an inter-rater agreement of 0.9869 (Cohen's Kappa = 0.4630) for bug-related and 0.8152 (Cohen's Kappa = 0.6786) for feature-related codes and optimal inter-rater agreement for ITS management and not SE related codes. The results of this test run, especially differences, were discussed again. Finally, the descriptions of the coding guideline were updated, such that the coders had a common understanding of every issue and information type.

For the further analysis, another 120 issues were randomly drawn, equally over every project and every set, I_s, I_m, and I_l, to make sure that the samples include all issue sizes and data from all projects. The taxonomy and code book were implemented in GATE

1. to calculate inter-rater agreements,
2. to enforce the use of the taxonomy for deeper analysis, and
3. for text analysis, e.g. to retrieve statistical data or to add additional metadata to the text. Own GATE plugins were developed for the analysis and the Stanford Parser (De Marneffe, MacCartney, and Manning, 2006) was used to annotate the NL.

Overall 120 issues and 3167 sentences, as summarized in Table 11.1 have been analyzed.

	ISSUES			COMMENTS[1]	SENTENCES
PROJECT	OPEN/CLOSED	EXTRACTED	ANALYZED	ANALYZED	CODED
c:geo	425/2829	first 1000	30	322	662
lighttpd	53/272	all	30	204	748
Radiant	425/2829	first 1000	30	218	592
Redmine	4400/9244	first 1000	30	374	1160
Sum			120	1118	3167

[1] Every issue has one title, one description and multiple comments, c. Therefore, c + 60 ITS data fields were analyzed altogether.

Table 11.1: Population, Sample and Coding Sizes.

11.2 RESULTS

The results are presented separated by research question. Section 11.2.1 and Section 11.2.2 discuss issue types and information types sepa-

rately, whereas the project differences in Section 11.2.3 are discussed broadly without this separation.

11.2.1 *Information Types and Issue Types*

Figure 11.3 shows the final taxonomy. The taxonomy includes six *issue types* and 28 *information types*. The information types *ITS Management*, *Clarification*, and *Rationale* are split into six, four, and two subtypes respectively. The white fields *Unclear / Other / Unknown* are used for information that cannot not be classified by the coders.

Figure 11.3: Issue Types and Information Types: A Taxonomy.

ISSUE TYPES The following issue types were discovered in the NL of the ITS data fields:

1. *Feature-Related*: information, related to a new SF or software requirement.
2. *Bug-Related*: software failures and problems.
3. *Refactoring-Related*: software changes that neither affect the functionalities nor the qualities of the software (besides maintainability).
4. *SE Process-Related*: discussions about the general SE process, e.g. if a developer notices that tests should be run more frequently in the project or if documentation should be relocated.
5. *User Problem-Related*: problems that are not related to software development, e.g. a user does not understand a configuration file and asks for help.
6. *Not SE-Related*: anything, that is not related to software engineering activities, such as social interaction between developers.

Feature-Related, *Bug-Related*, and *Refactoring-Related* are relatively obvious issue types for a software development project. However, it is noticeable the ITS was used to discuss changes in the SE development process itself, too. Especially *Non SE-Related* discussions were not expected before the coding started.

INFORMATION TYPES The taxonomy provides detailed information types for the issue types *Feature-Related*, *Bug-Related* and *Refactoring-Related*. Reoccurring information types in feature-, bug- and refactoring-related issues are shown using the same color in Figure 11.3.

Issues usually start with a *summary* (dark blue ■) in the title. In general the summary describes (certain aspects of) the issue and does not form a whole sentence: e.g. "more flexible bandwidth limiting". Some bug-related issues start with an *as-is clarification* (light pink ■) in the title to denote what needs to be fixed and some feature-related issues put the request in the title. For example "provide an infrastructure for content-filtering".

An obvious information type is the *request* (green ■) itself. A *request* can be found in all software engineering-related issue types and is sometimes accompanied with *rationale arguments* (brown ■), emphasizing why the SF should be implemented. In general, however, rationales, are given later in a comment of the issue, e.g. when a user notices that more support is needed to get a SFR implemented, or by other users, who express that the issue is important for them as well. Some *rationales* that give *arguments* and other simply try to up-vote an issue (+1). Especially in SFs the question-/answer-pairs for *clarification* (old rose ■) often occur after the original SFR. If this happens, more elicitation is needed to understand and implement the feature

request. For bug-related information, the coders named the clarification phase *cause diagnostics* (light green ■), since the bug descriptions generally did not need clarification, but the actual cause of the problem had to be found, e.g. by providing reproducibility information. The *as-is* status of the software is often used to describe the problem in a bug. Sometimes even small *user stories*, consisting of one or two sentences, were given to clarify or motivate a new SF. Besides understanding the request and performing the actual implementation, *implementation proposals or solution ideas* (violet ■) are discussed.

Another common information type is *ITS management* (yellow ■), which is used in bug-, feature- and refactoring-related issues. *ITS management* describes NL data with respect to the management of the current issue and is therefore divided in subtypes like referencing other information, closing the issue, mentioning duplicated issues or changing attributes of an issue. Interestingly, this information can be handled by the ITS itself and is therefore not really necessary in the NL data. Some users, however, prefer not to use the ITS mechanisms and express duplicates or references in the NL data fields. In contrast to the ITS management information type, *SE process-related* issue types the SE process of the respective project as a whole, e.g. process improvement.

Not much information with respect to prioritization can be found. Generally, most *scheduling* (magenta ■) is done in a very pragmatic way. E.g. developers comment: "I will look into this tomorrow" or "We should delay this feature". On the other hand, information regarding the *implementation status* (light blue ■) is often communicated, e.g. "this was fixed in update 10" or "I already implemented part X of this issue".

Since RE aspects and SFs of ITS usage are the main interest of this study, the issue types *SE Process-Related, User Problem-Related and Not SE-Related* were not analyzed in detail.

11.2.2 *Distribution of Issue and Information Types*

Reporting in detail on every information type would go beyond the scope of this chapter since many information types did not show patterns in their distribution. However, mappings of all issue and information types have been created during this study. These matrices include, how issue and information types are distributed in the NL ITS data fields (title, description and comment $c_1 \ldots c_n$), combinations of the types, and relations between information and issue types. This data is available for download at the location given in Chapter 7. Additional information on the distribution of issue and information types can also be found in Appendix B. The following paragraphs report on the findings from these mappings and focus on feature-related information, since this is most relevant for Part V.

ISSUE TYPES: Table 11.2 shows how issue types are distributed in every project. The maximum numbers are printed in bold font. Table 11.3 shows the most often occurring combinations of issue types. Qualitatively, as expected, most issues include only *bug-related*, *feature-related*, or *refactoring-related* information.

	c:geo		lighttpd		radiant		redmine	
	occ.	sent.	occ.	sent.	occ.	sent.	occ.	sent.
Bug-Related	13	**300**	**22**	**614**	13	**246**	11	372
Feature-Related	**17**	267	6	58	7	141	20	**674**
Not SE-Related	14	29	19	53	**16**	58	**23**	82
Refactoring-Related	6	48	1	1	9	132	1	11
SE Process-Related	6	10	2	2	3	7	-	-
User Problem	1	1	2	20	1	8	4	11
Unclear or Unknown	2	7	-	-	-	-	2	10
Sum	59	662	52	748	49	592	61	1150

Occ. = Number of overall occurrences and Sent. = Number of sentences

Table 11.2: Distribution of Annotated Issue Types.

However, in some issues aspects of multiple issue types are discussed. Four issues include, bug- and feature-related information. An example is c:geo issue #365. It first describes the *as-is* situation of a bug. In this particular case, a certain color marking (similar to an icon) is missing in the application. Then, *cause diagnostics* of the bug are performed and during this discussion, *implementation ideas* for new features (e.g. user configurable color markings and priority handling for color markings) are proposed.

ISSUE TYPES	OCCURRENCES
BR only	46
FR only	38
RR only	11
FR, BR	4
BR, SE process-rel.	4
user problem	3
FR, user-problem	3
BR, refactoring-rel.	2
SE process rel., refactoring rel.	2

Table 11.3: Top Combinations of Issue Types ($\geqslant$ 2 Occurrences).

Although the SFRs are related to the original bug description, they go beyond the original problem and describe new SFs. This combination could be found in longer issues, only ($1 \subset I_m$ and $3 \subset I_l$), since the discussion starts on one topic and takes some time to drift into other topics. In four issues the *SE process* is discussed as a digression of a *feature-related* discussion. This combination occurred, when con-

ditions of a certain issue have enough generality to discuss the overall *SE process*. However, not a single issue explicitly discusses the *SE process*, only. Another interesting combination are *user problems* and *feature-related* information. This occurs in two ways: Firstly, when a user has a very specific problem, that actually does not affect software changes and then ideas for new features pop up. Secondly, if users suggest a feature and it is already implemented. Then the *feature requests* turns into a *user problem*, e.g. how to find a certain checkbox (Redmine issue 638). In practice these combinations of issue types in a single issue suggest that ITSs should offer better refactoring possibilities, e.g. the extraction of a related issue from one or many comments.

Surprisingly, *Not SE-related* information occurs most often and is at the same time very short (in terms of the number of sentences). Mostly, because of small acts of politeness between the stakeholders. For example, users often thank for "the great project" or for a "fast reaction" with respect to SFRs or bugs. Although the sentiment in ITS NL data was not explicitly analyzed, no coder found any maleficent social interaction in the ITS. Communication is inoffensive and even major bugs are reported in a neutral or friendly way.

INFORMATION TYPES: Table 11.4 shows the 20 most often occurring information types. The full list of information types is shown in Appendix B. Bug-, feature- and refactoring-related *overview*s are used in the title only. However, in bugs sometimes the *as-is* situation is used in the title to describe the bug, and in features and refactorings, sometimes a request (e.g. "please add functionality . . . ") is used instead of a short overview.

Out of the 51 issues with feature-related information 18 include *clarification questions* and 16 *clarification explanations*. These information types are used to detail the SF or related solution ideas. For about 50% of the issues, no further clarification is added. In only six issues, *implementation or solution-related* information is added. Solution ideas are mentioned mostly before any clarification information. One explanation for this is, that the *solution* helped to start a discussion.

In contrast, almost all bugs (58) contain *cause diagnostics* and 28 times *technical information*, like stack traces or log files, is added. For 28 bugs, explicit *reproducibility* information is added so that the bug can better be resolved.

OTHER OBSERVATIONS: Some hypotheses evolved during coding, which could partially be confirmed: Firstly, that *bug-related* issues contained much more *technical information* (e.g. source code, stack traces, and log files) than feature-related issues (H1). Secondly, that *technical information* in *feature-related* issues is posted later than in bug-related issues (H2). Whereas H1 seems to be true (642 sentences of technical information in bugs vs. 58 in features), H2 only partially holds.

ISSUE TYPES	OCCURRENCES	SENTENCES
BR → Technical Information	28	642
BR → Cause Diagnostics → Explanation	31	218
FR → Solution Implementation	28	188
Not SE → Social Interaction	62	167
BR → Cause Diagnostics → As-Is	42	149
BR → Cause Diagnostics → Reproducibility	28	141
FR → Implementation Status	32	110
FR → Rationale → +1	17	105
FR → Rationale → Argument	25	99
FR → Request Functionality	40	91
FR → Scheduling	22	71
BR → Cause-Diagnostics → Question	19	68
FR → Clarification → Explanation	16	66
BR → Implementation-Status	29	63
FR → ITS-Management → Reference	25	63
BR → Solution-Implementation	19	59
FR → Technical Information	6	58
User Problem	9	46
FR → Clarification As-Is	17	44
FR → Clarification Question	18	42
BR → Overview	40	40
FR → Clarification → User-Story	4	35
FR → Overview	32	32
BR → ITS-Management → Reference	19	32

Table 11.4: Most Used Information Types.

Although 20 technical sentences could be found in feature descriptions, none was found in the first comments c_1 and c_2, and most in comments c_3 to c_{11} up to c_{22}, the situation for bugs was similar: 305 technical sentences in the description and another 337 in comments c_1 to c_{12}. However, the combination of early and much technical information may be used as an indication to identify bug-related issues.

In terms of issue lengths, *feature-related* and *bug-related* issues are roughly the same size. Another hypothesis was that *bug-related* issues are shorter, since they need to be resolved quickly and they generally do not involve so many users (H3). Although more bug-related issues (39% $\in I_s$) than feature-related issues (25% $\in I_s$) were very short, the same medium and long issues were found (25% $\in I_{ml}$). However, late comments in feature-related issues are often longer and include more discussions. *Refactoring-related* issues are mostly short (none more than 9 comments). Assumably, these issues are mostly used as a reminder for the developers and no further discussion is necessary.

Besides issues, that contain multiple issue types, as mentioned above, only one wrongly classified issue was found in the researched projects tat use the Redmine ITS. It seems that the number of correctly labeled issues varies between different projects, since Herzig, Just, and Zeller, 2013 researched other projects than this study and found about a third wrongly classified issues. Furthermore, Redmine itself is an ITS, so the users may have a higher discipline to categorize issues correctly.

In the GitHub based projects, most issues are not categorized at all. This is likely because issues do not need to be marked as bug or feature. Although a tag can be assigned, this simply is not done most of the times. So, besides the project, the ITS's architecture seems to influence categorization quality. In practice, the ITS should be chosen and customized according to the needed metadata, since optional metadata (e.g. tags) are often omitted. Furthermore, defaults for metadata fields should be chosen wisely (e.g. if an issue is categorized as bug per default this may never be changed. A neutral category such as 'undecided' could be used as default to prevent such problems. This indicates that the actual category was not yet chosen).

KEYWORD	BR	FR
function	0.04	0.96
implement	0.04	0.91
feature	0.06	0.91
problem	0.87	0.06
exception	0.14	0
cause	0	0.89

Table 11.5: Top TF-IDF Scores.

TF-IDF was used to find potential keywords in the ITS NL data that can be used to classify issues. An excerpt of promising keywords is shown in Table 11.5. For issue types, some obvious keywords, e.g. "bug", "problem" or "feature" can give a strong hint on the correct issue type, but there is still a chance of false positives. E.g. in Redmine the keyword "bug" was found in two feature-related issues and only one bug-related issue. Furthermore, the keywords only occur in a minority of the issues so that the recall is also low. For information types, no keywords could be identified.

11.2.3 *Project Differences*

As shown in Table 11.2, the 30 analyzed issues for each of the projects c:geo, lighttpd, and radiant contain roughly the same amount of sentences ($592 - 748$). The Redmine issues are significantly larger with 1160 sentences. The Redmine project is also older than the other

projects, so one possible explanation is, that features and bugs in Redmine are harder to describe due to the project size. Furthermore, some issues in Redmine were significantly older than in the other projects. Hence, another explanation is that old issues (and especially features, see Table 11.2) get reactivated after some time and need to be discussed again. An example can be found in issue #285 comment #27: "Holy cow, this issue is [...] over six years old, and we're still asking what the feature means?".

Besides different issue sizes, some information types were also used differently. In the lighttpd project, 51% of all sentences are *bug-related technical information*. Redmine and Radiant have around 15% technical information. c:geo on the contrary, includes only 2% technical sentences, even though in this project, the maximum amount of sentences (25%) was composed of *bug-related cause diagnostics* and *reproducibility* information. Our hypothesis is, that this is due to the audience and project type. lighttpd is a server application and bug- as well as feature issues often include configuration snippets. Bugs are also reported by technicians who run a server. Since stack traces and log files are an important artifact in server administration, they are often included in issues, too. c:geo on the other hand, is mostly for ordinary users who want to play the geo caching game. They seem to report bugs as well as feature requests on a higher level of abstraction and do not want to deal with technical details. In practice this implies that the content of a (good) feature or bug report largely depends on the project type and audience.

In addition, scheduling activities, such as prioritization, differ in the projects. In Radiant and c:geo, there is more talk about scheduling (~ 30 sentences) than in lighttpd (7). Redmine (normalized over all issues) is about in between. The high amount of explicit scheduling mentioned in the ITS NL data is likely due to the fact that the GitHub issue tracker does not provide the same flexible mechanisms for scheduling as the Redmine ITS does. E.g. in the Redmine and lighttpd projects, the *Milestone-* and the *Roadmap-Feature* of the Redmine ITS are extensively used and issues get prioritized and scheduled by assigning them to a certain milestone or software version. In the GitHub based projects, this needs to be communicated in the NL[3]. Still, it can be observed that scheduling is mentioned in the NL of all projects, although the Redmine ITS offers extensive scheduling features.

3 Please note that GitHub added a Milestone feature while this analysis was done. Although this feature is not as flexible as in Redmine at the time of writing, more flexibility might be added.

11.3 THREATS TO VALIDITY

CONSTRUCT VALIDITY The taxonomy was created with a grounded technique. To ensure the validity of the taxonomy, all coders created and discussed a coding guidebook that was used during the rest of the study. However, the taxonomy is not very fine-grained and may therefore not be appropriate for other research without modification or addition.

The content analysis was done by two coders without redundancy and inter-rater agreement. This threat was minimized by (1) the use of test issues (gaining a high inter-rater agreement, considering that every possible NL sentence was coded), (2) extensive discussions on coding and (3) the use of a coding guidebook. Furthermore, the coders worked in the same room, so that they could ask each other whenever a sentence was unclear.

EXTERNAL VALIDITY Data from four different OSS projects was sampled. These projects represent different characteristics of software development projects as discussed in Section 11.1.2. The results can be transferred to similar project settings. However, only 30 issues per project could be annotated due to limited resources. Hence the results are not statistically significant. Other factors influencing the use of ITSs and ITS NL data need to be investigated in depth, to make results transferable between different projects.

11.4 RELATED STUDIES

Kunz and Rittel, 1970 first introduced the concept of issues in the area of cooperatives. Their paper, published in 1970, already discusses many prevalent concepts in modern ITSs, especially the possibility of discourse, arguments and counter arguments. The information types presented in this chapter narrow down these concepts or arguments and counter arguments in the context of SE, making these abstract concepts tangible.

There is a related block of research that tries to improve ITSs by observing OSS developers or the analysis of ITS data. Zimmermann et al., 2009 discusses how bug tracking systems can be improved from an architectural standpoint. They point out that improvements need to be tool-, information-, user-, and process-centric. Just, Premraj, and Zimmermann, 2008 derive recommendations for improving ITSs from developer interviews. Lotufo and Czarnecki, 2012 focus in improving bug reports and suggest that game mechanisms could encourage ITS users (Lotufo, Passos, and Czarnecki, 2012). Finally, Baysal, Holmes, and Godfrey, 2013 focus on personalizing issue tracking systems for different roles, similar to the idea of viewpoints in software architecture (Finkelstein and Sommerville, 1996).

All this research focuses on ITS users and provides suggestions to solve some of the problems certain user groups face in ITSs. In contrast this study provides insights from the information-centric view on ITSs.

Bertram et al., 2010 study communication and collaboration of issue tracking in small, collocated teams. They focus on the social nature, not on the SFs themselves. They state that ITSs are not designed as requirement documentation systems and instead focus on achieving a task (e.g. by tracking the progress of an issue) which fosters communication in the team. The taxonomy developed in this chapter can complement similar studies to state information types explicitly and thus provide not only qualitative but also quantitative insights.

One of the first studies, that includes information types in SE was provided by Kitchenham et al., 1999. They define an ontology that includes multiple SE aspects, such as product and process information of software maintenance. The ontology is defined on a document level and does not dig deeper into the content of these documents. Herzig, Just, and Zeller, 2013 present different categories for *bug-related* issue types. Ko and Chilana, 2011 analyze discussions in bug reports in depth. They provide detailed categories for the discussion elements.

No study, however, analyzed all ITS NL data on a per sentence basis. The provided taxonomies or categories of other studies are always very specific for a certain aspect of ITS NL data or ITS usage. The taxonomy defined in this chapter can serve as an umbrella for a broad range of information types and details can be added using particular taxonomies from related work.

11.5 IMPLICATIONS FOR SOFTWARE FEATURE DETECTION

This study presents a taxonomy of issue and information types in ITSs. In addition it analyzes the ITS NL data of 120 issues in depth. In about 50% of the feature requests that were analyzed no further clarification of the request was needed and also in only about 50% a solution was described. This implies that often a single sentence or a small description can be sufficient to implement a feature.

The study also found that the information types in ITS NL data are influenced at least by the project type, audience, and even the technical capacities of the used ITS. However, other issues include clarifications and solution ideas related to the SFRs. Thus SFR detection should be conducted on the sentence level in the best case. If this is not feasible, the data field level should be considered, as it still delivers precise information on the SFR location and related clarifications or solutions.

With the analyzed data of 120 issues, it was almost impossible to identify keywords that can be used to classify information types. Keywords should be compiled again on a larger dataset. Although no

clear communication patterns could be found, SFRs are often formulated using a similar wording. Thus a ML algorithm might be able to create a statistical model from these similarities. However, SFR detection cannot rely on certain sequences of information types and no order could be identified within the information types. Thus the relatedness of clarifications or solution ideas to an SFR is almost impossible to judge automatically. Hence a LE (rule-based) approach is likely insufficient to detect SFRs.

Finally, feature-related information can be found in any part of the issue. SFRs are common in issue titles and descriptions, but they can occur in comments, too. This implies that an SFR detection should be carried out on all data fields. In practice issues are usually categorized according to their title or description. Hence SFRs in issue comments cannot be found based on the ITS meta data, even if it is available and well maintained.

12 SUMMARY OF INVESTIGATION STUDIES

"It is not possible to separate 'understanding the problem' as a phase from 'information' or 'solution' since every formulation of the problem is also a statement about a potential solution." - Kunz and Rittel, 1970

The previous two chapters researched SFs in ITS data. In the following the main findings relevant for SF detection are summarized. These main findings will be picked up in Part V of the thesis to build a method for SF detection.

SOFTWARE FEATURES IN ISSUE TRACKING SYSTEMS The first study found that ITS can be considered a reliable source to detect SFs, at least compared to the UD. Most importantly, the study found that the ITS has the most fine grained information on SFs and that the available information is usually up-to-date in contrast to the UD. Even if the ITS does not contain the complete feature information as for example in the Radiant project, this had a simple reason: the ITS was not used since the beginning of the project, hence no data could be recorded.

Digging deeper in the feature-related information, the study in Chapter 10 found that issues describe SFs on different abstraction levels. Although the ITS usually has a low abstraction level this is not guaranteed. E.g. in the Mixxx and OFBiz ITSs, no requirement level feature descriptions could be found, but the Radiant ITS contains about 35% feature descriptions on the requirement level.

Furthermore, relations between issues with feature descriptions could be identified. Some of these relations exist since course grained features had to be refined, others had a technical nature. Interestingly, these relations are seldom expressed as explicit trace in the ITS. Thus trace information on related SFs or SFRs cannot be extracted easily.

Finally, the study found two problems with issue types: (1) issue types are only assigned if the issue type is an obligatory field in the ITS but not if the issue type is optional, and (2) a single issue type is not always sufficient to categorize an issue as many issues discuss feature- and bug-related information together. This implies that the issue type, even if assigned correctly, cannot be used to detect SFRs automatically.

ISSUE TYPES AND INFORMATION TYPES The study in Chapter 11 created a comprehensive taxonomy of issue- and information types. Out of these information types some are clearly related to SFs. The most important information related to SFs are the SFR, the *request for qualities*, *clarifications* with respect to the SF, and *implementation proposals and solution details*. This information is closely related to the description of an SF. However, with respect to SF detection, other in-

formation types can be considered noise in the ITS. Examples for information types that do not describe the SF are social interaction, *ITS management* related information, *scheduling* information, or questions regarding the current *implementation status*.

Furthermore, the study confirmed another important finding from the first investigation study: feature related information can be found in all parts of the issue, not only title and description. This is another reason, why an SF detection should be approached on finer grained scopes than the issue level. Although the detection should focus on the SFR as central artifact describing the SF, other information types such as *clarifications* should also be considered for a detection.

Finally, the study did not identify clear patterns in the NL of an issue or in the information distribution. However, many NLP-based approaches rely on such patterns to identify information by the use of rules. E.g. Vlas and Robinson, 2012; Vlas and Robinson, 2013 use rules to detect requirement descriptions automatically. The studies in this part disclosed that such an approach cannot be adopted in the context of SF detection for ITS data. At least not for the researched projects.

Part V

ITSOFD: THE ISSUE TRACKING SOFTWARE FEATURE DETECTION METHOD

The *Issue Tracking Software Feature Detection Method*, a solution design to detect software features in ITS data, is described in this part. First, the overall method is derived by analyzing findings from the previous part and their implications on a solution design. Together with the description of the method, related work is presented. Thereafter, each chapter represents a solution study validating a major part of the method.

The first study validates a solution for separating technical artifacts from natural language. This solution can be used to preprocess issues in the following two studies. Second, the main study of this thesis is presented: a machine learning based method to detect expressions that are usually used to describe software features in issues. The evaluation includes different machine learning techniques, data preprocessing techniques and evaluation measures. A third study discusses how well related issues can be detected using information retrieval techniques.

13 DESIGNING ITSOFD

"Issues are specific to particular situations; positions are developed by utilizing particular information from the problem environment."
- Kunz and Rittel, 1970

The previous chapter summarizes many insights from the two problem analysis studies in Chapter 10 and Chapter 11 with respect to ITS data and especially with respect to NL in issues. These findings are now used to create an applicable method for SF detection.

The next section derives implications for a solution design. Then Section 13.2 introduces ITSoFD and finally Section 13.3 addresses other work related to this design.

13.1 CHALLENGES IN SOFTWARE FEATURE DETECTION

Table 13.1 summarizes the findings of the investigation studies presented in Part iv that impact the solution design. The implication on the solution design (I n) of each finding (F n) is discussed on the right hand side.

FINDINGS	IMPLICATIONS
F1 Deriving a complete SF set semi-automatically will be very difficult and success will depend on the project and the quality of the ITS data.	I1 SF detection cannot reliably derive a complete feature set from ITS data due to completeness issues. That said, recall or the F1-Score should be maximized depending on the users intention.
F2 SF information is likely contained in only few issues of an ITS and can be found in title, description, and comments.	I2 SF detection should be applied to all ITS data fields including the complete set of comments.
F3 The quality of issue type information depends largely on the project and the way issue types are implemented in the ITS (obligatory issue types vs. optional issue tags).	I3 Issue types cannot be used as (the only) indicator to detect SFs in ITS data.
F4 Issue types are implemented on the basis of issues in modern ITSs. However, a categorization per ITS data field is not possible (i.e. comments cannot be categorized explicitly).	I4 SF categories are not available for SF detection in ITS data fields (e.g. if an SFR appears in a comment to the issue).
F5 Project related factors like project type, project audience, project maturity, or the ITS used in the project influence the information types that reside in the ITS and how they are structured.	I5 It is important to evaluate to what extend SF detection needs to be implemented/tuned per project and whether a method can be applied to different projects.

Table 13.1: Implications of Investigation Studies for the Solution Design (Continued on the Next Page).

FINDINGS	IMPLICATIONS
F6 Keywords, communication patterns, or reliable rules to detect SFs could not be identified in the problem analysis studies.	I6 A reliable SF reliable detection cannot be implemented using rules or simple LE, only.
F7 The NL in issues is not always used perfectly. Typos, or grammatical errors can be found rather often as the ITS users are seldom native English speakers and there is usually no need to correct issue descriptions if they are understandable by the developers.	I7 Techniques such as grammatical parsing depend on well written NL. Although such techniques can be applied, one cannot rely on their results. That is another reason why LE rules should not be used in SF detection.
F8 Issues consist of NL mixed with other data such as code snippets, stack traces, or log file excerpts. To disclose technical information from NL in ITSs the user needs to apply special XML tags oder formatting instructions which are not always applied.	I8 Depending on the detection method and dataset this technical data might influence the detection results (e.g. if clustering or ML algorithms are used).
F9 On the one hand a small part of the NL in an issue often contains the SFR. On the other hand long sentences or clarifications are often necessary to describe the SF.	I9 SF detection should ideally be flexible enough to deal with this situation. It should be applicable to other SF related information types such as clarifications. It should be able to find information distributed across the ITS in multiple issues.
F10 Other issues, such as bugs or refactoring tasks are also tracked in the ITS and are often related to SFs.	I10 Traces to information from other issues that are related to a specific SF can be important.

Table 13.1: Implications of Investigation Studies for the Solution Design (Continued).

The findings and implications from Table 13.1 are important requirements for ITSoFD, described in the following. Therefore, the next sections include references to every finding and implication.

13.2 SOLUTION DESIGN

ITSoFD basically consists of four parts. First, the data is extracted from the respective ITS APIs and converted to the GATE XML format. Second, the data is preprocessed depending on the technical artifacts and NL data present in the issues. Third, NL describing an SFs is detected using ML techniques. Fourth, issues that contain related information to those SF components are identified using IR techniques.

In the following, the term SFR is used as a synonym for "NL describing an SF". This NL can be a request for an SF, a clarification with

respect to an SF, or a solution idea for implementing an SF. SFRs are elaborated in more detail in Section 13.2.3.

The flowchart in Figures 13.1 and 13.2 gives an overview of the ITSoFD method. Both figures are complemented by Table 13.2, which summarizes the obligatory and optional manual activities and the decisions to be made whenever ITSoFD is applied. The left hand side of the table shows index and name of the activity or decision. These indices and names accord with the activities[1] and decisions in Figures 13.1 and 13.2. The right hand side of the table gives a longer description of the activity or decision.

13.2.1 *ITSoFD Phase 1: Issue Extraction*

This section describes Phase 1 in Figure 13.1. First, the ITS data is extracted from the respective ITS APIs. The extraction is realized as an adapter pulling the issues from the ITS API and transforming the extracted data in an XML format processable by GATE. This way GATE's annotation editor can be used in Phase 2 to annotate issues as training data[2].

During this transformation all ITS data fields, the ITS metadata and the data field metadata as well as links to the original issues should be preserved.

13.2.2 *ITSoFD Phase 2: Issue Preprocessing*

This section describes the left hand side of Phase 2 in Figure 13.1. Now that the data is available in a standardized XML format after Phase 1, every ITS data field needs to be preprocessed. As issues may contain a mixture of NL and technical information (F/I 8), two decisions need to be made to properly configure the necessary amount of preprocessing. First, it needs to be determined whether the ITS contains technical information (see D_1 in Table 13.2 and Figure 13.1). If it does not, or if there is only a negligible amount of technical information in the ITS, then there is no need for separating the technical information. If a non-negligible amount of technical information resides in the ITS, the technical information should be separated from the NL for two reasons:

1. Preprocessing techniques do not work as intended on technical data (e.g. removing English stop words in source code does not make sense.).

1 In the flowchart notation the activities are modeled as processes.

2 And evaluation data (e.g. the gold standard) for the according experiments in Chapter 15. Note that any tool can be used to annotate the training and evaluation data and GATE was used for convenience.

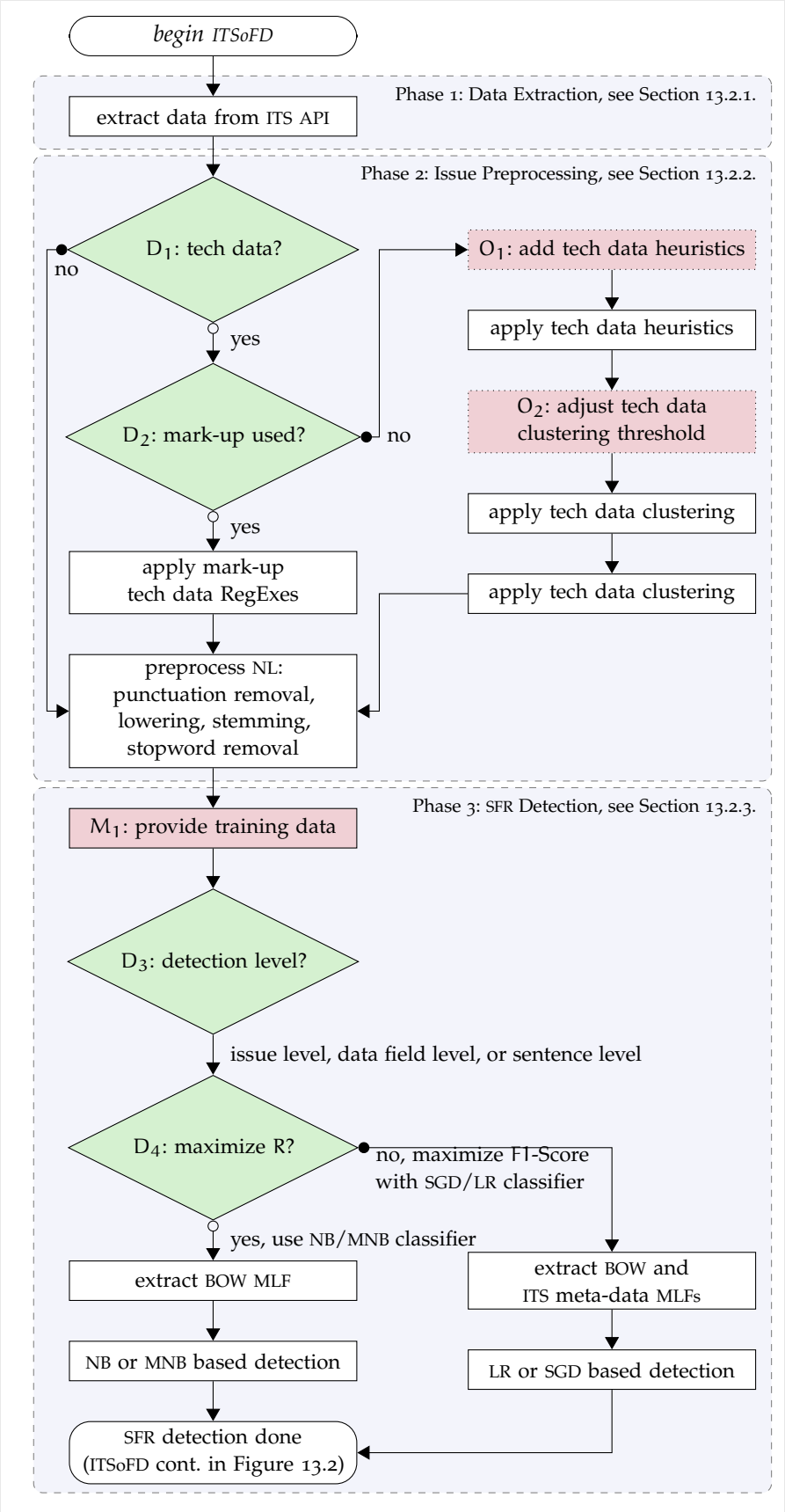


Figure 13.1: ITSoFD Overview (Part 1).

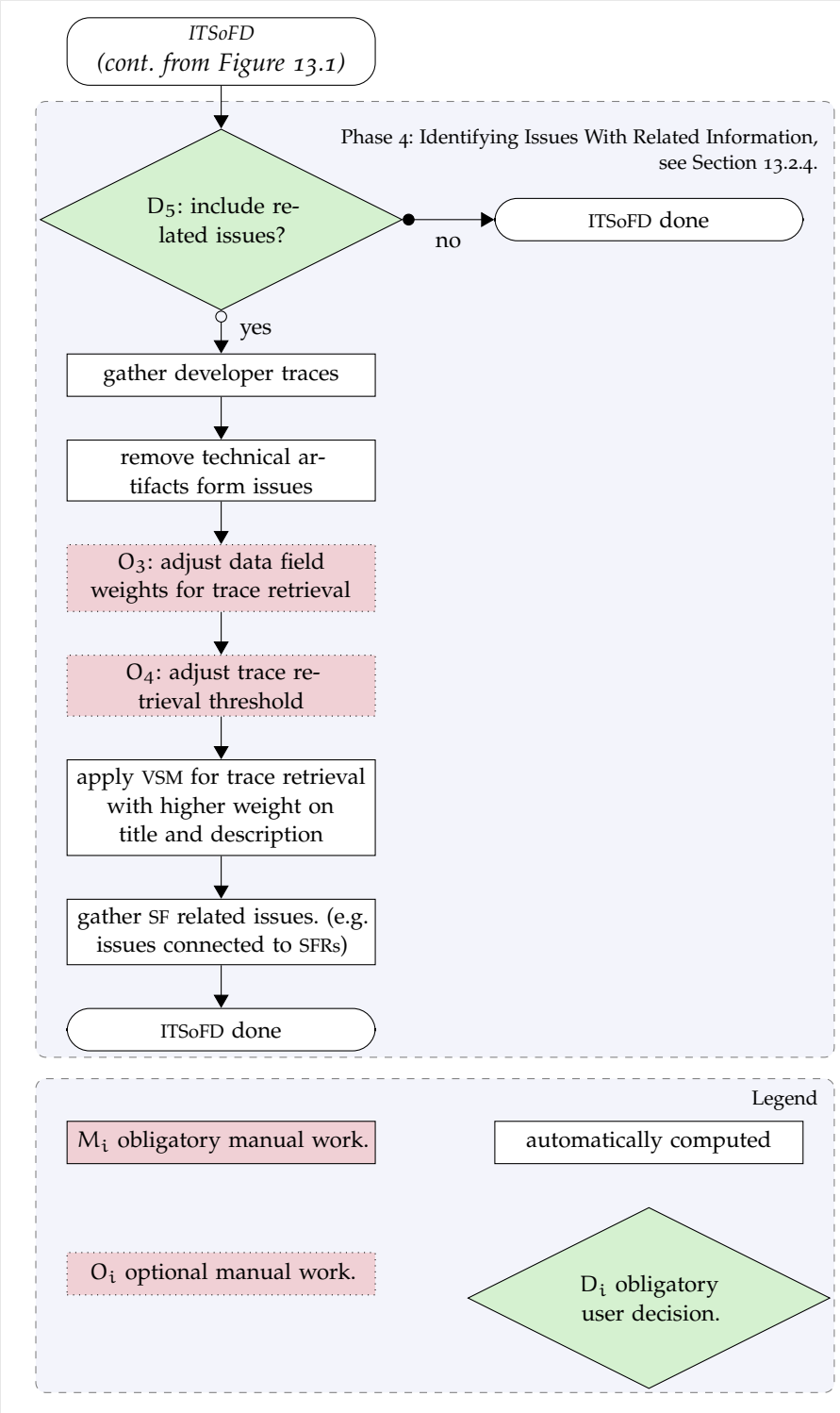


Figure 13.2: ITSoFD Overview (Part 2).

DECISION / TASK		DESCRIPTION
D_1:	tech data?	Do issues in the ITS contain a non-negligible amount of technical data? E.g. do issues contain multi-line code snippets, log files, or stack traces?
D_2:	mark-up used?	If the ITS contains technical data, is this technical data marked-up properly? Marked-up technical data can easily be identified as it is properly formatted in the ITS.
D_3:	detection level?	The user needs to decide whether SFR components should be detected on the issues level, data field level or sentence level (e.g. should issues be classified as a whole or every title, description, and comment or every single sentence, see Section 5.3 and Section 13.2.3).
D_4:	maximize R?	Should ITSoFD SFR detection maximize the recall or balance precision and recall? E.g. does the user want a rather complete list of SFRs that might contain false positives, too, or is completeness not an issue but the reported SFRs need to be as correct as possible (see Section 13.2.3)?
D_5:	include related issues?	Should ITSoFD report on issues that are related to the SFRs (see Section 13.2.4)?
M_1:	provide training data	The user needs to input 100 or more examples for each SFR component that should be detected. Components can be one of requests, clarifications, or solution ideas (see Section 13.2.3).
O_1:	add tech data heuristics	The user can add project specific heuristics for the separation of technical data or remove heuristics that are not relevant for the project (e.g. keyword lists for programming languages that are not used in the project, see Section 13.2.2.1.)
O_2:	adjust tech data clustering threshold	The user can experiment with the clustering threshold to improve the results for technical data separation (see Section 13.2.2.1).
O_3:	adjust data field weights for trace retrieval	The user can override the default weights for trace retrieval (see Section 13.2.4).
O_4:	adjust trace retrieval threshold	The user can adjust the threshold how similar issues need to be, so that a trace is created ($trace_t(i, i')$, see Equation 4.5 in Chapter 4 and Section 13.2.4).

Table 13.2: Summary and Descriptions of Manual Activities in ITSoFD.

2. The following steps in ITSoFD perform better if technical data is removed (see Section 13.2.3 and Section 13.2.4).

If the technical information is properly marked up (see D_2 in Table 13.2 and Figure 13.1), regular expressions can be used to separate the technical data form the NL[3]. However, if the technical information is not properly marked-up, ITSoFD proposes to apply a mixture of heuristics and hierarchical clustering to separate one from another. Details of this separation method are described in the following section and the method is validated in Chapter 14.

13.2.2.1 *Separating Natural Language and Technical Data*

This section describes the right hand side of Phase 2 in Figure 13.1. To separate technical data form NL that is not properly marked-up three steps are executed:

1. The document is segmented by white space tokenization.
2. Every token is classified as NL or technical information by multiple heuristics.
3. Accumulations of technical information and NL are united by bottom-up hierarchical clustering[4] to improve the detection rate of the heuristics.

These steps work independently of the NL in which the issue is written, so that it can be applied to projects in any language, and the programming language(s) that represent the technical data in the document.

Heuristics can be considered a rather basic method to identify technical information such as code snippets, stack traces or log files within NL documents. They usually do not retrieve good recall or precision rates, because heuristics do make mistakes. E.g. a heuristic based on programming language keywords such as "if, case" or "interface" would mark those words as code even if they occur in NL texts.

HEURISTICS FOR TEXT DETECTION After a manual analysis of multiple documents, some patterns could be found within technical information. The detection of these patterns is implemented using the following heuristics, which are used for the initial detection of technical information within the NL:

1. *Keyword Detection*: the document is scanned for keywords that occur in different programming languages, such as `class`, `if`, `switch`, ... The list of keywords is compiled from the ten most popular languages according to the TIOBE index[5].

3 Appropriate regular expressions for the GitHub and Redmine ITSs can be found in Appendix D.
4 Also called agglomerative hierarchical clustering.
5 `http://www.tiobe.com` accessed on 14th September 2014.

2. *Fuzzy Line Equality Detection*: the equality of two or more lines is calculated based on the words these lines have in common. The words are compared as a set, i.e. independently of their position. This way, lines with similar patterns can be found if words are swapped or positions are shifted, which is usually the case for log files or stack traces.

3. *Fuzzy Line Equality Detection*: the equality of two or more lines is calculated, based on the words these lines have in common. The words are compared as a set, i.e. independently of their position. This way, lines with similar patterns can be found, even if words are swapped or positions are shifted, which is usually the case for log files or stack traces.

4. *Regular Expression Detection*: Regular expressions that search for occurrences of special characters (e.g. parentheses or asterisks), indentations that usually occur in source code, or words written in a special formatting such as `CamelCase`, or `under_score` are implemented.

Each heuristic is applied to the text and marks matching tokens as technical information. This step can be compared to using a highlighter on the document. Since several heuristics are in use, text can potentially be marked multiple times. This, however, has no influence: i.e. no weight is added to the tokens marked by multiple heuristics.

Optionally, the heuristics can be customized according to the analyzed repository or new heuristics can be added (see O_1 in Table 13.2 and Figure 13.1). For example additional regular expressions can be added to detect certain code snippets. Similar to the regular expression detection, the *Fuzzy Line Equality* heuristic can compare n lines. Tweaking this setting can be useful if a log file or a stack trace has a similar format in general but includes different looking lines in between. Experiments using different settings for this parameter with $n \in \{1, \ldots, 10\}$ showed that $n = 3$ is a reasonable default, which is used in the evaluation, too.

CLUSTERING AND CLASSIFICATION Although the heuristics detect parts of the technical information, they generally fail in two ways: firstly, they mark parts of the NL as technical information as described above. Secondly, they miss some technical information (e.g. because a NL string occurs in the code). This is illustrated in Figure 13.3: the heuristics detect the Java keyword "interface" and the surrounding quotation marks erroneously as code.

To overcome the weakness of heuristics, bottom-up clustering (Manning and Schütze, 1999, pp. 500) is applied to the tokens of the text. In simple terms it 'fills the gaps' that are left out by the heuristics. The clustering presumes that smaller, differently classified clusters c_{diff}

Initial tokenization and heuristics:
`Natural language text can contain keywords such as " interface ". However, it should not be detected as code.`

First clustering step:
`Natural language text can contain keywords such as " interface ". However, it should not be detected as code.`

Second clustering step 'swallows' the smaller cluster:
`Natural language text can contain keywords such as " interface ". However, it should not be detected as code.`

where ◯ marks an NL cluster and
◯ marks a technical artifacts cluster.

Figure 13.3: Example for Clustering Steps.

between clusters of the same class c_{same} have been classified wrongly by the heuristics beforehand. The those smaller clusters c_{diff} are absorbed when the clustering algorithm merges the clusters c_{same}. Two clustering steps in Figure 13.3 illustrate how the clusters evolve.

Details of the clustering algorithm are shown in Figure 13.4. Here the influence of different cluster similarities with respect to cluster merging is illustrated in line seven and eight. Choosing an appropriate cluster similarity significantly influences precision and recall, which is confirmed by the empirical study in Chapter 14.

Initially, every token t_i in Figure 13.4 is its own cluster $c_i \in C$, where C is the set of all clusters. In every iteration, two clusters in C are merged based on their similarity $\text{sim}(\chi)$:

$$\text{sim}(c_i, c_j) = \begin{cases} 1 & \text{if intersect}(c_i, c_j) = 1 \\ 1 - \frac{\text{dist}(c_i, c_j)}{\sum_{c \in C} |c|} & \text{if class}(c_i) = \text{class}(c_j) \\ 0 & \text{otherwise} \end{cases} \tag{13.1}$$

Thus, similar clusters can be merged even if they are not close to each other. Every time two clusters c_i and c_j are merged, the classification of the newly created cluster c_{ij} is determined by the ratio R of characters marked as technical information. Two clusters are considered technical information for $R \geqslant 0.7$. This settings yields the best results for all evaluated documents in the evaluation study presented in Chapter 14. The $\text{class}(c_i)$ function in Equation 13.1 outputs the classification of a cluster with respect to R. The $\text{dist}(c_i, c_j)$ function in Equation 13.1 returns the amount of tokens between c_i and c_j.

The clustering stops, if the minimum similarity s_{min} cannot be reached and the text is classified so that every token is either NL or technical information.

Given: a set $\chi = x_1, \ldots, x_n$ of objects and
a function sim: $P(\chi) \times P(\chi) \rightarrow \mathbb{R}$ (see Equation 13.1)

```
for i := 1 to n do
    c_i := {t_i}
end for
C := {c_i, ..., c_n}
j := n + 1
while |C| > 1 do
    (c_n1, c_n2) := arg max_(c_u,c_v)∈C×C sim(c_u, c_v)
    if sim (c_n1, c_n2) < s_min then
        stop
    end if
    c_j := c_n1 ∪ c_n2
    C := C\{c_n1, c_n2} ∪ {c_j}
    j := j + 1
end while
```

Figure 13.4: Bottom-up Clustering, Adapted from Manning and Schütze, 1999, pp. 500.

The empirical study in Chapter 14 shows that a s_{min} of 0.9 represents a safe default value for the clustering step. Nevertheless, varying this value in the range of 0.8 and 0.99 may improve precision and recall depending on the project (see O_2 in Figure 13.1 and Table 13.2). However, the empirical study also shows that values very close to 1 have a negative impact on the results[6], which is why a value above 0.9 is not recommended without further experimentation.

13.2.2.2 *Standard Preprocessing Techniques*

This section describes "preprocessing NL" in Phase 2 of Figure 13.1. At this point, the NL inside every issue is properly identified and separated from technical data. To summarize, this was done because,

1. only a negligible amount of technical data was present in the issues, so that all text can be considered NL, or because
2. the technical information was properly marked up and could be separated using regular expressions (D_2), or because
3. the steps described in Section 13.2.2.1 were applied (D_2).

Now the following three standard preprocessing techniques are applied on the NL data of every issue:

- Lowering

6 Confer Figure 14.2 on page 129 for a good range of values for s_{min}.

- Stemming
- Punctuation Removal
- Stopword Removal

The above preprocessing techniques have the highest positive and the least negative impact for all further steps in ITSoFD as confirmed by the two empirical studies in Chapters 15 and 16. These studies evaluate the influence of all preprocessing techniques mentioned in the Background Section 3.2 on page 24.

After Phase 2, all issues, comments and ITS metadata are extracted. Depending on D_1, the NL is separated from technical data and all ITS data fields that contain text are preprocessed. This status is summarized in Figure 13.5.

13.2.3 *ITSoFD Phase 3: SFR Detection*

This section describes Phase 3 in Figure 13.1: the actual SF detection.

UPFRONT DESIGN DECISIONS As a first solution idea, a rule-based LE method, similar to the one successfully applied to RAs (Vlas and Robinson, 2012; Vlas and Robinson, 2013), was considered for SFR detection. However, as no clear communication patterns (F/I 6) can be found in issues and the NL contains much flaws and grammatical errors (F/I 7), this design could not be pursued any further: rules rely on a precise and flawless NL and especially on the accurate use of grammar. Therefore, the SF detection approach in ITSoFD needs a technology which can adapt to such flaws: as introduced in Section 5.2, ML is such an adaptive technology that can learn from examples.

As discussed in F/I 2, the method needs to be applied to all ITS data fields individually and should detect the sentences containing the SFR in the best case. However, many MLFs that can be derived from an ITS are not available on the level of data fields or sentences, so ITSoFD uses two concepts to include MLFs from other levels in the classifier models: (1) inheritance of MLFs from broader levels of detail, and (2) aggregation of MLFs from narrower levels of detail. Both concepts were introduced in detail in Section 5.3 on page 36.

DETECTABLE INFORMATION TYPES In theory any SF related information, such as the information types developed in Chapter 11, can be detected by ITSoFD's ML-based approach. However, to evaluate ITSoFD, training and evaluation data has to be created manually which takes time and effort[7]. Hence ITSoFD was evaluated with three information types as described in Chapter 15:

7 E.g. the training and evaluation data used in Chapter 15 took about two business days per annotator or eight business days in total.

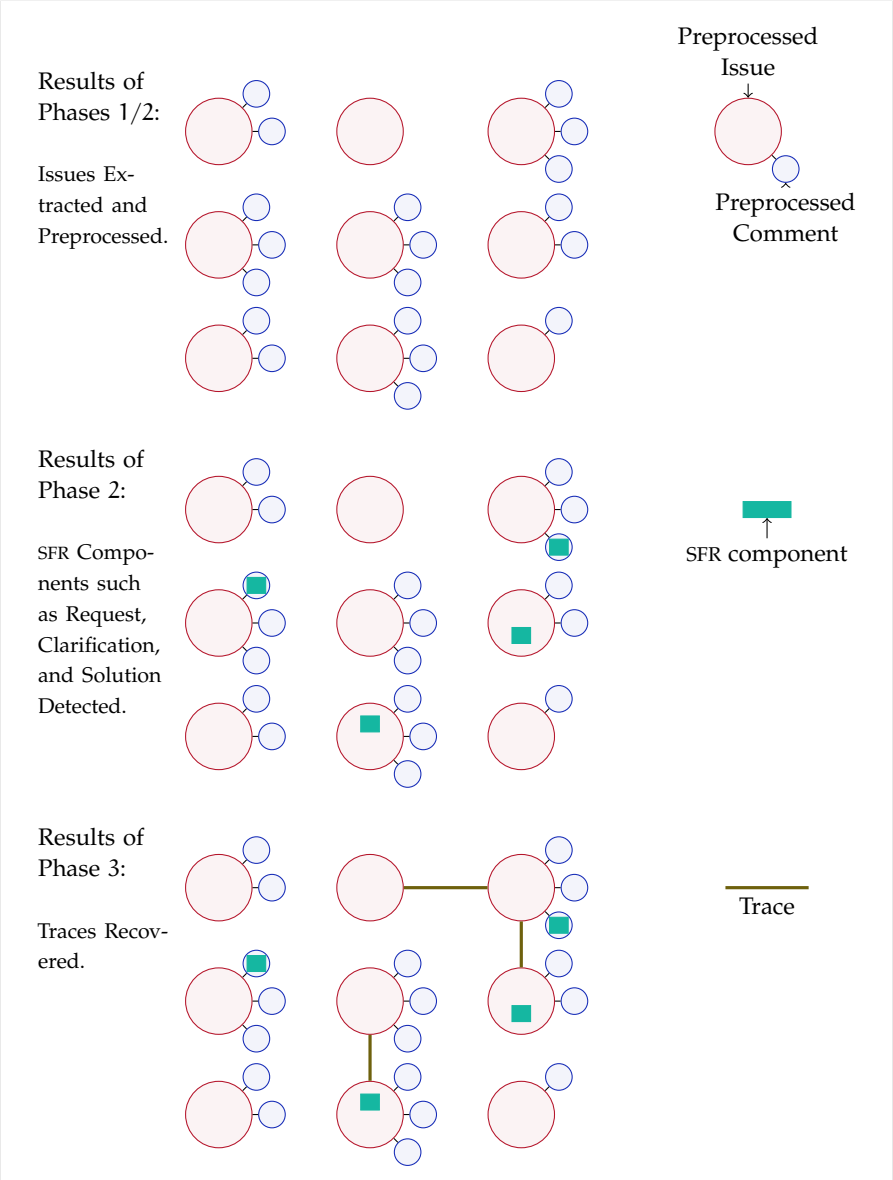


Figure 13.5: ITSoFD Results for Each Phase.

1. the SF request itself,
2. clarifications that are related to a SF request, and
3. solution ideas that are related to a SF request.

In general, various factors influence the SFR detection. However, information types that are expressed using certain language patterns can be detected best. For example the requesting part in an SFR often follows a pattern: "I/we would like to have ...", "could you please add ...", "you/we should add ..." or "the software should ...". The evaluation in Chapter 15 shows that SF requests and clarifications can be detected best, whereas a detection of solution ideas does not work well. This is consistent to the claim above: solution ideas are often very long and the NL seldom follows patterns as solutions are very individual compared to requests. Thus only a detection of SF requests and clarifications can be recommended for ITSoFD without further experimentation. Both information types are important components to understand an SF as discussed in Chapter 11[8]. See Chapter 15 for more details with respect to the evaluation.

In the following it will not be distinguished explicitly between these information types. The umbrella term SFR will be used substitutionally for SF requests and clarifications alike.

TRAINING DATA The user needs to annotate at least 100 SFRs on the level of sentences to train the classifiers, first (see M_1 in Table 13.2). As a rule of thumb: the more examples are provided, the better the detection. However, the evaluation in Chapter 15 shows that 100 annotations is a reasonable minimum for SFR detection. Maalej and Nabil, 2015 compare classifier results using different amounts of training data in a similar ML context. On the one hand the classifiers in their experiments continuously improve up to the maximum provided amount of 300 training instances. On the other hand more than 200 training instances impede the classifier training time significantly. Thus between 100 and 200 instances are recommended. To provide the training data for ITSoFD, GATE's annotation editor can be used for convenience. Figure 13.6 shows a screenshot of annotated issues in GATE.

Training data is a prerequisite for all ML approaches. Thus it is preferable that a model, once trained, can be applied to another dataset or project as well (see F/I 6). The evaluation in Chapter 15 shows that reuse of ML models if possible. However, the projects should use a similar wording to describe issues and models cannot be transferred arbitrarily. Hence it is recommended to validate results whenever ML models are reused across projects. This is discussed in more detail in Section 15.4.

8 Refer to Figure 11.3 on page 88 for a complete overview of information types.

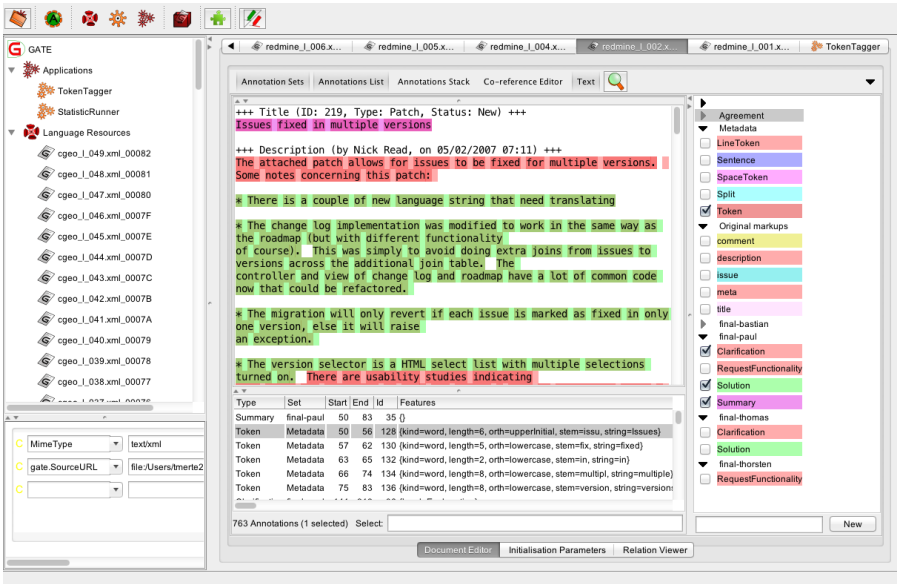


Figure 13.6: Text Annotation with GATE.

DETECTION LEVEL We know that SF related information can differ in size and that multiple information types can be considered related to an SF (F/I 9). Hence ITSoFD suggests three different ways to detect SFRs. The user should choose whether the detection should take place on the level of issues, the level of ITS data fields or on the level of sentences. On the issue level every issue is classified whether it contains an SFR, on the data field level every data field is classified and on the sentence level every sentence. The evaluation in Chapter 15 shows, however, that a detection on the sentence level is not recommended due to a low precision and recall, whereas issue and data field level can both be recommended.

IMPORTANCE OF RECALL AND PRECISION Once the training data is provided, models for various classifiers can be build with this data. The evaluation in Chapter 15 shows that different classifiers perform differently well depending on the user goal (cf. F/I 1). Some classifiers, such as NB and MNB detect almost all potential SFRs, thus having a high recall. Others, such as SGD and LR have a lower recall but a higher precision. Hence, the user needs to decide whether a high recall is more important than precision or whether a lower recall can be taken into account improving precision (see D_4 in Table 13.2). Depending on the decision either the NB or MNB classifier or one of SGD or LR should be used in ITSoFD.

CLASSIFICATION We know that issue categories are often wrong or missing and thus cannot be used for a reliable detection (F/I 3). Even if categories are maintained well in a particular project they offer a classification on the issue level, only. However, ITSs do not

offer categories on the level of ITS data fields (F/I 4) and especially not on the level of sentences. However, a good categorization on the issue level can support the detection SFRs.

Therefore, the NL in the ITS is analyzed in addition to looking at issue categories. ITSoFD recommends to train a ML model based on the NL and on the ITS metadata, which is backed by the evaluation in Chapter 15. In particular two cases are shown in the evaluation: (1) the BOW MLF alone delivers the best results if the user aims for a high recall. (2) If the user aims for a high F-score, ITS meta-data should be considered together with the BOW. Hence, different sets of MLFs should be considered for SFR detection, depending on the user decision D_4. This is visualized as two different paths in Figure 13.1.

Using the MLFs, the classifier decides for each issue, data field, or sentence whether it contains an SFR. The classification level depends on user decision D_3. Chapter 15 shows a detailed evaluation of this phase and discusses the performance of the classifiers under different circumstances. Roughly, however, F-scores around 0.4 or a recall up around 0.98 with a precision of about 0.15 can be expected for a detection on the data field level. On the issue level F-scores are usually slightly better and worse on the sentence level. All three levels result in a similar maximum recall with varying precision.

Finally, all SFRs are detected after Phase 3. Depending on decision D_3 either issues, ITS data-fields or sentences are detected and depending on decision D_4 either the F-score or $MAX(R), P_{\geqslant p}$ is maximized. These intermediate results are reflected in Figure 13.5.

13.2.4 *ITSoFD Phase 4: Related Issue Tracing*

This section describes Phase 4 in Figure 13.2: tracing to related issues.

INTRODUCTION Detecting SFRs or other SF-related NL is a good start to find the information describing a particular SF. However, findings F/I 9 and F/I 10 both suggest that SF-related information can be spread over multiple issues. Although ITSs provide means to create traces between issues, these means are seldom used by the developers and thus the traceability between issues is usually bad.

Hence the third part of the solution improves the traceability between issues so that related issues can easily be found. ITSoFD suggests to extract the traces provided by the developers in the ITS, first, as the study in Chapter 16 finds that these traces are seldom wrong. Then an IR based trace retrieval approach should be used to improve the traces. With a good traceability, the information related SF can easily be found by following those traces.

Traces created by the developers can easily be extracted by parsing domain specific languages that create links between issues, as

explained in Section 2.2. After extracting the links from the issues NL, these links can be added to the trace matrix.

Such a trace matrix is usually sparse, thus trace retrieval should be applied next. However, the evaluation study in Chapter 16 shows very ambivalent results with respect to trace recovery in ITSs. However, some best practices can be suggested, well knowing that trace retrieval in issues is still a topic for further research. Due to this volatile nature of ITS trace retrieval, the user should explicitly decide whether this part should really be used (see decision D_5).

WEIGHTING ITS DATA FIELDS In general IR algorithms consider every term in a document equally. Data fields such as issue title and description are often focused clearer on the topic, whereas comments can digress and even discuss unrelated problems (discussed in Part iv or by Bertram, 2009). Thus titles and descriptions should be considered more important by the IR algorithm. This can be achieved by assigning higher weights on title and description. In addition, the evaluation described in Chapter 16 shows that not considering technical information usually improves trace retrieval results. Overall ITSoFD recommends weighting the title four times higher than comments and descriptions twice as high as comments for IR. Since technical information should not be considered at all, it should be weighted zero.

ALGORITHMS AND THRESHOLDS Due to the results of the evaluation study in Chapter 16, the VSM IR algorithm delivers the most reliable results for trace retrieval between issues. However, as discussed in the Background Chapter on page 31, Equation 4.4 shows that IR algorithms cannot be used to define traces directly. Instead they measure the similarity between documents. This similarity value – usually a value normalized between 0 and 1 – needs to be thresholded to create a link (cf. Equation 4.5).

Modifying the threshold can significantly improve results for a particular project (i.e. a particular set of issues) and experimentation with the threshold is recommended (see O_4 in Table 13.2). A higher threshold improves the trace retrieval precision on the cost of a lower recall, whereas a lower threshold improves recall on the cost of precision. In the evaluation in Chapter 16 F-scores up to 0.5 are achieved in one project, using thresholds of 0.35 for VSM. Hence this threshold represents a good starting point.

13.3 RELATED WORK

Related work for this solution was compiled by screening the last five years of the "IEEE International Requirements Engineering Conference (RE)", the "International Working Conference on Requirements

Engineering: Foundation for Software Quality Conference (REFSQ)", the "International Conference on Mining Software Repositories (MSR)" and the "International Conference on Software Engineering (ICSE)". Forward and backward snowballing was then applied to relevant publications. In addition, an explicit search for systematic reviews in the respective fields of each study using the IEEE[9], Springer[10], ACM[11], and Sciencedirect[12] databases was performed[13].

In the following sections, the related work for every part of the solution is presented.

13.3.1 *ITS and Task Specific Preprocessing*

SEPARATION OF TECHNICAL INFORMATION AND NATURAL LANGUAGE Various techniques have been presented to deal with the separation of NL and technical information. This section presents these approaches and discusses the differences with respect to the approach described in Section 13.2.2.1 and evaluated in Chapter 14.

Bettenburg et al., 2011 presented two techniques to separate NL and technical information. The first approach (Bettenburg et al., 2011) is the most similar to the one described in Chapter 14 as it includes text heuristics. First, they use a spell-checker to identify wrongly written words and treat those words as technical information. Then, they validate the findings of the spell-checker by applying additional heuristics. Their second approach uses island grammars (Moonen, 2001) to identify patches, stack traces, source code and enumerations (Bettenburg et al., 2008). The outcome of this approach, however, is a binary classification, since it only checks whether code, stack traces, patches or enumerations are present in the document or not. However, the exact location of these artifacts is not identified. In contrast ITSoFD

9 http://ieeexplore.ieee.org
10 http://link.springer.com/
11 http://dl.acm.org
12 http://www.sciencedirect.com
13 This approach was used in favor of a complete systematic database search as suggested e.g. in (Kitchenham and Charters, 2007) due to the circumstance that most relevant search terms with respect to ITSoFD deliver extremely broad results that can hardly be curtailed further. Although systematic literature reviews were proven very reliable, a broad search term usually deteriorates results (MacDonell et al., 2010) and the thesis' topics adhere many broad search terms. A systematic database search was approached and although some relevant items could be identified, this review could not be conducted thoroughly. Examples for problematic terms in the context of this thesis are: "issues", as issues are addressed in every scientific paper; "feature", a term that occurs in many publications describing a software implementation; "tracking", a well-known technique the robotic and computer vision fields; or "natural language", which is relevant and used as a term in publications from almost any scientific field. However, any relevant intermediate results from the database search are included in the related literature. Finally, this thesis is based on the findings of five peer reviewed publications. Any related work mentioned by the referees of the respective publications was included if appropriate.

classifies the specific part of the document, i.e. the line or word that contains the technical data.

Bacchelli et al., 2012 introduced an approach that combines parser-based techniques with an NB classifier as a term-based technique to classify documents in NL, stack traces, patches, code and a class they call junk[14]. The approach performs very well, but it relies on a manually annotated training dataset for the NB classifier. Their approach is based on earlier work published in Bacchelli, D'Ambros, and Lanza, 2010; Bacchelli et al., 2009 and Bacchelli et al., 2011. In contrast, the approach described in Chapter 14 does not need manually annotated training data.

Cerulo et al., 2013 address the problem using Hidden Markov Models, which train from the data. Their approach does not require any manual tuning or the definition of regular expressions. Similar to the approach by Bacchelli et al., 2012 it tries to learn from the data in a supervised manner. Thus both approaches rely on a training dataset. In contrast ITSoFD performs well on different datasets without project specific tuning. It can, however, be further improved by adding project specific heuristics or regular expressions.

13.3.2 *SFR Detection*

This section presents approaches related to SFR detection and discusses the differences with respect to the solution described in Section 13.2.3 and evaluated in Chapter 15.

Ryan, 1993 discussed the role of NL RE. He states that "although the computing professional will not be replaced by a super-intelligent NL, there are still a number of realistic uses for NLP in the RE process". This statement is still valid over 20 years later. SFR detection is one possibility to support the computing professional in modern ITS driven environments.

To the knowledge of the thesis' author, no published work tries to detect SFRs in ITSs using ML techniques. Hence, the related work with respect to this solution component is limited.

First of all, there is the field of feature extraction. Recently, Bakar, Kasirun, and Salleh, 2015 performed a systematic literature review on feature extraction approaches from NL requirements in the context of software product lines. Their review includes multiple methods to summarize, cluster, or abstract from software requirements and to generate representations of SFs. In contrast to the approaches reviewed by Bakar, Kasirun, and Salleh, 2015 ITSoFD uses ITS data as input, whereas the reviewed approaches utilize requirement documents or product descriptions. However, in contrast to ITS data, such require-

14 Junk is "text that does not add valuable information, such as auto-generated disclaimers . . . ." (Bacchelli et al., 2012).

ment documents or product descriptions are usually better structured and less error prone.

In addition to feature extraction, the detection quality requirements, also called non-functional requirements, is an issue for automated classification tasks. Cleland-Huang et al., 2006 detect and classify quality requirements. Rahimi, Mirakhorli, and Cleland-Huang, 2014 extract and visualize quality concerns from SRS. Both approaches, however, rely on a structured and well written SRS. In addition, both approaches use mainly keywords, or hierarchies of keywords to do the classification. In contrast, ITSoFD uses ML to achieve the classification and incorporates many text-based MLFs, not only keywords.

Another branch of related work with respect to RAs is automatic RA classification. Ott, 2013 and Knauss and Ott, 2014 classify large volumes of NL requirements with multi class NB classifiers. The classification of large volumes of RA data has one advantage, however: it provides a large amount of well-written training data, which usually improves prediction rates.

Furthermore, some mining techniques have been proposed on the side of software architecture and structured RAs. Casamayor, Godoy, and Campo, 2012 presented a comprehensive review on mining textual requirements to assist architectural software design. Most of the reviewed approaches use structured RAs as basis for these mining tasks. Boutkova and Houdek, 2011 proposed a semi-automatic identification of features in SRSs. Their approach too relies on the document structure of the SRS and manual work to align structural information with the identification algorithm.

Much work has been done to automate the classification of issues:

1. The classification of bugs vs. software features (Antoniol et al., 2008; Chawla and Singh, 2015; Neelofar, Javed, and Hufsa, 2012; Zhang and Lee, 2011).

2. The problem of issue triagingr, e.g. assigning a suitable developer to resolve the issue (Ahsan, Ferzund, and Wotawa, 2009; Duan et al., 2009; Kagdi et al., 2012; Matter, Kuhn, and Nierstrasz, 2009; Nagwani and Verma, 2012; Shokripour and Anvik, 2013; Zhang and Lee, 2013; Zou et al., 2011).

3. The priority or severity prediction for bugs (Duan et al., 2009; Lamkanfi et al., 2011; Mukherjee and Garg, 2013; Tian, Lo, and Sun, 2012).

Such classification tasks show improving results over the years and are an ongoing topic in the MSR community (Hemmati et al., 2013). The SFR detection in ITSoFD builds on this knowledge with respect to preprocessing techniques and MLFs that have been applied in issue classification tasks and it extends the body of knowledge by combining existing with additional preprocessing techniques. The evaluation

study in Chapter 15 contains the most complete set of classifiers and presents a combined evaluation of MLFs that performed well in previous work. All of the studies shown above classify either the issue as a whole, the title, and/or the description of an issue. The SFR detection in ITSoFD is complementary, as it identifies SFRs in ITSs on different levels of detail (in other words ITSoFD classifies data fields or sentences).

It should be noted, that some ITS related classification methods are validated by comparing the classifier results to given ITS meta-data. However, Herzig, Just, and Zeller, 2013 showed that ITS meta-data is often assigned wrongly and that this course of action can introduce evaluation errors. Hence the evaluation of all studies in this thesis relies on manually created gold standards.

Maalej and Nabil, 2015 presented similar work to the SFR detection method in ITSoFD: they classify the reviews in app stores as bug report or SFR using ML. Similar to the findings presented in the beginning of this chapter, they too face problems with respect to badly written NL and incomplete sentences. Preprocessing and evaluation of ITSoFD builds on this knowledge and complements their work. In particular ITSoFD includes ITS meta-data as MLFs in the classification task.

Vlas and Robinson, 2012 and Vlas and Robinson, 2013 present a bottom-up approach for requirements detection. Their definition of requirements includes SFRs. They use heuristics, keywords, and an LE based approach. However, they assume that a software requirement is formulated using a single sentence, which in is not always the case as discussed in Section 13.1. Hence, SFR detection in ITSoFD uses supervised ML algorithms to overcome the limitation of classifying single sentences, only.

So far the discussed approaches use NL or structured NL as input. However, many automation techniques in RE are based on formal or semi-formal language. Such methods are not included in related work, since they rely on different techniques (e.g. mostly parsers relying on the formalism in combination with finite state machines) than the ones applicable for ITSoFD. Even though methods came up recently that derive models and formal language from NL (Ghosh et al., 2016), such methods are still applied to semi-formal NL that cannot be found in issues.

Finally, it is important not to confuse the detection of SFRs with the research field of feature location. Feature location methods discover the source code implementing SFs. Taxonomies and surveys on feature location methods were compiled by Dit et al., 2013 and Rubin and Chechik, 2013.

13.3.3 *Trace Recovery in ITS Data*

This section presents trace retrieval approaches and discusses differences with respect to the solution described in Section 13.2.4 and evaluated in Chapter 16.

One of the first studies in trace retrieval was conducted by Lucia et al., 2007. They found that the performance of trace retrieval methods depends largely of the type of RAs that should be traced and, of course, on the data quality.

Borg, Runeson, and Ardö, 2014 conducted a systematic mapping study of trace retrieval approaches. Their paper reveals that much work has been done in trace retrieval between RAs, but only few studies use ITS data. Only one of the reviewed approaches in (Borg, Runeson, and Ardö, 2014) uses the BM25 algorithm, but VSM and LSA are used extensively. The study that evaluates ITSoFD trace retrieval in Chapter 16 fills both gaps by comparing VSM, LSA, and three variants of BM25 on unstructured ITS data. Borg, Runeson, and Ardö, 2014 too report on preprocessing methods saying that stop word removal and stemming are most often used. The evaluation of ITSoFD focuses on the influence of ITS-specific preprocessing and ITS data field-specific term weighting beyond removing stop words and stemming.

Gotel et al., 2012, too, summarize the results of many approaches for automated trace retrieval in their roadmap paper. They recognize that results vary largely: "[some] methods retrieved almost all of the true links (in the 90% range for recall) and yet also retrieved many false positives (with precision in the low 10-20% range, with occasional exceptions)." The evaluation in Chapter 16 shows that the results on ITS data are comparable or worse depending on the project, although ITS specific improvements are included.

A field largely worked on in MSR is the detection of duplicated issues, using IR techniques (Amoui et al., 2013; Borg et al., 2014; Dang et al., 2012; Kaushik and Tahvildari, 2012; Runeson, Alexandersson, and Nyholm, 2007; Sureka and Jalote, 2010; Tian, Sun, and Lo, 2012; Wang et al., 2008). Heck and Zaidman, 2014 performed experiments with ITS data for duplicate detection with good recall rates. They found that extensive stop word removal can be counter-beneficial for ITS data. The ITSoFD trace retrieval study in Chapter 16 complements this work and shows how well duplicates can be detected on ITS data if results are compared to a manually created gold standard[15]. Furthermore, the experiments presented in Chapter 16 are performed with and without stop word removal, thus showing the impact of this preprocessing technique.

Both, Borg, Runeson, and Ardö, 2014 and Gotel et al., 2012, mention a considerable amount of related work that uses IR techniques

15 Considering the findings by Herzig, Just, and Zeller, 2013 discussed in the previous section.

for trace recovery, mostly on structured RAs. In the experiments for ITSoFD trace retrieval is restricted to standard IR methods, only. In the following extended approaches are summarized that could also be applied to ITS data and/or combined with the contribution from ITSoFD's trace retrieval component:

- Nguyen Duc et al., 2011; Nguyen et al., 2012 combine multiple properties, like the connection to a version control system to relate issues.
- Guo, Cleland-Huang, and Berenbach, 2013 use an expert system to calculate traces automatically. Although their approach is very promising, it requires a significant amount of manual work to build up the expert system and a complete automation of the approach is not yet feasible.
- Sultanov and Hayes, 2013 use reinforcement learning, which shows superior results compared to the VSM in their experiments.
- Gervasi and Zowghi, 2014 use affinity mining, e.g. a measure for sets of nouns, adjectives, adverbs, and verbs that re-occur in previously traced high-level and low-level requirements. Their approach is build on previous work (Gervasi and Zowghi, 2011).
- Niu and Mahmoud, 2012 use clustering to group links in high-quality and low-quality clusters respectively to improve accuracy. This way they can filter out low-quality clusters from their trace retrieval results.
- Comparing multiple techniques for trace retrieval, Oliveto et al., 2010 found that no technique considerably outperformed the others. Due to this finding, they combine LSA with other techniques, which improves the result in many cases.

Finally, it should be noted that even human experts are no guarantee for perfect traces. Cuddeback, Dekhtyar, and Hayes, 2010 studied the performance of human analysts in trace retrieval tasks and they found that even experts cannot fulfill this task perfectly, especially not on larger datasets.

14

SEPARATING NATURAL LANGUAGE AND TECHNICAL DATA – AN EMPIRICAL STUDY

"Domain knowledge [...] can be used in text mining preprocessing operations to enhance concept extraction and validation activities."
- Feldman and Sanger, 2006

The ITSoFD method presented in the previous chapter proposes different preprocessing techniques on the text data in issues. Most of these techniques overlap with what is considered standard in NLP. However, the separation of technical data from NL is a complex task on its own.

This chapter presents an empirical study evaluating the separation of technical data and NL introduced in Section 13.2.2.1. Although this is not directly related to the RQs defined in Section 1.3, the separation of code and NL is an important data preparation step in ITSoFD and is thus necessary to answer RQ 3 and RQ 4 stated on page 5.

14.1 STUDY DESIGN

First, 225 documents are extracted from nine OSS projects. These documents are converted into XML to be processed in GATE. Second, GATE is used to create a gold standard, i.e. the technical information in every issue is manually annotated. As this Chapter is based on the publication (Merten et al., 2014), the first and second author of this paper created the gold standard. Finally, the algorithm proposed in Section 13.2.2.1 is run and results are reported with and without applying hierarchical clustering. Figure 14.1 summarizes the study setup.

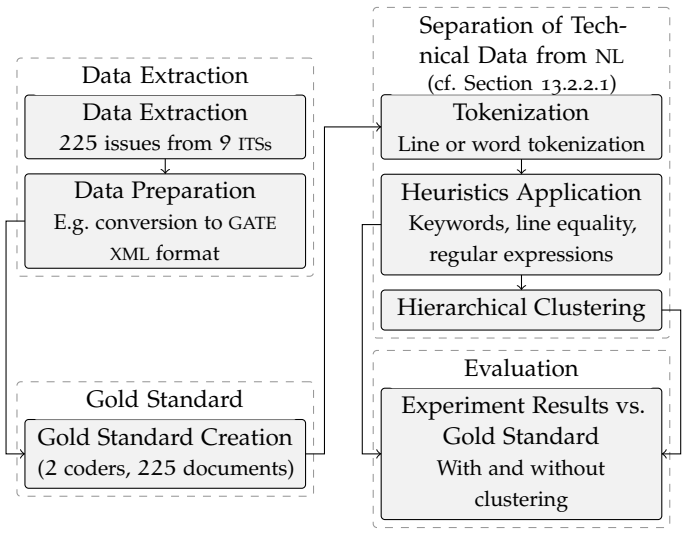


Figure 14.1: Study Setup.

Every document was manually annotated based on white space tokenization to create the gold standard. Then the method described in Section 13.2.2.1 is applied and its outcome is compared to the gold standard.

As introduced in Section 13.2 a *regular expression* can be introduced to detect marked-up technical information. However, such a mark-up arguably structures the documents. In contrast the goal of this study is to evaluate the approach on unstructured documents. Thus, no regular expression that matches marked-up technical information is used in this particular study [1].

14.1.1 *Research Question*

> How well can technical data be separated from NL using 'simple' heuristics and hierarchical clustering?
> *Expected is that the separation can be done well without additional clustering for line tokenization. However, for word tokenization heuristics often make mistakes and clustering should improve the results.*

14.1.2 *Data*

To evaluate the approach, 225 documents were randomly sampled from issue trackers and mailing list of nine OSS projects (i.e. 25 documents per project) as introduced in Section 8.1. On average the documents are 1698 characters long.

The most important rationale for this data set is that the projects employ multiple programming languages and the evaluation should show that the approach can be applied to different ITSs. Table 14.1 provides information on the sources of the evaluated documents: it includes the programming languages that are used in the projects and whether the documents were retrieved from ITSs or mailing lists. In addition, Table 14.1 shows how many documents contain NL, only, a mixture of NL and code (C), a mixture of NL and log files or stack traces (L/S), and a mixture of NL and patches (p) as annotated in the gold standard.

14.1.3 *Evaluation Procedures*

True positives, true negatives, false positives and false negatives are calculated by comparing the output of the algorithm to the gold standard. Experiment results are evaluated for both, white space and line

1 Note that mark-up detection boosts the separation of technical data to to an F_1 score close to 1.0 for the data used in the following two studies. Mark-up detection is used for the separation of technical data in the following studies presented in Chapter 15 and Chapter 16.

PROJECT	LANGUAGE	SOURCE	# OF DOCUMENTS WITH			
			NL ONLY	C	L/S	P
Apache ActiveMQ	Multiple*	ITS	1	13	14	1
Linux Kernel	C	ITS	0	12	9	11
Mozilla Core + JSS	C++, Java	ITS	1	13	2	13
Apache OpenOffice	C++	ITS	0	23	2	0
Apache Jmeter	Java	ITS	2	13	7	3
Apache OFBiz	Multiple*	ITS	2	19	6	0
Apache Avro	Multiple*	Mail	2	19	6	2
Apache Camel	Multiple*	Mail	1	19	12	1
Apache Thrift	Multiple*	Mail	1	19	6	1

With C: number of documents that contain NL and code;
L/S: number of documents that contain NL and log files or stack traces;
P: number of documents that contain NL and patches.

* More than 4 programming languages are used in the project, i.a. ActionScript, C, C++, C#, D, Delphi, Erlang, Groovy, Java, JavaScript, Perl, PHP, Ruby, Python ...

Table 14.1: Corpora for NL and Technical Separation.

tokenization. For line tokenization, the lines in the gold standard that contain more NL than technical information were considered NL and vice versa. In the evaluated documents, however, such mixed lines generally contained $< 5\%$ of technical information.

The same parameters and settings are used for all documents, to show that the approach provides reasonable results with a default setting. Note that this implies that results can be improved whenever special heuristics or parameters are introduced for a particular corpus.

14.2 RESULTS

Table 14.2 shows the results using line segmentation and Table 14.3 shows the results using white space tokenization. Overall, the results are comparable for both tokenization techniques. Although line tokenization performs somewhat better on most projects, white space tokenization provides far more accurate results. Since results vary only $< 3\%$ in both, precision and recall between the two tokenization techniques there is no need to fall back to line tokenization. In general, the results are always $\geqslant 80\%$ except for the Apache Avro and Apache Thrift projects.

PROJECT	s_{min}	P	R	F_1
Apache ActiveMQ	0.8	0.91	0.91	0.91
Apache Avro	0.85	0.65	0.83	0.73
Apache Camel	0.95	0.74	0.90	0.81
Apache JMeter	0.85	0.88	0.89	0.89
Apache OFBiz	0.95	0.82	0.88	0.84
Apache OpenOffice	1.0	0.80	0.91	0.86
Apache Thrift	0.9	0.58	0.83	0.68
Linux Kernel	0.85	0.87	0.94	0.91
Mozilla Core + JSS	0.75	0.78	0.86	0.82
Average	0.9	0.77	0.88	0.82
Average (no clustering)	n/a	0.69	0.88	0.77

Table 14.2: Results for Line Tokenization.

PROJECT	s_{min}	P	R	F_1
Apache ActiveMQ	0.85	0.93	0.89	0.91
Apache Avro	0.95	0.74	0.87	0.79
Apache Camel	0.9	0.76	0.86	0.81
Apache JMeter	0.95	0.91	0.83	0.87
Apache OFBiz	0.95	0.80	0.84	0.82
OpenOffice	0.95	0.87	0.89	0.88
Apache Thrift	0.95	0.71	0.86	0.78
Linux Kernel	0.9	0.96	0.92	0.94
Mozilla Core + JSS	0.95	0.80	0.82	0.81
Average	0.9	0.84	0.85	0.84
Average (without clustering)	n/a	0.67	0.84	0.74

Table 14.3: Results for White Space Tokenization.

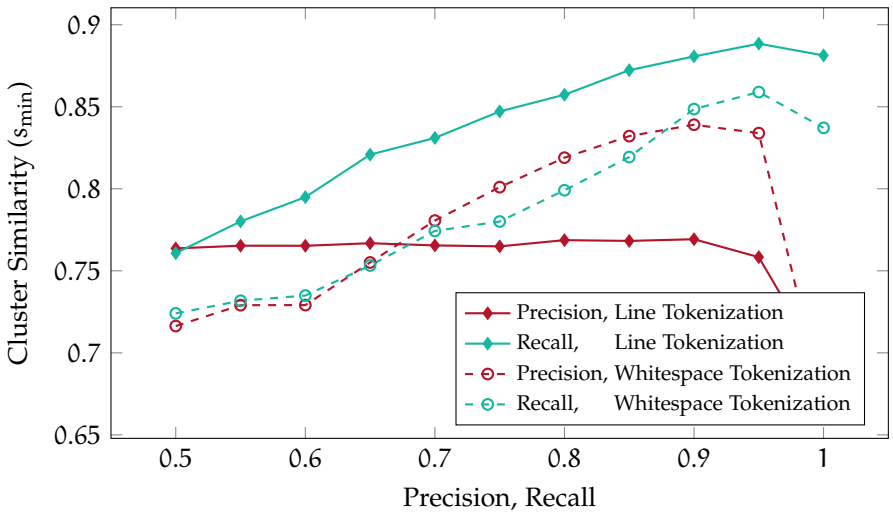


Figure 14.2: Precision and Recall for Different Cluster Similarities.

Experiments are performed with with varying values for s_{min}, varying from 0.5 to 1.0 in 0.05 steps. Figure 14.2 shows precision and recall for the different cluster similarities. Solid curves represent line tokenization and dotted curves white space tokenization. For line tokenization, the best results over all projects are generated with s_{min}= 0.9. This results in a precision of 0.77 and a recall of 0.88 ($F_1 = 0.82$). Without clustering, the heuristics classified the documents with $P = 0.69$ and $R = 0.88$. Hence the clustering improved the precision by about 8%. For white space tokenization, the best results are generated with $0.9 \leqslant s_{min} \leqslant 0.95$. This results in $P = 0.84$ and $R = 0.85$ ($F_1 = 0.84$). Without clustering the precision is significantly lower with $P = 0.67$ and $R = 0.84$. Hence the precision increased by over 17% and the recall by 1%.

14.3 DISCUSSION

This section first discusses the implication of the above results for ITSoFD in practice. Then it discusses the main implications of this study for future research on separating technical data from NL.

PRACTICAL IMPLICATIONS For an adoption of ITSoFD into practice, the results of this study are satisfying.

However, even with a maximum F_1 score of 0.94 and an average F_1 score of $\geqslant 0.82$ a non-negligible amount of text is classified wrongly. Thus the approach should not be used blindly. Especially, if ITSoFD is applied to an ITS with very few technical information false positives could worsen the results. On the other hand it should be considered that (1) the approach was not tailored to any of the datasets and (2) regular expressions to detect explicitly marked-up technical

information were intentionally not used. Whenever the approach is tailored to a specific project, better results can be expected.

Furthermore, the approach cannot only be used as a preprocessing step as intended for ITSoFD. It can also be applied on its own. If for example code snippets or excerpts of log files should be extracted from a developer mailing list, the separation of technical data from NL provides the classification for such an extraction.

INFERIOR RESULTS FOR TWO PROJECTS Due to the inferior results for the Apache Avro and Thrift projects, the documents from these projects were inspected manually. The inspection yields that developers of both projects tend to communicate using a mixture of NL and code. E.g. Figure 14.3 shows a document extracted from the Apache Thrift mailing list. In this project NL and code are intermingled word-wise and line-wise. Hence, both, the heuristics and the clustering algorithm can hardly detect an associated block of NL or code, respectively. For this kind of data specific heuristics need to be designed or other approaches, as e.g. in Section 13.3, might be applicable. If this approach is used in such documents, a detection on the level of lines instead of words is recommended.

```
[...]
struct_v TMVBase «– struct_v tell the generator to created the
TBase class with a factory ("v" for virtual)
service USB {
    void martin(1:TMVBase base);
}
This is then generated into someting like this:
uint32_t USB_martin_args::read(::apache::thrift [...]
```

Figure 14.3: Example of Intermingled Code and NL from Apache Thrift.

DEFAULT VALUES FOR s_{MIN} In the experiments for all projects, the clustering algorithm reaches maximum performance with s_{min} from 0.8 to 0.95. Thus it is likely that no tuning or further experimentation below or above these values for s_{min} is necessary. The best performance can be expected around 0.9 as shown in Figure 14.2. Furthermore, the values for s_{min} are increased in 0.05 steps in this empirical study. Additional experiments with a finer scale might yield marginally better results.

14.4 THREATS TO VALIDITY

INTERNAL VALIDITY Results were evaluated using a manually annotated gold standard. The gold standard was annotated by two researchers and some documents were double-checked. However, it is

an assumably simple task for a trained software engineer and software engineering researcher to tell technical information apart from NL. Hence, I argue that a dual coding is not necessary in this case in contrast to more complex annotation tasks.

EXTERNAL VALIDITY To ensure generalizability, 225 documents were sampled out of 9 projects for evaluation. In addition, the same configuration was used for all projects, so that the results should represent a lower bound. This lower bound can be improved by tailoring the approach to a certain setting. For example regular expressions to detect mark-up in ITSs can be added or keywords from unused programming languages can be removed.

14.5 CONCLUSION

In this chapter an approach for separating technical data and NL is evaluated. The results show that good heuristics already deliver reasonable results for this task. However, at some point heuristics fail, as they can never be specific enough for all cases. It is shown that clustering helps to eliminate some of the 'mistakes' that heuristics make, especially on the level of word tokenization.

15

DETECTING SOFTWARE FEATURE REQUESTS IN ISSUES – AN EMPIRICAL STUDY

"It is in supporting that social [requirements engineering] process, and not in supplanting it, that natural language processing will have its proper role."
- Ryan, 1993

The empirical study presented in this chapter investigates Research Question 3 defined in Section 1.3:

> RQ 3 Can SFs descriptions be detected automatically in ITS NL data?

The study evaluates the SFR detection of ITSoFD described in Section 13.2.3. The main goal of this chapter is to study whether particular expressions to formulate an SFRs can be detected automatically. To achieve this goal multiple ML algorithms, different text preprocessing techniques, and MLFs are evaluated and the following three parts of an SF are detected:

REQUESTS (or SFRs) Text that requests for a distinguishing characteristic of a software item (e.g. a quality or functionality) that provides value for users of the software.

CLARIFICATIONS Text that explains an request, e.g. because the functionality needs further context.

SOLUTION PROPOSALS Text that describes implementation ideas for an request.

In the next section the study design is presented. Then the above RQ is systematically refined in three fine grained RQs in Section 15.1.1. The experiment setup is described thereafter in Section 15.1. The results are presented in detail in Section 15.2 and discussed in Section 15.3. Finally, Section 15.4 discusses related threats to validity and the final section concludes this chapter.

15.1 STUDY DESIGN

The study is composed of four steps:

1. The extraction and preparation of the data,
2. the creation of a gold standard,
3. the engineering of MLFs, and
4. the evaluation of different ML algorithms, MLFs, and preprocessing techniques.

Details of this process are described in the following sections and an overview is given in Figure 15.1.

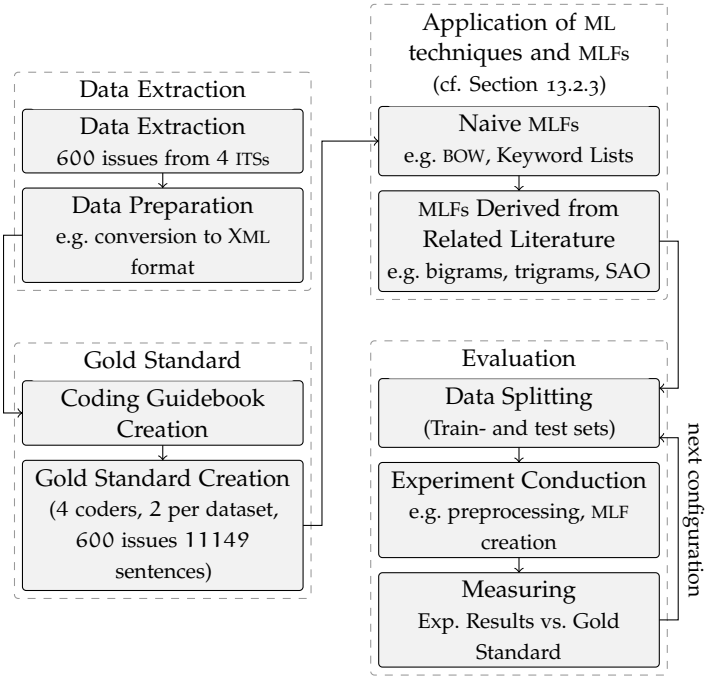


Figure 15.1: Study Setup.

15.1.1 *Research Questions*

The overall research question of this chapter, whether software feature requests can be detected automatically in ITS NL data, is divided in the following three fine grained RQs:

RQ 3.1 How should text be preprocessed for SFR detection?
Expected is that standard preprocessing techniques and the separation of technical data from NL improve the results for SFR detection. However, stop-word removal may worsen the results as stopwords such as "should", "could", or "would" are often used in SFRs.

RQ 3.2 Which combinations of MLFs derived from the NL and ITS meta-data should be used for SFR detection on different levels of detail?
Expected is that ITS meta-data improves the results in comparison to text based MLFs. Meta-data such as the issue type should help detecting SFRs.

RQ 3.3 How well can trained prediction models be reused?
Expected is that reusing prediction models does not work well across different projects. Especially because the projects used in this study have different characteristics[1].

Section 2.2 states some of the problems with ML and ITS data, that can often be mitigated with the preprocessing techniques introduced in

1 Confer Section 8.2 on page Section 8.2.

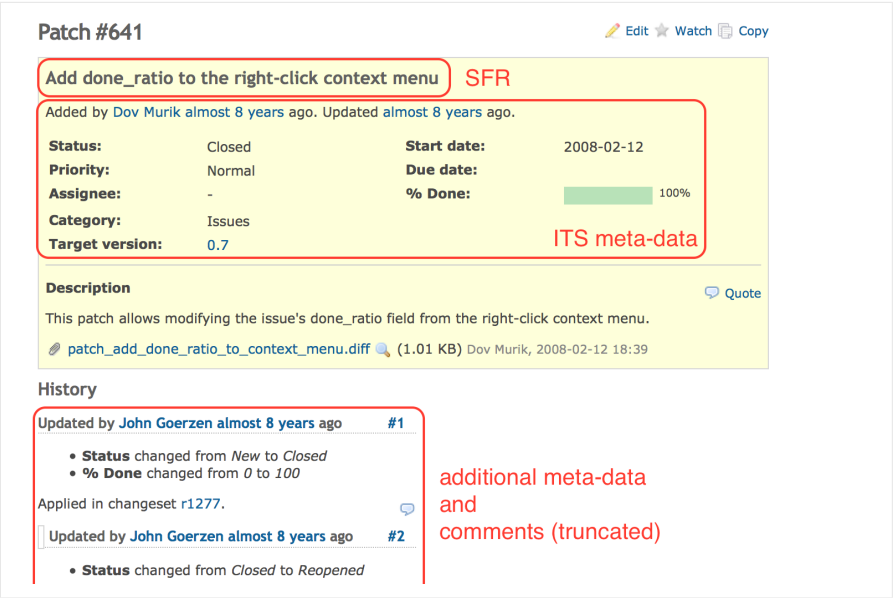


Figure 15.2: Redmine Issue Example.

Section 3.2. RQ 3.1 discusses which combination of such techniques should be used. RQ 3.2 contrasts rather traditional MLFs with ideas building upon related work and takes MLFs based on issue and data field meta-data into account. Furthermore, it investigates whether the very sentence that contains the SFR can be extracted or if more data (e.g. from the comprising data field or issues) is needed. RQ 3.3 asks whether models have to be trained for every project or if trained models can be applied to other projects, different from the projects used for training.

15.1.2 *Data*

All projects researched in this thesis are described in detail in Section 8.1. In this study the data of the same four projects as in the study on issue and information types (confer Chapter 11) is utilized:

- c:geo
- lighttpd
- Radiant
- Redmine

An example issue from the data set given in Figure 15.2. The figure shows a screenshot of the issue as it originally appears in the Redmine ITS and depicts the SFR, ITS data fields, and ITS meta-data in red.

A gold standard was manually created based on the taxonomy developed in Chapter 11 and shown in Figure 11.3. Relevant parts of the related coding guidebook, developed for the study in Chapter 11 and shown in Appendix A, were re-used to code the 599 issues for this study. Every sentence of every issue (including all comments) was annotated – if relevant – by two coders independently. The coders used the following labels[2]:

REQUEST FUNCTIONALITY to denote functional SFRs.

REQUEST QUALITY to denote quality SFRs.

SOLUTION to denote technical solutions for an SFR.

CLARIFICATION to denote additional explanations to an SFR.

In the following the umbrella term "SFR detection" will be used for all above labels. An exception is *Request Quality*, as too little data was found by the coders to train a model for requests for software quality.

The above labels represent an SFR relevant subset of the complete taxonomy of issue and information types[3]. The above selection includes the information types that are directly related to an SFR. Obviously, all issues can be coded using the complete taxonomy to evaluate the algorithms. However, a selection was necessary because of the following two reasons: (1) dual coding of all sentences using the complete taxonomy would have taken too much resources and (2) manual coding is usually more error prone, the more codes/coders are involved Neuendorf, 2002.

Due to the dual coding two datasets are created composed of the very same issues:

1. The sentences that were annotated identically by two annotators. This dataset will be referred to as the *agreed upon cases*.
2. The sentences that were only labeled by a single annotator. This dataset will be referred to as the *uncertain cases*.

On average, the annotators agreed with a rather high Cohens kappa (Cohen, 1960) of 0.91 on the labels for issue titles and 0.88 on the labels for issue descriptions. Due to limitations in GATE, the exact kappa for the issue comments cannot be reported. However, a significantly lower agreement was observed in random samples from comments in comparison to title and description. This can be seen in Table 15.1, which shows a disparity of labels for the two datasets. The full interrater agreements can be found in Appendix A expressed as F_1 scores and Cohens kappa, respectively. The differences in agreement imply that even human experts have a tendency not to agree whether a sentence contains an *SFR* , a *request for quality* , an *clarification*, or a *solution*. The influence of agreement factors is discussed in Section 15.4.

2 More exhaustive descriptions can be found in Section A.2.
3 Refer to Chapter 11 for the full taxonomy.

	ANNOTATIONS AGREED UPON BY TWO CODERS				
LABEL	C:GEO	LIGHTTPD	RADIANT	REDMINE	ALL
Request Functionality	23 / 1.08	9 / 1.0	10 / 1.0	34 / 1.12	76 / 1.05
Request Quality	4 / 1.0	0	0	0	4 / 1.0
Solution	18 / 1.0	2 / 1.0	6 / 1.16	10 / 1.0	36 / 1.04
Clarification	46 / 1.04	17 / 1.12	11 / 1.27	26 / 1.23	100 / 1.17
	UNCERTAIN CASES				
Request Functionality	70 / 1.17	28 / 1.0	43 / 1.07	155 / 1.21	296 / 1.11
Request Quality	8 / 1.0	1 / 1.0	4 / 1.0	4 / 1.0	17 / 1.0
Solution	72 / 1.36	14 / 1.5	28 / 1.36	148 / 1.41	262 / 1.41
Clarification	223 / 1.50	77 / 1.44	110 / 1.21	234 / 1.38	644 / 1.38

Annotations / Avg. # of Sentences per Annotation

Table 15.1: Extracted Data and Annotations.

During the annotation phase, the annotators collected keywords for the *Request Functionality* label. New keywords were added to the keyword list whenever a word was noticed repeatedly by the annotator, whereas "repeatedly" was generally as "more often than $5-15$ times". Examples from this collection are modal verbs such as *should*, *would* and *could* as in "we *should* implement [...]" or "*could* you please add [...]". The full keyword list is used as one MLF in the detection. A complete list of MLFs is given in the following section.

Finally, both datasets, the *agreed upon cases* and the *uncertain cases* are used to train the ML models and to evaluate the SFR detection. Table 15.1 shows the number of annotations and the average number of sentences for every label in both datasets.

15.1.3 *Algorithms and Settings*

MLFS TO IDENTIFY SOFTWARE FEATURES This paragraph summarizes those parts of the ITSoFD related work presented in Section 13.3 that adds MLF ideas to the SFR detection and describes how these MLFs are used in this study. Then, a summary of all MLFs is given.

Guzman and Maalej, 2014 use collocations to denote fine grained SFs in app store reviews. Their idea is evaluated SFR detection in this study. However, in contrast to the original study, tri-grams are evaluated, too. Tri-grams are useful whenever word triples such as "we should add" are used in SFRs, as discussed in Chapter 11.

Fitzgerald, Letier, and Finkelstein, 2012 use a mixture of ITS meta-data MLFs and linguistic MLFs for early failure prediction in ITS. Maalej and Nabil, 2015 show that meta-data such as star ratings in app stores can improve SE-related classification tasks. Based on these ideas MLFs are derived from the ITS meta-data and the data field meta-data in this chapter.

Besides other heuristics, the approach by Vlas and Robinson, 2012 uses a subject-action-object pattern based on the work of Fantechi and Spinicci, 2005 for their heuristics. The action is actually the main verb of the sentence filtered by different heuristics and keyword lists, e.g. "need" in "The software *needs* logging". Though a replication of all their heuristics is unfeasible in the ML context, their idea is applied in this study and MLFs are derived from subject-verb-object triplets based on grammatical parsing.

In summary, the following MLFs were used to conduct the experiments. All MLFs are modeled as binary features. MLFs that are not binary by nature, such as the number of comments of an issue, are quantized accordingly.

BOW: As reference for other MLFs in the experiment, the BOW[4] is employed.

Bi- and Tri-Grams: The BOW is expanded with bi-grams and tri-grams in the measurements. Both, bi- and tri-grams, denote words that occur together in tuples or triplets. Due to the fact that a huge amount of bi- and tri-grams can occur in the textual content of a single issue, the number of extracted bi- and tri-grams is restricted to 200, as rated by a chi-square-test (Bird, Klein, and Loper, 2009). This restriction, however, takes effect on the issue level only. Data fields and sentences usually contain less bi- and tri-grams.

SAO: To replicate the SAO pattern from (Vlas and Robinson, 2012), the Stanford Dependency Parser (Chen and Manning, 2014) is employed. As described for grammatical tagging in Figure 3.2 on page 25, the Stanford Dependency Parser outputs a directed acyclic graph of all words in a sentence. If a sentence contains a main verb, a check is performed for every subject and object whether a path between subject, main verb, and object exists. If and only if (1) a sentence contains a main verb, and (2) at least one subject and object, and (3) the subject(s) and object(s) are indirectly connected to the verb, these triplets are used as MLFs.

Keywords: The keyword list gathered during data annotation is used to check whether a simple gathering technique improves the ML model. In addition, a keyword list containing positive words, is employed since SFRs tend to be written gently and politely as discussed in Chapter 11.

Issue Meta-Data: In addition to the linguistic MLFs described above, MLFs derived from issue meta-data are considered in the experiments:

4 BOW is described in detail in Chapter 5.

1. the issue type (or, if appropriate, the issue tags),
2. the users that participated in the issue (the author and all users that commented or changed the issue),
3. the author of the issue,
4. the duration of the issue[5],
5. the number of comments to the issue[6],
6. all possible former states of the issue (e.g. open, closed, in development, in test, ...), and
7. the current status (open, in progress, closed, etc.).

Data Field Meta-Data: The following meta-data was extracted from ITS data fields: (1) the data field author, (2) whether the issue status changed while the data field was updated, and (3) the duration since the previous comment[7]

MACHINE LEARNING ALGORITHMS The following classifiers, all of which are fully introduced in Section 5.2, are used in this study. The selection represents the most prominent classifiers for text analysis tasks:

- Naive Bayes (NB)
- Multinomial Naive Bayes (MNB)
- Linear Support Vector Machine (SVM)
- Logistic Regression (LR)
- Stochastic Gradient Descent (SGD)
- Decision Tree (DT)
- Random Forrest (RF)

TEXT PREPROCESSING Five techniques, as introduced in Section 3.1.3 are employed to preprocess NL in the experiments:

1. Lowercasing
2. Stemming
3. Punctuation removal
4. Stop-word removal

5 Quantized as < 4 hours, $\leqslant 8$ hours, $\leqslant 1$ day, $\leqslant 10$ days or > 10 days.
6 Quantized as < 1, $\leqslant 2$, $\leqslant 4$, $\leqslant 6$, > 6; quantization made on the basis of Figure 11.1 on page 85.
7 Quantized as < 1h, $\leqslant 2$h, $\leqslant 4$h, $\leqslant 8$h, $\leqslant 1$d, $\leqslant 10$d.

5. Separation of technical noise. However, in contrast to the study presented in Chapter 14, an additional heuristic to detect the code mark-up for Redmine and Github was employed. This resulted in a nearly complete and error free detection of technical noise according to random samples[8].

15.1.4 *Evaluation Procedures*

Every experiment run applies all classifiers and combines them with different preprocessing settings and different sets of MLFs.

To evaluate the classifiers, a classical ten-fold-cross validation is employed. First, the issues are randomized and divided into 10 equal groups. Every experimental run then uses the issues of 9 groups for training. The issues of the remaining group serve as evaluation data. Every experiment runs 10 times, thus utilizing every issue 9 times as training data and one time as evaluation data.

For RQ 3.3 a different setup is necessary. The selected issues of three projects are used for training and the issues of the fourth project are used for evaluation. This results in a $\frac{75\%}{25\%}$ split.

15.2 RESULTS

To answer RQ 3.1, the performance among different ML algorithms with different preprocessing settings on linguistic MLFs is compared. Then, the best performing preprocessing setting is used in further evaluations to answer RQ 3.2 and RQ 3.3[9].

15.2.1 *Best Preprocessing Techniques*

As the combination of all preprocessing techniques results in 2^5 combinations, the reporting is constrained to the preprocessing settings defined in Table 15.2: the first setting uses no preprocessing as a reference. Lowering and stemming are considered best practices (Feldman and Sanger, 2006; Manning and Schütze, 1999) and are therefore included in all further preprocessing settings. Finally, all combinations of punctuation removal, stop-word removal, and separation of technical data are evaluated. These preprocessing techniques influence all the linguistic MLFs derived from the NL text, so that the preprocessing results are reported for all linguistic MLF-sets shown in Table 15.3.

Table 15.4 summarizes average and best achieved F_1 scores (1) for every of the 7 ML algorithms, (2) for every of the 3 labels, and (3) for every of the 3 levels of detail. The results on the left hand side of Ta-

8 Related regular expressions are given in Appendix D.

9 Preprocessing settings and MLF-sets need to be fixed for further experiments: calculating all permutations of preprocessing techniques, MLFs, levels of detail, and labels results in more than one year runtime on a standard laptop.

PREPRO. SETTING	LOWERING	STEMMING	PUNCTUATION REMOVAL	SEPARATION OF TECHNICAL DATA	STOPWORD REMOVAL
$\bar{1}$	-	-	-	-	-
$\bar{2}$	✓	✓	-	-	-
$\bar{3}$	✓	✓	-	-	✓
$\bar{4}$	✓	✓	-	✓	-
$\bar{5}$	✓	✓	-	✓	✓
$\bar{6}$	✓	✓	✓	-	-
$\bar{7}$	✓	✓	✓	-	✓
$\bar{8}$	✓	✓	✓	✓	-
$\bar{9}$	✓	✓	✓	✓	✓

Table 15.2: Evaluated Preprocessing Techniques and Linguistic MLF-Sets.

MLF-SET	BOW	BI- & TRI-GRAMS	SAO
$\ddot{1}$	✓	-	-
$\ddot{2}$	-	✓	-
$\ddot{3}$	-	-	✓
$\ddot{4}$	✓	✓	-
$\ddot{5}$	✓	-	✓
$\ddot{6}$	✓	✓	✓
$\ddot{7}$	-	✓	✓
$\ddot{8}$	✓	-	✓

Table 15.3: Evaluated Linguistic MLF-sets.

AGREED CASES		Max F_1, BOW Only			Avg.	Max F_1, All Ling. MLF Sets			Avg.
Level	Label	Conf.	Alg.	Value	F_1	Conf.	Alg.	Value	F_1
I	Req. Funct.	$\bar{6}$ / $\ddot{1}$	LR	0.495	0.279	$\bar{3}$ / $\ddot{6}$	SGD	0.643	0.168
	Clarification	$\bar{3}$ / $\ddot{1}$	SGD	0.401	0.201	$\bar{3}$ / $\ddot{1}$	SGD	0.401	0.124
	Solution	$\bar{3}$ / $\ddot{1}$	MNB	0.343	0.097	$\bar{2}$ / $\ddot{6}$	MNB	0.492	0.062
DF	Req. Funct.	$\bar{7}$ / $\ddot{1}$	MNB	0.130	0.088	$\bar{3}$ / $\ddot{6}$	MNB	0.195	0.054
	Clarification	$\bar{8}$ / $\ddot{1}$	SGD	0.102	0.089	$\bar{3}$ / $\ddot{6}$	MNB	0.119	0.055
	Solution	$\bar{7}$ / $\ddot{1}$	MNB	0.104	0.036	$\bar{6}$ / $\ddot{6}$	MNB	0.109	0.022
Se	Req. Funct.	$\bar{7}$ / $\ddot{1}$	MNB	0.078	0.048	$\bar{9}$ / $\ddot{6}$	MNB	0.106	0.030
	Clarification	$\bar{0}$ / $\ddot{1}$	SVN	0.055	0.047	$\bar{9}$ / $\ddot{5}$	MNB	0.078	0.029
	Solution	$\bar{7}$ / $\ddot{1}$	MNB	0.059	0.02	$\bar{6}$ / $\ddot{6}$	MNB	0.088	0.013
UNCERTAIN CASES		Max F_1, BOW Only			Avg.	Max F_1, All Ling. MLF Sets			Avg.
Level	Label	Conf.	Alg.	Value	F_1	Conf.	Alg.	Value	F_1
I	Req. Funct.	$\bar{7}$ / $\ddot{1}$	LR	0.752	0.501	$\bar{7}$ / $\ddot{4}$	LR	0.757	0.314
	Clarification	$\bar{7}$ / $\ddot{1}$	LR	0.551	0.412	$\bar{7}$ / $\ddot{4}$	LR	0.556	0.304
	Solution	$\bar{9}$ / $\ddot{1}$	SVM	0.498	0.258	$\bar{9}$ / $\ddot{1}$	SVM	0.498	0.181
DF	Req. Funct.	$\bar{9}$ / $\ddot{1}$	LR	0.325	0.272	$\bar{9}$ / $\ddot{8}$	SGD	0.33	0.174
	Clarification	$\bar{2}$ / $\ddot{1}$	LR	0.435	0.403	$\bar{2}$ / $\ddot{4}$	LR	0.435	0.298
	Solution	$\bar{9}$ / $\ddot{1}$	SGD	0.227	0.197	$\bar{9}$ / $\ddot{4}$	SGD	0.227	0.129
SE	Req. Funct.	$\bar{9}$ / $\ddot{1}$	LR	0.164	0.122	$\bar{9}$ / $\ddot{4}$	LR	0.164	0.078
	Clarification	$\bar{2}$ / $\ddot{1}$	LR	0.249	0.229	$\bar{2}$ / $\ddot{1}$	LR	0.249	0.161
	Solution	$\bar{9}$ / $\ddot{1}$	LR	0.117	0.099	$\bar{7}$ / $\ddot{6}$	MNB	0.125	0.066

Table 15.4: Average and Maximum F_1 Scores for Preprocessing.

ble 15.4 are achieved using the BOW MLF only, whereas the right hand side shows the results for all linguistic MLF-sets. For cells marked with a gray background in Table 15.4, additional details with respect to the preprocessing settings can be found in Figure 15.3. Figure 15.3 utilizes a combination of a scatter and a box plots to show the influence of all preprocessing settings together with the related standard deviation, median and mean, on four representative examples.

In comparison to other preprocessing settings, Setting $\bar{1}$ performed best in a single experiment run only. This indicates that lowering and stemming can be considered best practices in SFR detection, too. Preprocessing settings $\bar{7}$ and $\bar{9}$ deliver the best results in nine and ten cases respectively, covering 55% of the entire cases. Both, settings $\bar{7}$ an $\bar{9}$ employ all preprocessing techniques. Additionally, setting $\bar{7}$ separates technical data from NL. These results suggest that most preprocessing techniques have a positive influence on SFR detection. In ten out of 36 cases, or 28%, settings $\bar{2}$ and $\bar{3}$, which include only lowering, stemming, and stopword removal achieve the best F_1 scores. This implies that SFR detection can be approached without punctuation removal or the separation of technical data. However, there is no pattern in terms of detection level, label, or dataset, and it cannot be stated that a specific preprocessing setting should always be employed: e.g. the detailed examples in Figure 15.3 show that the two

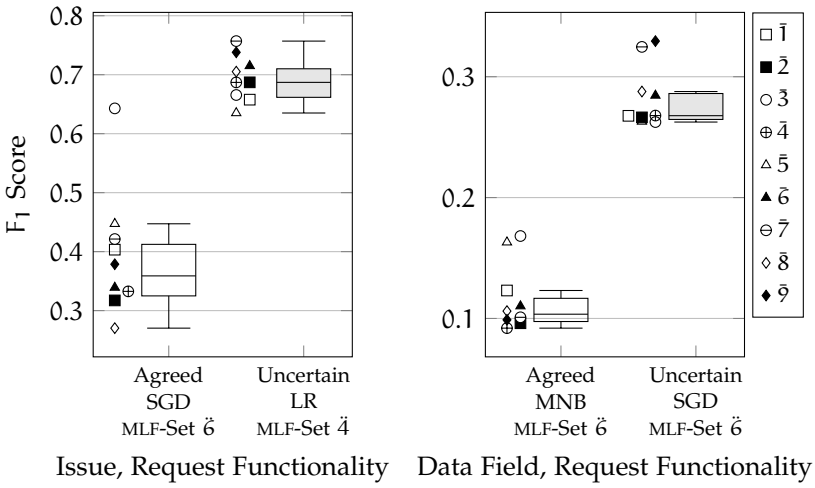


Figure 15.3: Detailed Examples for Preprocessing Setting Influences.

preprocessing settings that perform best on the agreed upon dataset, $\bar{2}$ [■] and $\bar{4}$ [⊕], perform bad on the uncertain cases and vice versa for the settings $\bar{7}$ [⊖] and $\bar{9}$ [◊]. But even with this uncertainty, *preprocessing setting $\bar{9}$ will be employed to answer the remaining RQs* for two reasons: (1) it generally shows a reasonable performance on random samples examined in detail, and (2) it results in the best reduction of MLF vectors, which saves memory, computation time, and accounts for better scalability.

In terms of ML models, Table 15.4 shows that MNB performs best for the agreed upon cases whereas LR performs better on the uncertain cases. There are two possible explanations for this:

1. LR is known, in essence, to outperform NB variants with more training data (Ng and Jordan, 2001), and

2. more annotations are contained in the uncertain dataset and thus more linguistic MLFs are included in the training data. MNB assigns independent weights to repetitive MLFs that correlate with the label and with every other due to a conditional independence assumption. In contrast LR compensates such inter-MLF correlations, which should improve the prediction rate.

15.2.2 *MLFs and Detection Levels*

The previous section includes various linguistic MLF-sets to study appropriate preprocessing settings for SFR detection. These are shown in Table 15.3. The right hand side in Table 15.4 shows that settings $\ddot{6}$ (the combination of all linguistic MLFs), $\ddot{4}$ (BOW combined with bi- and tri-grams) and $\ddot{1}$ (BOW only) gives the best results across all preprocessing settings. In fact, only those preprocessing settings that include the BOW MLF train usable models for SFR detection. E.g. on the agreed

MLF-SET	$\mathring{1}$	$\mathring{2}$	$\mathring{3}$	$\mathring{4}$	$\mathring{5}$	$\mathring{6}$	$\mathring{7}$	$\mathring{8}$	$\mathring{9}$	$\mathring{10}$	$\mathring{11}$	$\mathring{12}$	$\mathring{13}$	$\mathring{14}$	$\mathring{15}$	$\mathring{16}$	$\mathring{17}$	$\mathring{18}$
BOW	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
bi- & tri-grams	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓
SAO	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓
data field	-	-	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓
issue w/o type	-	-	-	-	✓	✓	-	-	-	-	✓	✓	-	-	✓	✓	-	-
issue with type	-	-	-	-	-	-	✓	✓	-	-	-	-	✓	✓	-	-	✓	✓
keywords	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	✓	✓	✓	✓

Table 15.5: Machine Learning Feature Sets.

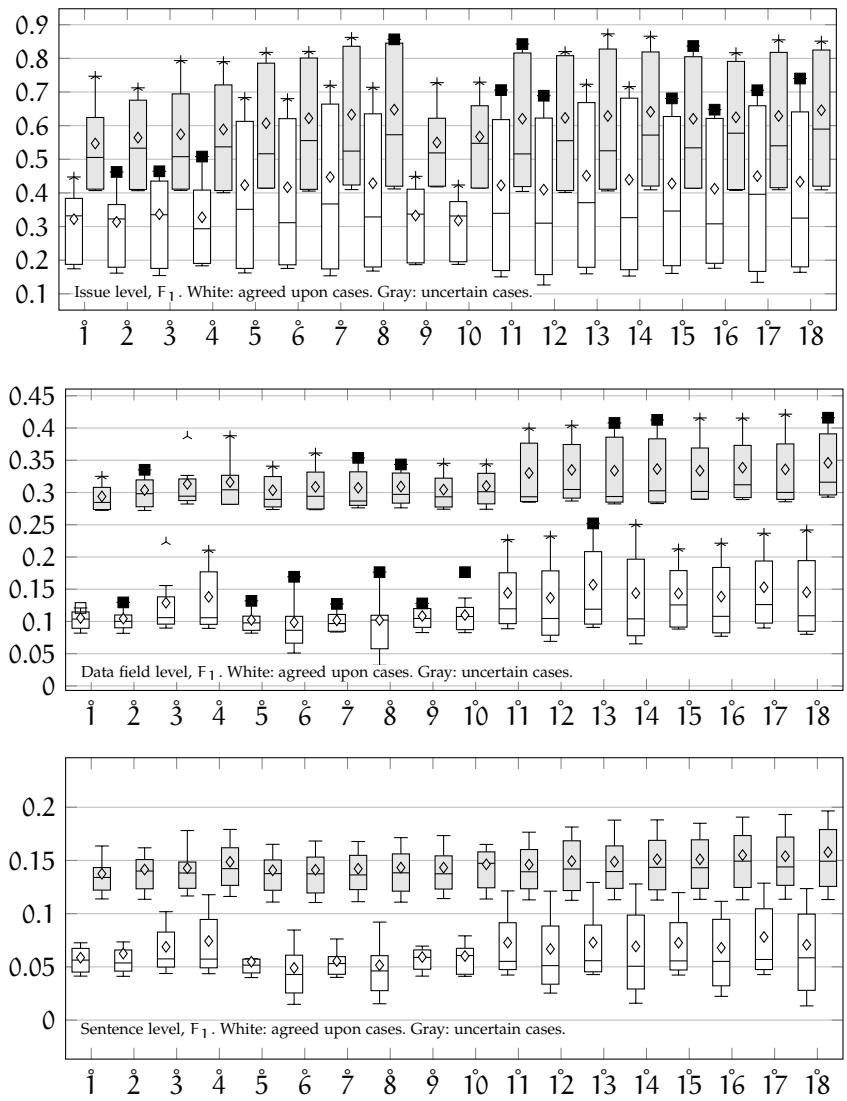
upon dataset, BOW results in a detection rate between 34% and 49% on the issue level and 10% to 13% on the data field level. In contrast, MLF-sets without BOW do not exceed 10% on any level of detail regardless of label, preprocessing techniques, and ML algorithms. However, the combination of linguistic MLFs generally improves detection rates (e.g. MLF-sets $\ddot{4}$, $\ddot{5}$, and $\ddot{6}$).

Table 15.5 summarizes the MLF-sets that are used for further experiments. The linguistic MLF-sets $\ddot{1}$ and $\ddot{2}$ are included in all further MLF-combinations: MLF-set $\ddot{1}$ can be derived easily and quickly, having a competitive performance, whereas MLF-set $\ddot{6}$ delivers the best results. MLF-sets $\mathring{3}$ to $\mathring{10}$ extend the linguistic MLFs with data field meta-data, issue meta-data, and keywords. The issue type or tag should be a very strong indicator for a functionality request. Therefore, the issue meta-data is split up into two separate sets: (1) excluding the issue type or tags, and (2) including the issue type or tags. MLF-sets $\mathring{11}$ - $\mathring{18}$ are combinations of linguistic MLFs, data field meta-data, issue meta-data and keywords.

For every MLF-set, the box plots in Figure 15.4 visualizes the F_1 scores over all ML models. White boxes represent the agreed upon cases, gray boxes the uncertain cases. In addition, a small symbol indicates the algorithms that performed best with respect to F_1 scores. The scatter plots in Figure 15.5 visualize the $\mathrm{MAX}(R), P_{\geqslant p}$ scores for the ML models that achieved the best score.

The inclusion of keywords hardly impacts the results, clearly evident in the comparison between MLF-sets $\mathring{1}$ and $\mathring{2}$ with $\mathring{9}$ and $\mathring{10}$. The inclusion of data fields, on the other hand, and especially ITS meta-data do have a positive impact. However, additional combinations of data field and ITS meta-data do not necessarily improve the results. A slight increase can be noticed on the data field level by comparing only the ITS meta-data MLFs ($\mathring{5} - \mathring{10}$) with combined MLFs ($\mathring{10} - \mathring{18}$).

Figure 15.5 shows that linguistic MLFs are generally sufficient whenever recall is to be maximized. On the issue level, the SVM model improves the precision by considering ITS meta-data (see $\mathring{5} - \mathring{8}$), while leaving the recall almost unaffected. Similar improvement can be noticed in the F_1 scores on the left hand side for SGD and LR algorithms. Remarkably, the issue type impacts detection results little. Thus the



Average values: ◇
Best algorithm markers: □ MNB, ■ SGD, ● NB, ⋆ SVM, ▲ DT, ⅄ LR, ⬠ RF

Figure 15.4: F_1 Scores for *Request Functionality* Detection: Comparison of MLF-sets for Different Scopes with Multiple Classifiers.

issue type, at least in the projects researched in this thesis, is a weaker predictor than initially assumed.

Overall, the combination of all MLF-types delivers the best performance in terms of F_1 score, whereas the linguistic MLFs on their own are competitive in terms of $MAX(R), P_{\geqslant p}$ scores.

The F_1 and $MAX(R), P_{\geqslant p}$ scores on the issue level are higher compared to other levels. However, the data field level delivers competitive scores with respect to the $MAX(R), P_{\geqslant p}$ measure, especially on the uncertain dataset. This is important for the practical application of the approach: detecting SFRs on the data field level implies less

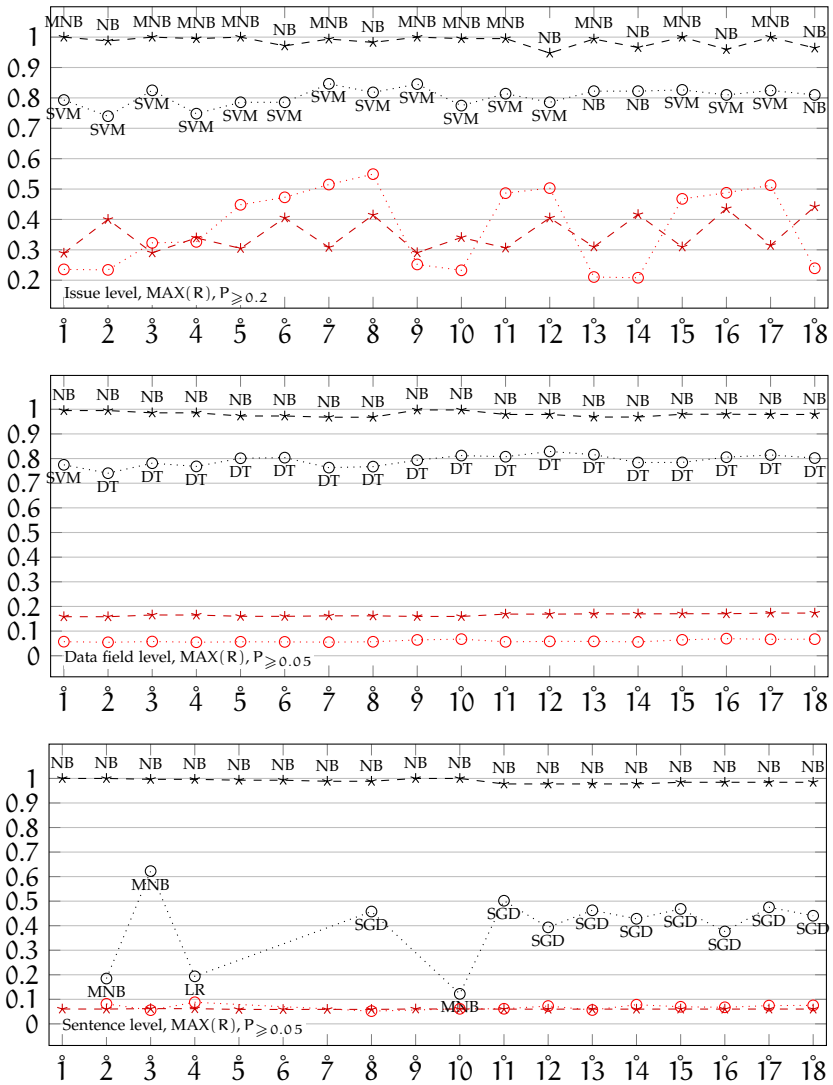


Figure 15.5: MAX(R), $P_{\geqslant 0.2}$ and MAX(R), $P_{\geqslant 0.05}$ Scores for *Request Functionality* Detection: Comparison of MLF-sets for Different Scopes with Multiple Classifiers.

manual work than a detection on the issue level. The worst results are generated on the sentence level. For example for MAX(R), $P_{\geqslant 0.05}$ the recall is only slightly over 60% in the agreed upon cases as shown in Figure 15.5. For the uncertain cases, however, a recall of 100% can be achieved for MAX(R), $P_{\geqslant 0.05}$ even on the sentence level with F_1 scores of about 11%.

The measurements reported on in this section are exemplary for the other two labels. *Clarifications* are detected with similar P and R. *Solutions* are detected with inferior detection rates compared to the other labels, but the results show the same characteristics with respect to the MLF-sets. Table 15.6 summarizes the best achieved F_1 and MAX(R), $P_{\geqslant p}$ scores with the related ML models and MLF-sets for all

Level	Label	MLF-Set	Algorithm	Values			MLF-Set	Algorithm	Values		
		F_1: AGREED		F_1			UNCERTAIN		F_1		
I	RF	18	SGD	0.74			13	LR	0.87		
	C	14	SVM	0.74			9	LR	0.78		
	S	15	SGD	0.54			7	SVM	0.75		
DF	RF	13	SGD	0.25			17	LR	0.42		
	C	4	MNB	0.15			17	SVM	0.45		
	S	9	MNB	0.09			17	SGD	0.27		
Se	RF	13	LR	0.13			18	SGD	0.19		
	C	4	MNB	0.08			4	MNB	0.25		
	S	1	MNB	0.05			4	MNB	0.13		
MAX(R), $P_{\geqslant p}$: AGREED				p	R	P	UNCERTAIN		p	R	P
I	RF	7	SVM	0.2	0.85	0.51	1	MNB	0.2	1.0	0.29
	C	7	SVM	0.2	0.84	0.66	7	NB	0.2	1.0	0.28
	S	17	SVM	0.2	0.74	0.39	7	SVM	0.2	0.78	0.75
DF	RF	12	DT	0.05	0.83	0.06	9	NB	0.05	0.99	0.16
	C	14	NB	0.05	0.99	0.05	3	NB	0.05	0.99	0.25
	S	1	MNB	0.05	0.29	0.07	17	NB	0.05	0.98	0.11
Se	RF	3	MNB	0.05	0.62	0.06	3	NB	0.05	1.0	0.06
	C	4	MNB	0.05	0.18	0.05	3	NB	0.05	0.99	0.13
	S	2	MNB	0.05	0.03	0.1	1	MNB	0.05	0.93	0.06

with: RF=*Request Functionality*, C=*Clarification*, S=*Solution*

Table 15.6: Best F_1 and MAX(R), $P_{\geqslant p}$ Scores for All Levels and Labels.

the labels on every level of detail. Appendix C includes complementary tables and figures for *Clarification* and *Solution* detection.

Finally, combining MLF-sets generally increases the standard deviation for the F_1 scores across all models as shown in Figure 15.4. Hence, the algorithm choice (or an evaluation of multiple algorithms) is important whenever features are combined otherwise results may deteriorate.

15.2.3 *Cross-training*

To answer this RQ, three of the four project datasets were used for training and the fourth project for detection. Table 15.7 shows the best achieved F_1 and MAX(R), $P_{\geqslant p}$ scores with according MLF-sets for the *request functionality* label. Again, similar to RQ 3.2, *request functionality* is representative for the other labels. Related results can be found in Appendix C.

Comparing these results to the results with 10-fold-cross evaluation from Table 15.6, it can be seen that cross-training delivers very similar

Project	Level	MLF-Set	Algorithm	Values		MLF-Set	Algorithm	Values	
		AGREED		F_1	avg.	UNCERTAIN		F_1	avg.
cg	I	11	SGD	0.74	0.72	15	SGD	0.85	0.87
li		7	LR	0.76		6	LR	0.84	
ra		5	LR	0.67		11	SGD	0.86	
re		14	SVM	0.70		17	SGD	0.91	
cg	DF	13	SGD	0.29	0.31	17	SGD	0.26	0.39
li		18	LR	0.42		18	LR	0.41	
ra		14	SGD	0.24		15	SGD	0.36	
re		14	SGD	0.29		14	SGD	0.52	
cg	Se	4	MNB	0.17	0.19	11	SGD	0.13	0.19
li		18	SGD	0.27		17	LR	0.16	
ra		13	LR	0.16		16	SGD	0.18	
re		3	MNB	0.16		4	MNB	0.27	
MAX(R), $P_{\geqslant p}$		AGREED		p	R	P	UNCERTAIN	p	R	P
cg	I	3	NB	0.2	0.89	0.22	1	MNB	0.2	1.00	0.24
li		3	SGD	0.2	1.00	0.38	7	NB	0.2	1.00	0.21
ra		1	SVM	0.2	0.80	0.22	1	NB	0.2	1.00	0.33
re		13	NB	0.2	0.97	0.27	13	NB	0.2	1.00	0.59
cg	DF	1	RF	0.05	0.71	0.06	9	NB	0.05	0.99	0.13
li		4	DT	0.05	1.00	0.06	7	NB	0.05	1.00	0.08
ra		13	SGD	0.05	0.53	0.14	1	NB	0.05	1.00	0.10
re		9	NB	0.05	1.00	0.06	7	NB	0.05	1.00	0.19
cg	Se	1	MNB	0.05	0.52	0.05	5	NB	0.05	1.00	0.05
li		13	SGD	0.05	0.89	0.07	16	DT	0.05	0.93	0.05
ra		12	SGD	0.05	0.40	0.06	1	MNB	0.05	0.98	0.06
re		3	SGD	0.05	0.71	0.06	7	NB	0.05	1.00	0.07

with: cg=c:geo, li=Lighttpd, ra=Radiant, re=Redmine

Table 15.7: Best Cross-Training F_1 and MAX(R), $P_{\geqslant p}$ Scores for Request Functionality.

F_1 and $MAX(R), P_{\geqslant p}$ scores with a variability of $\pm 5\%$, even though a lower amount of training data is available for cross-training than for ten-fold-cross validation.

15.3 DISCUSSION

This section first discusses the implication of the above results for ITSoFD in practice. Then it discusses the main implications of the study for future research on SFR detection in ITSs:

PRACTICAL IMPLICATIONS For an adoption of ITSoFD into practice, the results are satisfactory. On both, the level of issues and the level of data fields, recall rates close to 100% are achieved. In addition, the amount of false positives delivered by the approach is significantly lower than the amount of data that needs to be worked through when the same classification is done manually. Hence the approach saves time and money in comparison to a manual classification.

PREPROCESSING TECHNIQUES SHOULD BE EMPLOYED The application of preprocessing techniques improves F_1 and $MAX(R), P_{\geqslant p}$ scores in the context of SFR detection as shown in Section 15.2.1. Even stop-word removal improves the results although coders considered stop-words such as *should*, *could*, or *would* relevant for classification.

ENOUGH ANNOTATIONS SHOULD BE AVAILABLE FOR TRAINING AND EVALUATION Section 15.2 reports on models trained on $36 \times \frac{9}{10}$, $76 \times \frac{9}{10}$ and $100 \times \frac{9}{10}$ annotations for the agreed upon dataset. The F_1 and $MAX(R), P_{\geqslant p}$ scores increase relative to the amount of training.

SIMPLE LINGUISTIC FEATURES ARE SUFFICIENT Sets that add bi-, tri-grams and SAO to BOW (i.e. sets with even numbers in Figure 15.4 and Figure 15.4) show almost no improvement. If computation time or the amount of features need to be reduced, complexity can be downsized at this point.

MLFS DERIVED FROM META-DATA IMPROVE DETECTION RATES Sets $\mathring{1}1$-$\mathring{1}8$ in Figure 15.4 show that the inclusion of MLFs derived from meta-data improve detection rates. Further research (e.g. ML feature selection techniques (Guyon and Elisseeff, 2003)) is needed to identify the exact MLFs with the highest impact.

THE COMBINATION OF ML ALGORITHMS AND MLF-SETS IS IMPORTANT Different ML algorithms work differently. For example, NB treats correlating MLFs independently, LR compensates such cor-

relations (Manning and Schütze, 1999). The lower whiskers in Figure 15.4 indicate that some ML algorithms do not profit from additional features. Hence, the combination of ML algorithms and MLF-sets is important to consider.

ML ALGORITHMS SHOULD BE SELECTED IN ACCORDANCE TO THE DATA MINING GOAL Comparing Figure 15.4 with Figure 15.5 reveals that different ML algorithms are responsible for the best F_1 and the best $MAX(R), P_{\geqslant p}$ sores. Consequently, algorithms performing best in order to maximize the SFR detection rate might not perform best in order to balance detection rate and precision.

TRAINED MODELS CAN BE RE-USED FOR OTHER PROJECTS Section 15.2.3 reveals that cross-training works competitive to ten-fold-cross validation for *request functionality* and *clarification* labels. However, this does not not hold true for every project and thus needs further evaluation or replication by additional studies.

IN SOME PROJECTS SFRS ARE LIKELY COMPOSED OF NL PATTERNS In five cases setting $\mathring{9}$ (BOW and manually compiled keyword lists) achieved the best F_1 and $MAX(R), P_{\geqslant p}$ scores as shown in Tables 15.6 and 15.7. This indicates that at least some *request functionalities* are described with reoccurring words or certain NL patterns. As for other requirements (Vlas and Robinson, 2012), such patterns would be a powerful heuristic and/or ML feature for SFR detection.

USING ONLY THE AGREED UPON CASES FROM DUAL CODED DATA CAN BE INFERIOR FOR ML In almost every case, better predictions can be made for the uncertain cases. As already mentioned, the annotators did not annotate sentences they felt uncertain about so that these sentences did not make it in the agreed upon dataset. However, I may argue that the annotators, in general, annotated SFRs correctly. Therefore, more presumably correct data might be found in the uncertain dataset. Assuming that the algorithm detects most of these uncertain cases, this leads to fewer false positives and thus a higher precision, which is underpinned by the $MAX(R), P_{\geqslant 0.05}$ scores for the data field level in Figure 15.5. This implication is advantageous for the practical applicability of the approach: multiple coding is usually not applicable in industry, or generally in practice and the algorithm seems to replicate the 'opinion' of a single coder even better.

EXPERIMENT RUNTIME The runtime on a standard laptop is reasonable, considering that the code was only slightly optimized for speed. For example, an experiment using BOW features, three classifiers, and 599 issues takes < 10 seconds. An experimental run including all MLFs takes < 10 minutes and a higher level of detail decreases

the runtime. These measures exclude the calculation of typed dependencies, which takes about an additional hour and is needed only for the SAO feature[10]. However, ten-fold-cross validation and permutations of different MLF-sets quickly increase the overall runtime.

Limitations of these implications are discussed in the next section.

15.4 THREATS TO VALIDITY

INTERNAL VALIDITY Manual data annotation always involves the risk that human coders do make mistakes. This can result in unusable ML models and thus influence predictions and results. To mitigate this risk best practices in content analysis (Neuendorf, 2002) were deployed. In particular, a coding guidebook was established and annotations were discussed on test data before the actual coding. Although the coders worked on the level of sentences, they achieved reasonable kappa scores. Finally, one dataset with only those annotations on which two coders agreed was created. However, the coders had a lower agreement on comments. This could be due to increasing tiredness or even inadvertence. Annotating up to 50 comments in a single issue is an arduous task. A low agreement might lead to missing training and validation data and thus to decreasing detection rates and/or increasing false positive rates.

However, the assessment whether a sentence describes a SFR or not is arguably subjective. Although the agreed upon dataset has a higher scientific quality, using the data created by a single annotator is not necessarily bad. E.g. the perception of a single analyst might be exactly what an ML model should replicate when applied in an industry project. This subject was addressed by including the uncertain cases in the experiments.

Finally, the overall size of the gold standard is limited. Although over 10,000 sentences in over 4500 ITS data fields were analyzed, the gold standard contains only 76 *request functionality*, 36 *solution* and 100 *clarification* labels for the agreed upon cases. Although promising results were received on the issue and data field level, other research indicates that more than 100 labeled instances are necessary for a reliable classification (Maalej and Nabil, 2015). Hence, this study should be seen as a first step in SFR detection and needs further replication or extension studies.

EXTERNAL VALIDITY Considering the external validity, it cannot be ensured that the results can be transferred to ITS data of other projects[11]. It is still likely that the overall approach is transferable

10 Typed dependencies are cached, yielding a > factor 100 speedup.

11 In an industrial ITS I encountered "coffee break issues". Those issues include information similar to "As discussed during coffee break". Needless to say, this is insufficient for text analysis.

since projects with a broad range of target groups, programming languages, etc. were employed intentionally. In a more consistent dataset, even better results can be expected.

15.5 CONCLUSION

In this chapter multiple preprocessing techniques, ML algorithms, and MLFs are evaluated to detect SFRs in ITSs. With the introduction of the $\mathrm{MAX}(R), P_{\geqslant p}$ measure, it is shown that some algorithms maximize the F_1 score whereas others maximize recall. Furthermore, the results suggest that SFR detection should be approached on the level of issues or data fields and that the exact sentences describing the SFR are hard to find.

16

RECOVERING RELATED ISSUES IN ISSUE TRACKING SYSTEMS – AN EMPIRICAL STUDY

"The case study contains no greater bias toward verification of the researcher's preconceived notions than other methods of inquiry. On the contrary, experience indicates that the case study contains a greater bias toward falsification of preconceived notions than toward verification."
- (Flyvbjerg, 2006)

The empirical study presented in this chapter investigates Research Question 4 defined in Section 1.3:

> RQ 4 Do trace retrieval algorithms perform effectively on ITS data?

The study evaluates the trace retrieval component of ITSoFD described in Section 13.2.4. Thus, the main goal of this chapter is to study whether trace retrieval techniques and enhancements to those techniques can be applied to ITSs.

The next section describes details of the study design and the above research question is systematically refined in three fine grained RQs in Section 16.1.1. The results are presented in detail in Section 16.2 and discussed in Section 16.3. Section 16.4 discusses the threats to validity and the final section concludes this chapter.

16.1 STUDY DESIGN

Overall, the study setup has three steps:

1. The extraction and preparation of the data,
2. the manual creation of a gold standard traceability matrix to evaluate the experiment results, and
3. the automated trace retrieval using different algorithms, different preprocessing techniques, and varying term weighting.

Details of this process are described in the following sections and an overview is given in Figure 16.1.

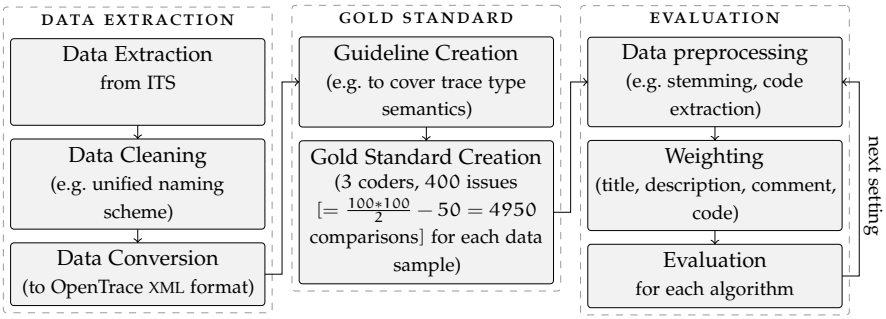


Figure 16.1: Study Setup.

16.1.1 *Research Questions*

The main question of this chapter, whether IR algorithms for trace recovery perform effectively on ITS data, is divided into the following four RQs:

RQ 4.1 How do IR algorithms for automated traceability perform on ITS data in comparison to related work on structured RAs?
Expected is a) a worse results in comparison to related work on RAs, due to little structure and much noise in ITS data, and b) that BM25[+/L] variants perform competitive since BM25 is often used as a baseline to evaluate new IR algorithms.

RQ 4.2 How do results vary if ITS-specific preprocessing and weighting is applied?
Expected is that removing ITS specific noise improves the results for all datasets.

RQ 4.3 How do results vary for different trace and issue types?
E.g. Heck and Zaidman, 2014; Runeson, Alexandersson, and Nyholm, 2007; or Wang et al., 2008 used IR algorithms on bug report duplicates. Since duplicates usually have a high similarity, good results for duplicates can be expected.

RQ 4.4 How do results vary between different projects?
Experiments are run with the data of four projects with distinct properties. Expected is a wide range of results due to these differences.

This particular study reports on trace retrieval for the trace types *duplicate* and *generic*. A duplicate relation exists between two issues, if both issues describe exactly the same SF. A *generic* relation exists, if two issues refer to the same SF. Such a *generic* relation can for example be used to determine the total amount of time and money that was spent to implement an SF and fix all potential bugs that were introduced with this SF.

16.1.2 *Data*

All projects researched in this thesis are described in detail in Section 8.1. The data of following four projects is utilized in this study. These are the same projects studied for SFR detection in the previous chapter:

1. c:geo
2. lighttpd
3. Radiant
4. Redmine

All researched projects of this thesis are described in detail in Section 8.1. An excerpt of the data researched in this particular study is shown in Figure 16.2. The figure shows the gold standard creation tool developed for this study. The tool is introduced in the following paragraph.

DATA PREPARATION In contrast to the previous study from Chapter 15, where early or older issues were sampled to include many SFRs in the dataset, 100 consecutive issues per project (in total 400 issues) were extracted in this study. It is more likely that consecutive issues are related, e.g. because they refer to the same SF or because induced bugs are reported promptly. Hence, the possibility to find meaningful traces is higher in a consecutive set of issues than in randomly selected samples. The extraction process followed existing links to other issues in a breadth-first search manner to make sure that the extracted dataset includes traces. Existing links were automatically parsed and collected into a traceability matrix. This traceability matrix is referred to as Developer Trace Matrix (DTM). Beside the NL, data fields, and the existing traces, meta-data such as authors, date- and time-stamps, the issue status, or issue IDs were extracted.

The selection includes features, bugs, and uncategorized issues. However, as discussed in Chapter 11 the projects that rely on the Redmine ITS categorize all issues, whereas the ones using the GitHub ITS seldom employ issue categories.

GOLD STANDARD TRACE MATRICES The first, third, and fourth author of the publication by Merten et al., 2016a created the Gold Standard Trace Matrix (GSTM). For this task title, description, and comments of each issue were manually compared to every other issue. Since 100 issues per project were extracted, this implies $\frac{100*100}{2} - 50 = 4950$ manual comparisons[1]. To have semantically similar gold standards for each project, a code of conduct was developed that prescribed e.g. when a generic trace should be created or when an issue should be treated as duplicate (the description of both issues describes exactly the same bug or requirement).

Since concentration quickly declines in such monotonous tasks, the comparisons were aided by a tool especially created for this purpose. It supports defining related and unrelated issues by simple keyboard shortcuts as well as saving and resuming the work. The tool is compatible with the OpenTrace format, so that reference trace matrices for OpenTrace can be created and experiment outcomes or intermediate steps can be visualized using GATE. At large, a GSTM for one project could be created in two and a half business days with this tool. A screenshot of the tool is shown in Figure 16.2. On the left

1 Comparisons were made from issue A to B and B to A at the same time and the trace direction was set accordingly.

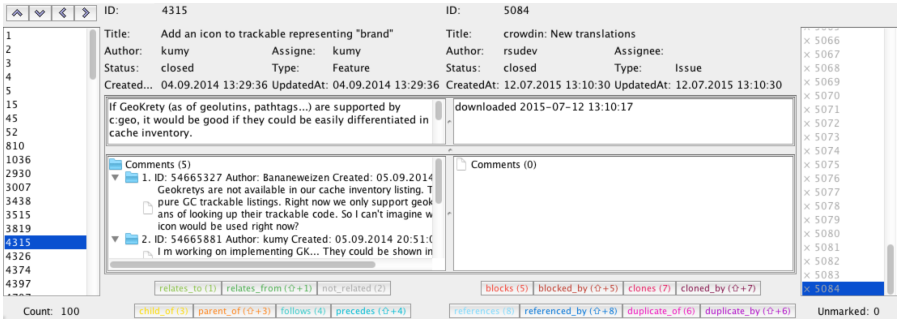


Figure 16.2: Gold Standard Creation Tool.

hand side c:geo issue #4315 is shown. The right hand side presents all other issues to the user. In this screenshot all issues are marked as 'not related'. Each trace type is color coded so that it can be visually distinguished in the tool. After every trace definition, the tool automatically advances the right hand side to the next issue. Hence a single keystroke defines a trace. In case of an erroneous trace definition, the user can scroll though the issues issues manually using the arrow keys for a quick revision.

In general, the GSTMs contain more traces than the DTMs (see Table 16.1). A manual analysis revealed that developers often miss (or simply do not want to create) traces or create relations between issues that are actually not related. The following are examples why GSTMs and DTMs differ:

1. Eight out of the 100 issues in the c:geo dataset were created automatically by a bot that manages translations for internationalization. Although these issues are related, they were not automatically marked as related. There is also a comment on how internationalization should be handled in another issue.

2. Some traces in the Redmine based projects do not follow the correct syntax and are therefore missed by a parser. E.g. in lighttpd #2378 two issues are referenced by an URL (`http://...id`) and not by # < id >, as specified by the Redmine Wiki Syntax.

3. Links are often vague and unconfirmed in developer traces. E.g. c:geo #5063 says that the issue "could be related to #4978 [...] but I couldn't find a clear scenario to reproduce this". No evidence could be found that suggests marking these issues as related in the gold standard. However, a link was placed by the developers.

4. Issue #5035 in c:geo contains a reference to #3550 to say that a bug occurred before the other bug was reported (the trace semantic in this case is: "occurred likely before"). There is, however, no real relation between the bugs except the occurrence,

therefore these issues were not marked as related in the gold standard.

5. The Radiant project simply did not employ many manual traces but traces could be found by the annotators.

	PROJECTS			
# OF RELATIONS	C:GEO	LIGHTTPD	RADIANT	REDMINE
DTM generic	59	11	8	60
GSTM generic	102	18	55	94
GSTM duplicates	2	3	-	5
overlapping	30	9	5	45

Table 16.1: Extracted Traces Versus Gold Standard.

16.1.3 *Algorithms and Settings*

In the experiment, multiple term weighting schemes for the ITS data fields and different preprocessing methods are combined with the IR algorithms VSM, LSA, BM25, BM25+ and BM25L. As preprocessing techniques stop word removal, lowering, and stemming are used. These are referred to as *standard preprocessing*. Due to the experience in issue preprocessing from the previous chapter, standard preprocessing is applied as is and not split and further. In addition *ITS-specific preprocessing* is employed. For the ITS-specific preprocessing, technical noise is removed and the regions marked as code are extracted and separated from the NL. Thus, different term weights can be applied to each ITS data field and to the technical data. Table 16.2 gives an overview of all IR algorithms and preprocessing methods and Table 16.3 shows the term weights and rationales for every weighting scheme.

ALGORITHM	SETTINGS	PREPROCESSING	SETTINGS
BM25	Pure, +, L	Standard Preprocessing	
VSM	TF-IDF	Stemming	on/off
LSA	cos measure	Stop Word Removal	on/off
		ITS-specific Preprocessing	
		Noise Removal	on/off
		Code Extraction	on/off

Table 16.2: Algorithms and Preprocessing Settings.

WEIGHT				RATIONALE / HYPOTHESIS
TITLE	DESCRIPTION	COMMENTS	CODE	
1	1	1	1	Unaltered algorithm
1	1	1	0	without considering code
1	1	0	0	also without comments
2	1	1	1	Title more important
2	1	1	0	without considering code
1	2	1	1	Description more important
1	1	1	2	Code more important
8	4	2	1	Most important information first
4	2	1	0	without considering code
2	1	0	0	also without comments

Table 16.3: Data Fields Weights.

16.1.4 *Evaluation Procedures*

The result of the IR-based trace retrieval techniques are compared to the GSTM. For every trace retrieval technique $trace_t$ is computed with different thresholds t in order to maximize precision, recall, F_1 and F_2 measure. Results are presented as F_2 and F_1 measure in general. However, maximizing recall is often desirable in practice, because it is simpler to remove wrong links manually than to find correct links manually. Therefore, maximized R with the corresponding precision is discussed, too, similarly to the $MAX(R), P_{\geqslant p}$ measure used in the previous chapter.

16.2 RESULTS

As stated in Section 16.1, a comparison to the GSTM results in more authentic and accurate measurements than a comparison to the DTM. It also yields better results: F_1 and F_2 both increase about 9% in average computed on unprocessed datasets[2]. A manual inspection revealed that this increase materializes due to the flaws in the DTM as discussed in Section 16.1.2, especially because of missing traces. Therefore, the results in this chapter are reported in comparison to the GSTM.

16.2.1 *IR Algorithm Performance on ITS Data*

Figure 16.3 shows an evaluation of all algorithms with respect to the GSTMs for all projects with and without *standard preprocessing*. The differences per project are significant with 30% for F_1 and 27% for F_2. It can be seen that standard preprocessing does not have a clear

2 Unprocessed means that neither preprocessing techniques are applied nor weighting is used.

positive impact on the results. Although if only slightly, a negative impact on some of the project/algorithm combinations is noticeable.

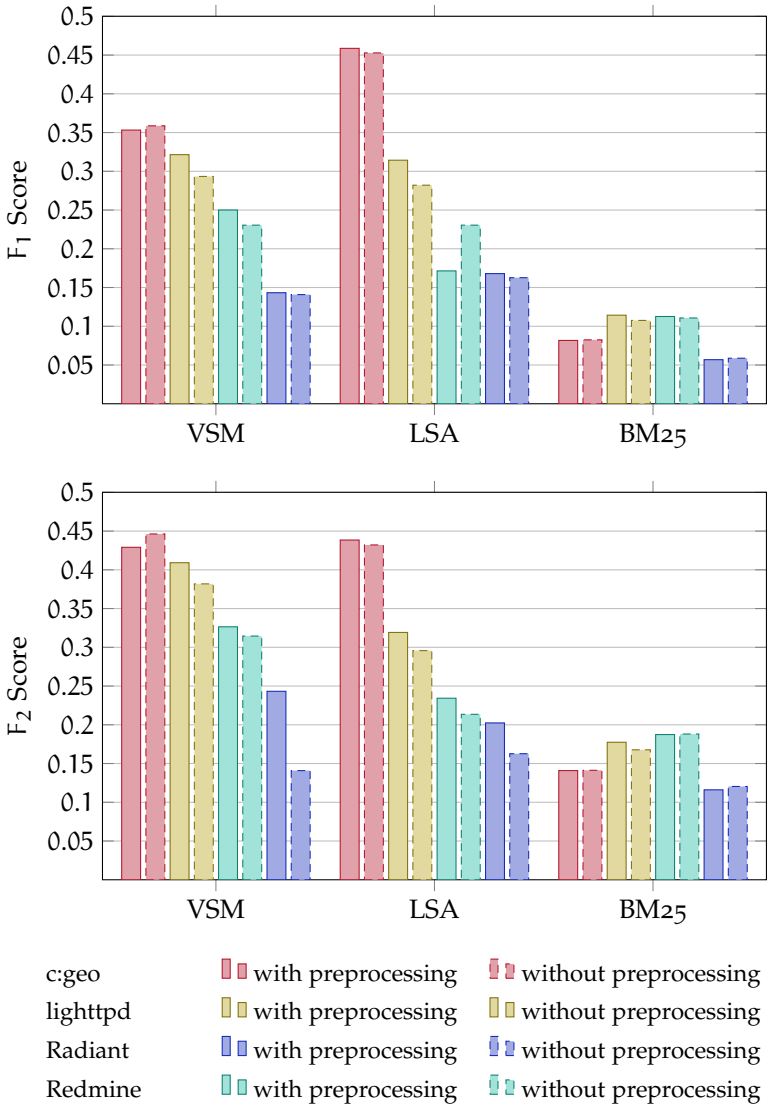


Figure 16.3: Best F_1 and F_2 Scores for Every IR Algorithm.

On a side note, the experiment supports the claim of Heck and Zaidman, 2014 that removing stop-words is not always beneficial on ITS data: Former experiments included different stop word lists and it was found that a small list removing essentially only pronouns works best.

In terms of algorithms, no variant of BM25 competed for the best results. The best F_2 measures of all BM25 variants varied from 0.09 to 0.19 over all projects, independently of standard preprocessing. When maximizing R to 1, P does not cross a 2% barrier for any algorithm. Even for $R \geqslant 0.9$, P is still < 0.05. All in all, the results are not good according to the definition from Table 6.1 in Section 6.1. This is inde-

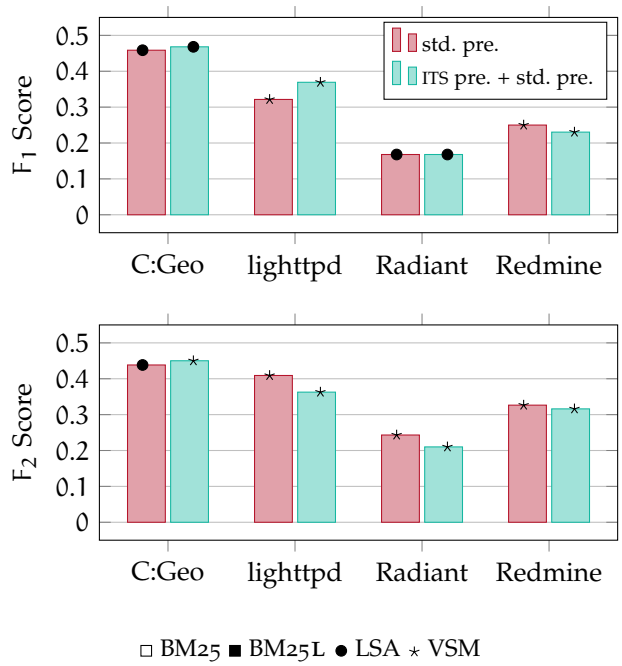


Figure 16.4: Best Results With and Without Removing Noise.

pendently of standard preprocessing and trace retrieval on ITS data cannot compete with related work on structured RAs.

With respect to preprocessing the results decrease only slightly in a few cases. However, this negative impact is negligible and the remaining measurements are reported with the standard preprocessing techniques enabled.

16.2.2 *ITS-specific Preprocessing and Weighting*

This RQ2 investigates in the influence of *ITS-specific preprocessing* and *ITS data field-specific term weighting* in contrast to *standard preprocessing*.

Contrary to the expectations, ITS-specific preprocessing impacts only c:geo clearly positive as shown in Figure 16.4. For the other projects, a positive impact is achieved in terms of F_1 measure only. Since preprocessing always removes data, it can have a negative impact on recall. This involves as a slight decrease of the F_2 measure for three of the projects (4% for lighttpd, 2% for Radiant, and 1% for Redmine). Overall however, precision improves with ITS-specific preprocessing.

Figure 16.5 shows the influence of different term weights in each of the projects. For a better comparison the results are shown with *standard* and *ITS-specific preprocessing* enabled. The left axis represents the term weight factors for: *Title - Description - Comments - Code*. In contrast to ITS-specific preprocessing Figure 16.5 shows that some term weights clearly performed best. In general, the weighting schemes that stress the title yield better results. In addition, the figure also

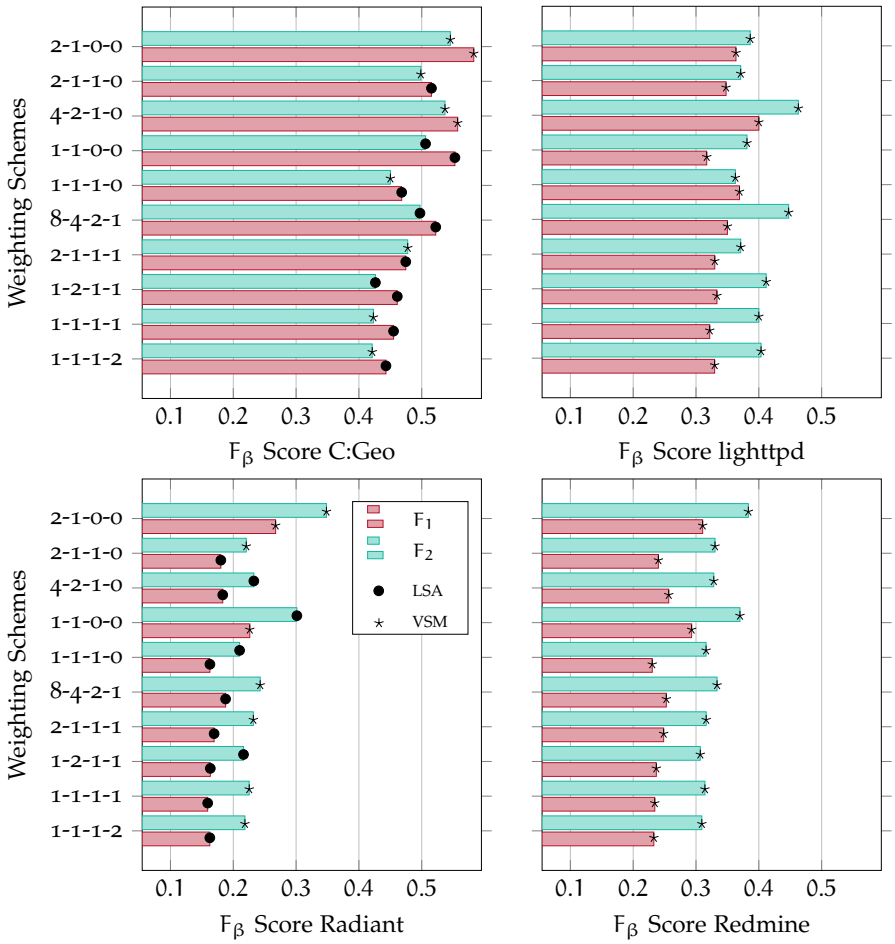


Figure 16.5: Influence of Term Weighting.

shows that code should not be considered by IR algorithms for trace retrieval: Term weights of 0 for code yield the best results.

16.2.3 *Trace Types and Issue Types*

ISSUE TYPES Table 16.4 shows the best achievable results for F_1, F_2 and R on fully preprocessed datasets. The best results per issue type are printed in bold font. Since the Radiant dataset does not provide information on issue types, it is excluded in Table 16.4.

Trace retrieval from feature to bug issues works best for the lighttpd dataset. For Redmine retrieval between features worked best and for c:geo retrieval between bugs worked best; here, however, retrieval for other cases is much lower. Interestingly, there was no issue type, that worked best or worst for all projects.

TRACE TYPES Table 16.5 compares the best achievable results for $trace_t : I \times I$ and $trace_t^{duplicate} : I \times I$. The comparison is restricted to generic relations and duplicates, since other annotated trace types in

the GSTM[3] left too much room for interpretation by the annotators. E.g. it is hard to define when exactly an issue "blocks" another issue, without detailed knowledge of the project.

Table 16.5 shows that duplicate issues are detected competitively for c:geo and Redmine and rather poorly for lighttpd. The latter contradicts the expectations for this RQ. However, a manual inspection of the data showed that duplicated issues often use different words to express the same matter, similar to the example given in Figure 2.3 in Section 2.2. This can only be resolved by domain knowledge and/or knowledge of domain-dependent synonyms. Both of which cannot be handled by standard IR algorithms without additional effort. Note, that it cannot be reported on the Radiant dataset, since the GSTM does not contain any duplicates.

16.2.4 *Results per Project*

Table 16.6 summarizes the best results per project for the $F_{1,2}$ scores, the best recall with the according precision ($R(P)$), and the settings that achieve these results. A baseline is represented by the best performing algorithm with *standard preprocessing* but without *ITS-specific preprocessing* and without *ITS data field-specific term weighting*. Although all results exceed this baseline, the positive impact of the ITS-specific efforts is only significant for c:geo and Radiant datasets ($F_{1,2}$ increase between 10 and 12%) and it has only a small impact on the lighttpd and Redmine datasets ($F_{1,2}$ increase between 5 and 8%). This correlates with the ITSs that the projects employ. One hypothesis is that data cleanup and weighting have a higher influence on the Github based projects, since the NL data looks a bit *untidy* in comparison to the Redmine based projects. With an improvement of 11% for F_2 the best values were achieved for c:geo and Radiant. A reason for this might be that both ITSs contain the least technical discussions and terms. On the contrary, the next best results are measured for lighttpd and the project's ITS contains most technical data. All in all, combinations of weighting and ITS-specific preprocessing were responsible for the best obtainable results. As discussed in RQ 4.3 not considering the code and emphasizing the title works best for each project.

In addition, the values of the fully preprocessed datasets from Table 16.4 were compared to the same baseline as in Table 16.6 (only standard preprocessing). This comparison reveals that the preprocessed dataset performs better for different trace and issue types as well. Improvements can be noticed in every case. Most significantly, improvements in both, F_1 and F_2, of over 36% are achieved for $trace_t : I_{bug} \times I_{bug}$ in c:geo and over 10% for $trace_t : I_{feature} \times I_{bug}$

3 The annotation of the following trace types was defined in the gold standard creation tool: I_1 precedes, is parent of, blocks, clones I_2.

		$trace_t : I_{feature} \times I_{feature}$		
		RESULTS	ALG.	WEIGHTS
c:geo	F_1	0.4	BM25	2,1,0,0
	F_2	0.53	VSM	4,2,1,0
	R (P)	1 (0.6)	BM25	1,1,0,0
Lighttpd	F_1	**0.67**	VSM	1,1,0,0
	F_2	0.56	VSM	1,1,0,0
	R (P)	1 (0.02)	BM25	1,1,0,0
Redmine	F_1	**0.49**	VSM	2,1,0,0
	F_2	**0.55**	VSM	2,1,0,0
	R (P)	1 (0.07)	BM25	1,1,0,0

		$trace_t : I_{feature} \times I_{bug}$		
		RESULTS	ALG.	WEIGHTS
c:geo	F_1	0.46	VSM	8,4,2,1
	F_2	0.41	VSM	8,4,2,1
	R (P)	1 (0.03)	BM25	1,1,0,0
Lighttpd	F_1	**0.67**	VSM	1,1,1,0
	F_2	**0.71**	VSM	1,1,1,0
	R (P)	1 (0.8)	BM25	1,1,0,0
Redmine	F_1	0.29	VSM	4,2,1,0
	F_2	0.30	VSM	1,1,0,0
	R (P)	1 (0.03)	BM25	1,1,1,0

		$trace_t : I_{bug} \times I_{bug}$		
		RESULTS	ALG.	WEIGHTS
c:geo	F_1	**0.64**	VSM	1,1,1,0
	F_2	**0.67**	VSM	1,1,1,0
	R (P)	1 (0.04)	VSM	1,1,0,0
Lighttpd	F_1	0.33	LSA	8,4,2,1
	F_2	0.43	VSM	8,4,2,1
	R (P)	1 (0.01)	BM25	4,2,1,0
Redmine	F_1	0.29	VSM	4,2,1,0
	F_2	0.38	VSM	4,2,1,0
	R (P)	0.04 (0.01)	VSM	1,1,1,0

Table 16.4: Best Results for Different Issue Types.

		$trace_t : I \times I$			$trace_t^{duplicate} : I \times I$		
		RESULTS	ALG.	WEIGHTS	RESULTS	ALG.	WEIGHTS
C:GEO	F_1	0.58	VSM	2,1,0,0	0.67	LSA	1,1,0,0
	F_2	0.55	VSM	2,1,0,0	0.56	LSA	1,1,0,0
	R (P)	0.1 (0.03)	BM25+	1,1,1,1	1 (0.11)	BM25	1,1,0,0
REDM. LIGHTT.	F_1	0.4	VSM	4,2,1,0	0.18	LSA	1,1,0,0
	F_2	0.46	VSM	4,2,1,0	0.36	VSM	2,1,0,0
	R (P)	0.97 (0.04)	BM25	1,1,1,1	0.97 (0.3)	BM25	1,1,0,0
	F_1	0.31	VSM	1,1,0,0	0.31	LSA	1,2,1,1
	F_2	0.38	VSM	2,1,0,0	0.36	LSA	1,2,1,1
	R (P)	0.99 (0.03)	VSM	1,1,1,1	1 (0.01)	LSA	1,1,0,0

Table 16.5: Best Results for Different Trace Types.

		BEST RESULTS $trace_t : I \times I$					BASELINE	
		RESULTS	ALG.	WEIGHTS	STD. PRE.	ITS PRE.	STD. PRE. ONLY NO WEIGHTING	
C:GEO	F_1	0.58	VSM	2,1,0,0	true	true	0.46	LSA
	F_2	0.55	VSM	2,1,0,0	true	true	0.44	LSA
	R (P)	1 (0.03)	BM25+	1,1,1,1	false	true	0.99 (0.03)	BM25+
REDM. RADIANT LIGHTT.	F_1	0.4	VSM	4,2,1,0	true	true	0.32	VSM
	F_2	0.46	VSM	4,2,1,0	true	true	0.41	VSM
	R (P)	0.97 (0.04)	BM25	1,1,1,1	false	false	0.94 (0.03)	VSM
	F_1	0.27	VSM	2,1,0,0	true	true	0.17	LSA
	F_2	0.35	VSM	2,1,0,0	true	true	0.24	VSM
	R (P)	1 (0.02)	BM25	2,1,0,0	false	false	1 (0.02)	BM25
	F_1	0.31	VSM	2,1,0,0	true	true	0.25	VSM
	F_2	0.38	VSM	2,1,0,0	true	true	0.33	VSM
	R (P)	0.99 (0.3)	VSM	1,1,1,1	stopword only	false	0.99 (0.03)	VSM

Table 16.6: Best Results per Project (Trace and Issue Type not Distinguished).

in c:geo. On average, F_1 increased by 19.5% and F_2 by 13.33% for all projects and trace types.

Since no BM25 variants performed best, the improvements in comparison to the baseline from Figure 16.3 were computed. BM25 still performs worse than VSM and LSI. However, the F_2 scores for BM25[+,L] improved by 23% for c:geo, 3% for lighttpd, 3% for Radiant, and 6% for Redmine.

16.3 DISCUSSION

This section first discusses the implication of the above results for ITSoFD in practice. Then it discusses the main implications of the study for future research on trace retrieval in ITSs:

PRACTICAL IMPLICATIONS Overall, the results show that there is neither the best algorithm, nor the best preprocessing for all projects. However, removing code snippets and stack traces can be considered a good advice (cf. the term weights for $n-n-n-0$, with $n \in \{1,2,4\}$ in Table 16.6). It generally improves the results, especially precision, and has a negative impact of $< 4\%$ on the F_2 measure for lighttpd in the experiments, only. Also, up-weighting title and down-weighting comments has an overall positive impact. In general however, trace retrieval in ITSs shows mixed results in this study and seems to depend largely on the specific project. Although trace retrieval can be used in ITSoFD to get a more complete view of issues that are related to an SFR, the results should be validated manually.

In practice one should not rely only on IR based trace retrieval results. Issues that are manually linked by the developers should be included, too. However, the comparison of the GSTM and the DTM shows that even explicitly created traces by the developers can sometimes be wrong depending on the project and how links are used.

RETRIEVING TRACES VERSUS RECOMMENDING TRACES In this study traces were retrieved retrospectively. However, the same technique can be applied to suggest traces, e.g. during issue creation. If ranked list of similar issues is shown during issue creation, the user can choose the ones that apply and thus improve the trace creation. Cleland-Huang et al., 2012, for example, use IR techniques to implement trace recommendation for RE artifacts.

TRACE RETRIEVAL ALGORITHMS The best measures in Table 16.6 are computed with the 'simplest' algorithm: VSM. Since VSM considers every term of the text that was not removed by preprocessing, it can be hypothesized that is an important property for trace retrieval in ITS data. All variants of the BM25 algorithm fell short. Thus further

experiments with BM25 cannot expected to be promising in ITS trace retrieval without further modification of the algorithms.

16.4 THREATS TO VALIDITY

INTERNAL VALIDITY Every GSTM was created by one person only due to limited resources and it is known that the manual definition of trace types is a rather complex task for human experts (Cuddeback, Dekhtyar, and Hayes, 2010). This thread was minimized in this particular study by (1) creating and discussing guidelines how the gold standard should be made and discussing examples of related issues, (2) peer reviewing the created gold standards by random samples, and (3) peer reviewing decisions in the GSTM that would remove links present in the DTM. Still, some traces were hard to decide on. In case of doubt, no trace was inserted in the GSTM.

Another aspect with respect to internal validity is the extension of OpenTrace. OpenTrace creates queries in Apache Lucene to calculate *similarity* : $I \times I$. This involves data transformations from and to the GATE and OpenTrace frameworks. To minimize this thread, the code was inspected and enhanced very carefully to mitigate potential implementation problems and the source code and all data is published along with this thesis for others to inspect or improve.

EXTERNAL VALIDITY Even though a rather large GSTM of 100×100 traces was created, the GSTM comprises only a small part of the issues per project. Therefore, a generalization from these results cannot be made. However, about a third of the issues of the Radiant project is included in the GSTM, which is a rather large sample. Overall, the study still gives an indication of the importance of preprocessing and term weighting and shows that ITS data cannot be handled in the same way as structured RAs.

In addition to the facts discussed in Section 16.2.3, and due to the low number of duplicates in the datasets (see Table 16.1) the low results for duplicates might have occurred by chance. It is important to note that the definitions of *related* and *duplicate* issues have a major influence on the results. Different definitions would certainly lead to different results since trace matrices are always use-case-dependent.

16.5 CONCLUSION

This chapter presented an evaluation of five IR algorithms for the problem of automated trace retrieval on ITS data. Since the NL in ITSs is not comparable to structured RAs, the results show that IR algorithms that perform quite well with RAs, perform significantly weaker with ITS data. A combination of ITS-specific preprocessing and ITS data field-specific term weighting can positively influence the results

in many cases. However, results seem to vary due to the entirely different usage of NL in the different projects. Some issues use different expressions or abstraction levels although they are actually related. This in turn misleads IR algorithms. Overall, this study shows that trace retrieval in ITSs is still an open issue.

Part VI

CONCLUSION

This part summarizes the results of the thesis and discusses and concludes the Issue Tracking Software Feature Detection Method and its overall applicability. On this basis it lays out future work in the area.

17 DISCUSSION

"Design and programming are human activities; forget that and all is lost."
- Bjarne Stroustrup

In Chapter 11 most information in issues can be annotated and categorized by human experts. A taxonomy of issue and information types is created and others showed that task-specific categories and taxonomies can be identified, too (Bertram et al., 2010; Ko and Chilana, 2011). For computers, however, the use of NL in issues remains a problem[1]. This problem seems to be inherent in the issues: (1) they contain noise, as stakeholders tend to talk about technical solutions and mix NL with technical data. Although this problem can be mitigated, as discussed in Chapter 14, spelling and orthography remain a problem: (2) NL is used in a rather 'sloppy' way in issues. This does not come as a surprise since issues aid the development and are not seen as documentation or specification artifacts. In contrast people use glossaries, specific keywords, or templates to properly formulate SFs and related information in traditional RAs.

One could argue that it is needless to discuss problems related to NL in issues: ITSs offer the possibility to tag or categorize issues and to trace related information. However, Part iv of the thesis finds that these mechanisms are rarely used in practice and categories as well as links are often wrong. Furthermore, all ITSs offer categorization on the level of issues only, but sometimes it is a comment containing the relevant information that should be tagged or categorized. Due to all these inconveniences automated methods have to consider the NL. The relevance of NL is shown in Chapter 15: the maximum recall of SFR detection does not improve at all, when issue tags or categories are added to the MLF set[2].

Validating the detection of SFs in ITSs is a challenging task, too. In both, Part iv and Part v of the thesis, ITS NL data is manually annotated. Although human experts agree on clearly identifiable SF related information and traceability information, even humans have problems agreeing on the question: 'what is an SFs?' in many cases. The difference between the agreed upon cases and the uncertain cases in the gold standards in Chapter 15 is an example for this agreement problem. While the gold standard was prepared, the human analysts usually agreed upon clearly formulated SFRs that appear in the issue title or description. However, subtile SFRs that appear in issue comments were often found by only one of two analysts.

1 Although this thesis is about SF detection, other automated tasks face the same problems. Whether issues should be categorized, re-structured or summarized, the NL in issues depends very much on the people using the ITS.

2 To be precise, no improvement is made for the $\mathrm{MAX}(R), P_{\geqslant n}$ measure. Note, that tags and categories help improving precision if a high F1 score is the detection goal.

That said, there is no silver bullet for automatic processing of ITS data. However, the good news is that ML can support such a task: the ML based SFR detection approach from ITSoFD performs with high recall, especially on the uncertain dataset. Thus, most of the SFRs can be found automatically even if false positives need to be removed manually.

In theory, this performance increases in a practical environment. Consider that four coders annotated the data for SFR detection evaluation Chapter 15. In practice, however, it is likely that only a single annotator annotates the training data. Due to the agreement problem discussed above, it can be assumed that the data annotation is more consistent if a single annotator works through the issues[3]. Usually, classifiers create better models of more consistently annotated sentences and finally this should yield better results.

But detecting SFRs is only one part of this thesis. The ITS analysis in Part IV showed that SFRs detection is not enough to find all the information related to an SF. E.g. one could be interested in the overall amount of time, work, and contributors that participated in the realization of a particular SFR. To calculate this, all issues that relate to the SF need to be considered and this can be achieved easily if all related issues are traced. The trace recovery study in Chapter 16 shows that tracing related issues is more difficult than SFR detection itself. Although related work reports good results on trace recovery in RAs and finding duplicated issues, trace recovery in ITSs is a task far from being completed.

ITSoFD is a proof of concept that shows what contemporary automated methods can achieve on ITS data in terms of SFR detection and trace retrieval. Although this thesis did not investigate in techniques like deep learning for these tasks, it is arguable whether automated methods can ever be good enough to handle these tasks: the stakeholders using the ITS need to understand that rather small efforts in terms of consistent formulation or the consequent usage of ITS features eases both tasks. Solution ideas that support an understanding or support a more consequent usage of NL are discussed in Chapter 19.

3 At the same time the resulting model is presumably less transferable.

18 SUMMARY

"The most astonishing result [introducting issue based systems] is the eager acceptance by the users, although the implementation induced major changes in organization and working styles."
- Kunz and Rittel, 1970

Overall, the thesis contributes to the body of knowledge in SE and RE with respect to the analysis of and methods for ITS data. In Part iv two empirical studies investigate the content of ITSs and the NL of issues. Both studies focus on SFs, how these SFs are related, and how they are composed. In Part v three approaches are presented. The first prepares the data for more complex tasks by separating NL from technical artifacts, the second extracts SF related information from issues, and the third recovers traces within ITSs. These three approaches are the main components of the ITSoFD method.

ITSoFD tackles two major parts of SF detection in ITSs. To do so, the study in Chapter 10 researches ITSs in comparison to UD and shows that ITSs are a fruitful source for SF related information. Then, meaningful information types are derived from ITS data and a taxonomy of issue and information types is created and discussed in Chapter 11. A subset of these information types describe SFs. The thesis researches the SFR, clarifications and solution proposals in detail. The main components of ITSoFD are:

1. An approach based on ML to detect the SF relevant subset of information types. It is evaluated in an empirical study in Chapter 15. It can be shown that this approach reduces the amount necessary work with respect to a manual inspection of issues to detect the information.

2. An approach based on IR to recover trace links in ITSs. It is evaluated in an empirical study in Chapter 16. Although trace link recovery does not work as good in ITSs compared to related work on structured RAs, it can be shown that term weighting and ITS specific preprocessing slightly improves results.

Overall, the five empirical studies in this thesis found that ITS data is and will likely be hard to process automatically in the near future. The thesis represents a first step in terms of SF detection and it elaborates many empirical insights on ITS data for others to build upon.

19 FUTURE WORK

"Knowledge is of no value unless you put it into practice."
- Anton Chekhov

During the investigations in ITS data and the evaluations of automatic approaches, insights beyond the scope of this thesis could be gained. Of course these insights trigger new solution ideas in the researchers mind. In the following a selection of those ideas is presented. At this point I would like to thank the scientific community and the reviewers of the publications that went along with this thesis. Many ideas were triggered by feedback at scientific conferences and statements in reviews.

From a technical point of view, research on SFR detection can tackle multiple challenges: MLF sets can be curtailed further to determine which exact MLF has a high impact on SFR detection. A popular technique in the ML community is variable and feature selection (Guyon and Elisseeff, 2003). Very recently, a method to explain classifier decisions was developed by Ribeiro, Singh, and Guestrin, 2016. In Chapter 15 I hypothesize about the reasons for false positives in the uncertain cases dataset and false negatives in the certain cases dataset. However, methods as the one by Guyon and Elisseeff, 2003 or Ribeiro, Singh, and Guestrin, 2016 can be applied to validate or falsify such hypothesis with higher certainty. Another way to go is to improve classification on a sentence level. At the time when the experiments for Chapter 15 were conducted, others used deep learning for sentence classification (see, e.g. Kim, 2014). Such an technique could be applied to the SFR detection problems of this thesis, too.

Section 13.3 already discussed the possibility of applying clustering in the context of trace retrieval. Similar efforts exist to cluster unstructured NL documents into topics (Hindle et al., 2012). Such methods can be incorporated into ITSoFD to cluster the detected SFRs or to check how closely clarifications or solution ideas are related to a particular SFR.

Similar to clustering solutions, the approach by Medem, Akodjenou, and Teixeira, 2009 creates a hierarchy from trouble tickets using a binary tree. Although a binary tree is a rather limited data structure to create a hierarchy, a tree or graph-like structure could largely improve the output of ITSoFD. If detected SFRs are already structured, the information could easily be transformed into feature trees or similar representations that can be used in industrial applications such as release planning. Finally, ITS such tree structures could be used to improve ITS traceability.

As the results of this thesis clearly suggest that IR approaches do not perform very well on the data of some OSS projects, future work

in ITS traceability could focus on trace recommendation. Whenever an issue or even a comment is authored or modified, a ranked list of candidate issues to trace to could be offered to the user. This way algorithm results can be combined with human knowledge and the combination of trace suggestions offered by a machine together with the experience of a human analyst could improve trace generation significantly.

Another solution is to improve the ITSs themselves. Although many ITS improvements were proposed (e.g. Just, Premraj, and Zimmermann, 2008; Lotufo, Passos, and Czarnecki, 2012; Zimmermann et al., 2009), those ideas usually focus on the bug tracking aspect of ITSs and not the management of SFs. One particular aspect that needs improvement in ITSs is that comments cannot be categorized by any means. For software code, refactoring is a task that is done on a daily basis in most development companies (Fowler et al., 2012), but there are no means to 'refactor' an issue. If e.g. a user suggests a new SF in a comment to a bug, a refactoring of this comment could create a new SFR. From a user perspective such an refactoring is only a single click or keyboard shortcut away. As a start the content of the comment could be copied in a new issue and a trace could automatically be generated. Furthermore, the categorization or tagging on the level of ITS data fields could improve tidiness and such metadata would additionally support searching in ITSs.

In addition an LE based approach that comprises an extensive catalog of NL patterns could support the classification of comments in real time and suggest categories to the users. Although the patterns that were found in this thesis are not enough for SFR detection in terms of a high precision, such patterns could be used to suggest potential categories to ITS users. Thinking this even further a ML approach could then be applied that classifies good and bad patterns (e.g. suggestions to the user that result in a categorization and suggestions to the user that are ignored) so that potentially 'bad' patterns can be refactored or removed in the long run.

Overall, the results of this thesis are promising to pursue the path of integrating ML and IR techniques in ITSs directly. This way, not only precision, recall, or F-Scores can be measured, but ITS users can directly rate the impact of these methods on their daily work. This in turn gives valuable insights for the researcher.

Part VII

APPENDICES

A

CODING GUIDELINES USED FOR C:GEO, LIGHTTPD, RADIANT AND REDMINE ITS NL DATA

The following sections show the coder distribution, agreement factors and the coding guidelines that were used to create the data analyzed in Chapter 11 and the gold standard for the study presented in Chapter 15.

A.1 CODER DISTRIBUTION AND AGREEMENT FACTORS

Table A.1 shows which issue IDs were assigned to which coder. Table A.2 shows the complete agreement factors for the final coding. The coding guidelines are stated in the next section.

	C_1	C_2	C_3	C_4
c:geo *	25 – 50	25 – 50	1 – 24	1 – 24
lighttpd	25 – 50	1 – 24	1 – 24	25 – 50
radiant †	25 – 50	25 – 50	1 – 24	1 – 24
redmine	25 – 50	1 – 24	25 – 50	1 – 24

* Most discussed project, coded first.

† To validate the discussions, same coder pairing as in *.

Table A.1: Coder Distribution per Project.

A.2 CODING GUIDELINES

The following coding guidelines were extracted from a wiki employed during the experiments[1]. This wiki was editable by all coders so that discussions and decisions could be documented promptly.

HOW TO CODE

- Code only feature related information types. Each information type is described in detail below.
- Use GATE for coding together with the `only-feature-related.xml` annotation schema. This schema reflects the information types described in this guide.

1 The data was extracted without modification. However, the formatting was adapted to suit this thesis.

	F_1 SCORE			
CODER PAIR	ISSUE	TITLE	DESCIPTION	COMMENTS
C_1-C_2	0.9167	0.9167	0.9167	0.75
C_1-C_3	0.8919	0.8378	0.8919	0.5135
C_1-C_4	0.8780	0.8293	0.8293	0.7317
C_2-C_3	0.8000	0.8000	0.8000	0.8000
C_2-C_4	0.9091	0.8831	0.9091	0.4156
C_3-C_4	0.8571	0.7714	0.8286	0.7143
Average	0.875			

	COHENS KAPPA			
CODER PAIR	ISSUE	TITLE	DESCIPTION	COMMENTS
C_1-C_2	1	0.9773	0.8409	n/a
C_1-C_3	1	0.8485	1	n/a
C_1-C_4	1	0.8889	0.8333	n/a
C_2-C_3	1	1	1	n/a
C_2-C_4	1	0.9714	0.9143	n/a
C_3-C_4	1	0.7667	0.6667	n/a
Average	1	0.9088	0.876	n/a

Table A.2: Coder Agreement Factors.

- Sentence is the unit of analysis. Code whole sentences (an annotation may span multiple sentences, since this can be cleaned up automatically). If a sentence happens to include two information types, code both information types for this sentence.
- Do not code citations (e.g. lines marked with >)
- If multiple sentences are exactly the same, codes can be duplicated automatically.
- Feature related information should only be coded for the feature(s) that are topic of discussion (in most cases this is the feature where you find a request and/or a summary for). If there is a request for another feature in a later comment both feature related information should be coded. However, we should not code other feature related information which is not topic of the issue (e.g. already implemented functions in Redmine).
- Rationals are not Clarifications (or Requests)! E.g. "I think this is a great feature/Our PMs want this, too."
- If you do not understand a sentence, data field, or issue, do not code it.

GENERAL TIPS

- Modified from here: `http://cado.informatik.uni-hamburg.de/coding-guide`
- Do not interpret too much. Stick to the guide while evaluating ~~knowledge~~ information types!
- Do not code for more that one hour at once!
- Only code as true if the ~~knowledge~~ information type is clear based on the description of the guide!
- Read the guide from time to time!
- Do not make too long breaks (e.g. ~~several~~ more than one week)!

INFORMATION TYPES All information types are described in the following form:

Name for Example Information Type:

optional An example and potential keywords (*potential*, since keywords are sometimes wrong. For example in c:geo large 28: *Please provide help* is not about a user problem but about a feature request. This can be noted only if the tagger skims through the description (or the whole issue), first.

optional An anti-example → what would be correct

Summary:

- A brief summary/overview of the feature. In general, this does not form a complete sentence but is a single a noun phrase (e.g. "Change maptype for Livemap in Livemap [c:geo_l_000]"). This is not true for every summary field of the ITS. Sometimes people do form a complete sentence which is more like a request (e.g. "compass should point to the direction initially [c:geo_l_24]").

Request Functionality:

- A request of (new) software functionality. Expresses a feature-related need, demand, desire. Typical keywords (which are not always the only clear indicators for a request) are "feature", "should/would (be)", "I want", "add[ing]", "wouldn't [it be cool]", "provide". Generally positive words are often used in a request, too. E.g. "nice", "good", ...

Request Quality:

- As request functionality but mentions a software quality (or non functional requirement (NFR)). This is a feature which typically can be found in the *-ilities* type of requirements (e.g. security, reliability, usability) or performance requirements (e.g. speed, timing, response times, ...).

Solution:

- Describes a (technical) solution to the feature. A solution to the feature divided in conceptional and implementation. Conception includes solution proposals, libraries that could be used, user interface descriptions, etc. Implementation includes description of class files, sentences paired with code snippets, etc.

- It is important to code only those solutions that are relevant for the feature itself not e.g. for some other refactoring problem which is also discussed.

- This can be further distinguished in "concept" and "implementation" which reflects the feature abstraction levels from Paech, Hübner, and Merten, 2014, whereas concept is anything with a higher abstraction level (e.g. "But perhaps we can make the scale for google a little shorter (it is drawn by c:geo) to avoid that and to align it better with the mapsforge built-in scale. [c:geo_l_001]") and code is anything related to the real coding (e.g. "we should extend the method XYZ" or code snippets).

Clarification: It is important to code only the sentences which clarify a feature directly not e.g. clarifications for a certain solution or implementation proposal. Clarifications can be further distinguished:

Question: Questions with respect to the feature

Answer: Only answers to the questions above

AS-IS: Describing how the software is not (as opposed to the request how the software should be in future)

Other-App: Describing the feature by referencing how the feature is implemented in another app (e.g. a competitor)

Explanation: Any other explanation that clarifies the feature which is not an Answer, not AS-IS and not Other-App

B

DISTRIBUTION OF ISSUE AND INFORMATION TYPES

INFORMATION TYPE	COUNT	DOCUMENTS	SENTENCES	WORDS
Refactoring-Related → Clarification/Explanation	1	1	1	30
Feature-Related → Implementation Status	26	10	26	406
Refactoring-Related → Reference → ITS Management	2	2	2	21
SE-Process	7	4	7	91
Feature-Related → Rationale/Plus One	11	3	11	107
Feature-Related → Overview	10	10	10	43
Feature-Related → Solution/Implementation	33	10	33	689
Unclear/Unknown	1	1	1	35
Feature-Related → Rationale/Argument	26	7	26	449
Refactoring-Related → Technical Information	3	1	3	22
Bug-Related → Request Fix	1	1	1	15
Bug-Related → Cause Diagnostics/Explanation	88	8	88	1334
Not-SWE → Unknown/Other/Unclear	3	1	3	48
User Problem	7	2	7	93
Refactoring-Related → Clarification/Question	9	4	9	132
Refactoring-Related → Implementation Status	5	3	5	70
Bug-Related → Scheduling	8	4	8	92
Bug-Related → Cause Diagnostics/Question	28	5	28	342
Bug-Related → Rationale/Plus One	4	3	4	37
Feature-Related → Technical Information	13	1	13	98
Refactoring-Related → Rationale/Plus One	1	1	1	5
Feature-Related → Already-Implemented → ITS Management	1	1	1	12
Feature-Related → Unknown/Other/Unclear → ITS Management	4	2	4	50
Feature-Related → Scheduling	16	9	16	275
Refactoring-Related → Request Refactoring	3	3	3	60
Not-SE → Social Interaction	26	13	26	296
Feature-Related → Clarification/Explanation	33	5	33	518
Bug-Related → Reference → Clarification/As Is	4	1	4	35
Bug-Related → Clarification/As Is/Feature	4	1	4	64
Feature-Related → Clarification/Other App	5	5	5	132

Continued on the next page.

Continuation from the previous page.

INFORMATION TYPE	COUNT	DOCUMENTS	SENTENCES	WORDS
Feature-Related → Clarification/Question	17	7	17	271
Bug-Related → Attribute Change → ITS Management	4	3	4	61
Refactoring-Related → Clarification/As Is/Feature	2	1	2	16
Bug-Related → Solution/Implementation	11	4	11	248
Bug-Related → Technical Information	14	2	14	126
Refactoring-Related → Solution/Implementation	8	3	8	100
SE-Process → Scheduling	1	1	1	34
Feature-Related → Unknown//Other	6	3	6	72
Refactoring-Related → Clarification/As Is	3	1	3	63
Refactoring-Related → Rationale/Argument	4	1	4	71
Bug-Related → Reference → ITS Management	9	4	9	102
Feature-Related → yes → Clarification/Question	2	1	2	8
Feature-Related → Cancellation → ITS Management	2	2	2	36
Feature-Related → Request Functionality	30	13	30	500
Feature-Related → Clarification/User Story	2	1	2	61
Bug-Related → Clarification/As Is	47	11	47	860
Refactoring-Related → Scheduling	3	3	3	33
Bug-Related → Cancellation → ITS Management	1	1	1	19
Bug-Related → Closing → ITS Management	5	3	5	58
SE-Process → Social Interaction	2	1	2	27
Bug-Related → Implementation Status	8	4	8	80
Feature-Related → Closing → ITS Management	2	1	2	16
Refactoring-Related → Overview	4	4	4	30
Feature-Related → Duplicate → ITS Management	1	1	1	5
Feature-Related → Reference → ITS Management	22	9	22	271
Bug-Related → Rationale-Argument	19	4	19	430
Bug-Related → Unknown/Other/Unclear	5	3	5	56
Bug-Related → Cause Diagnostics/Reproducibility	32	6	32	404
Feature-Related → Clarification/As Is	5	2	5	151
Bug-Related → Overview	8	8	8	62

Table B.1: Complete Information Type Count.

C

CLARIFICATION AND SOLUTION DETECTION

Table C.1 is a copy of Table 15.5 on page 144. It serves as a reference to the evaluated MLF-sets in Figure C.1 to Figure C.4.

MLF-SET	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
BOW	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
bi- & tri-grams	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓
SAO	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓	-	✓
data field	-	-	✓	✓	-	-	-	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓
issue w/o type	-	-	-	-	✓	✓	-	-	-	-	✓	✓	-	-	✓	✓	-	-
issue with type	-	-	-	-	-	-	✓	✓	-	-	-	-	✓	✓	-	-	✓	✓
keywords	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	✓	✓	✓	✓

Table C.1: Reference Machine Learning Feature Sets.

C.1 PLOTS FOR THE CLARIFICATION LABEL

Figure C.1 shows the F_1 Scores and Figure C.2 the $MAX(R), P_{\geqslant 0.2}$ and $MAX(R), P_{\geqslant 0.05}$ scores for *Clarification* detection. The figures are analogous to Figure 15.4 on page 145 and Figure 15.5 on page 146. The black lines in Figure C.2 represent the recall and the red lines the precision.

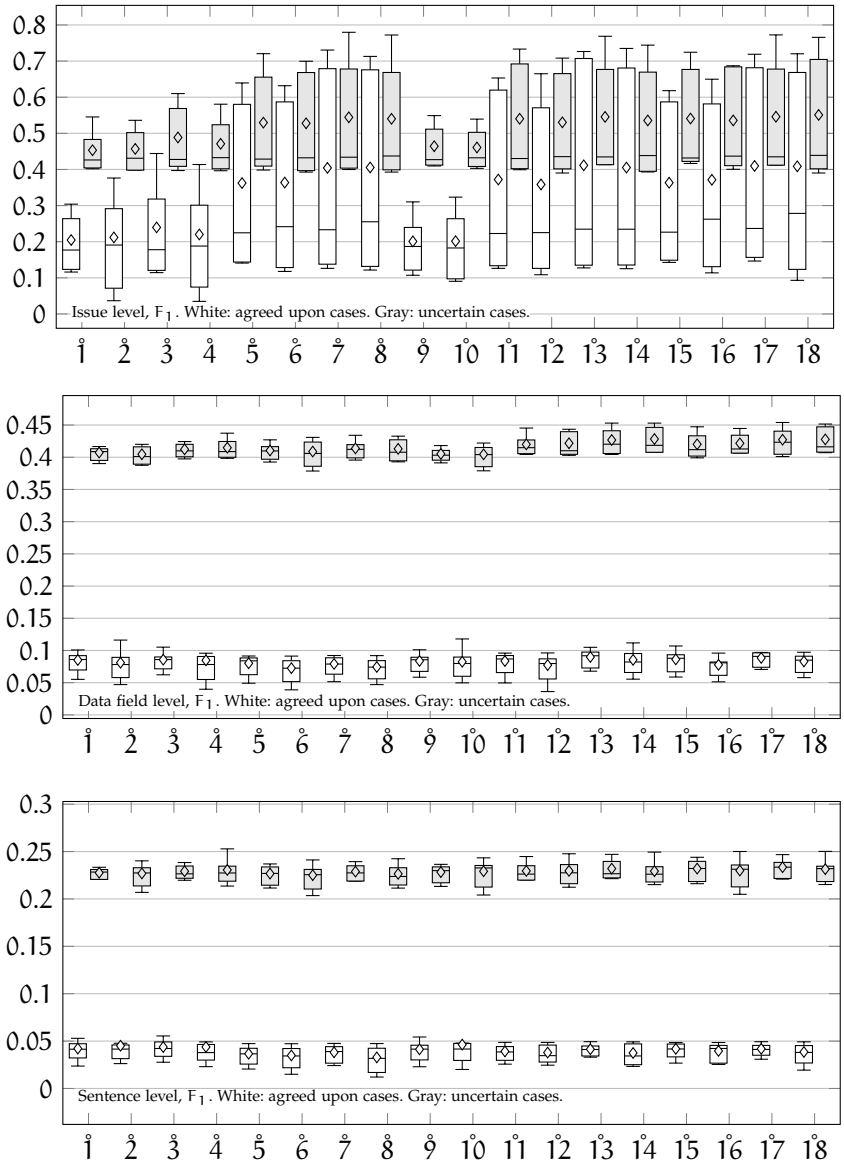


Figure C.1: F_1 Scores for *Clarification* Detection: Comparison of MLF-sets for Different Scopes with Multiple Classifiers.

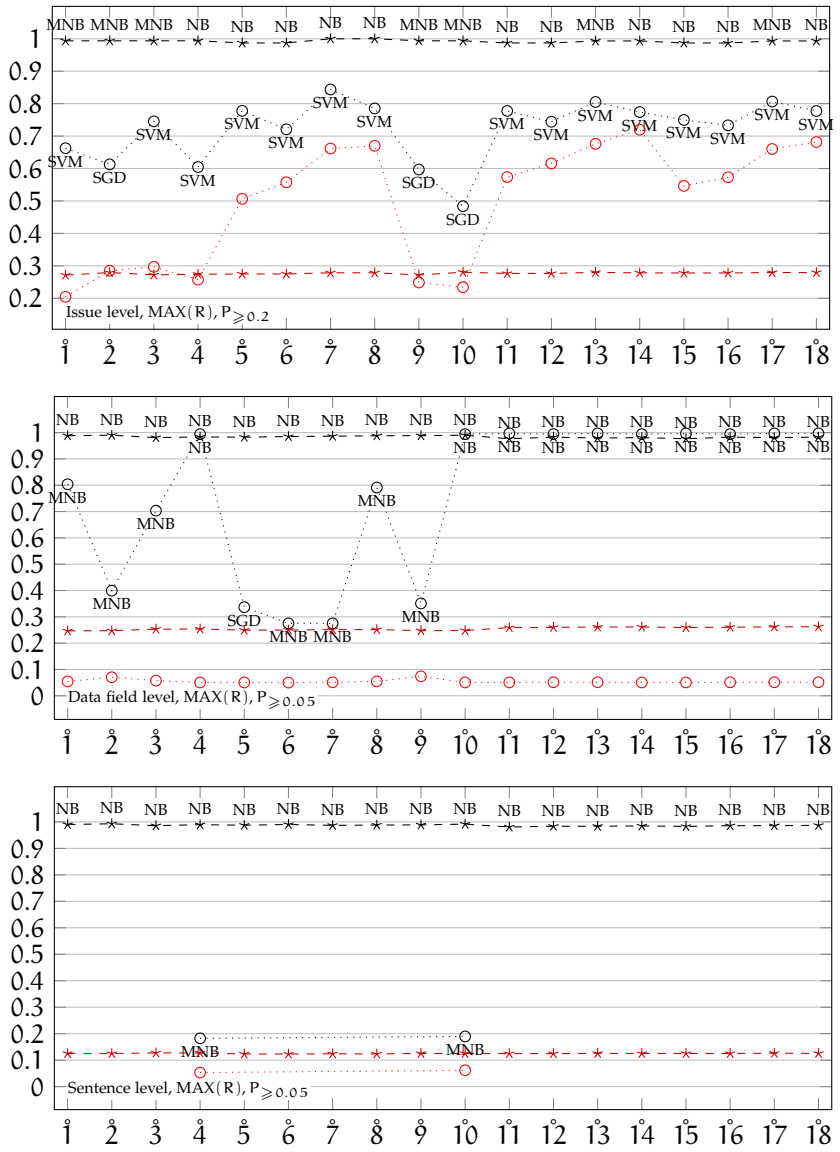


Figure C.2: MAX(R), $P_{\geqslant 0.2}$ and MAX(R), $P_{\geqslant 0.05}$ Scores for *Clarification* Detection: Comparison of MLF-sets for Different Scopes with Multiple Classifiers.

C.2 PLOTS FOR THE SOLUTION LABEL

Figure C.3 shows the F_1 Scores and Figure C.4 the $\mathrm{MAX}(R), P_{\geqslant 0.2}$ and $\mathrm{MAX}(R), P_{\geqslant 0.05}$ scores for *Solution* detection. The figures are analogous to Figure 15.4 on page 145 and Figure 15.5 on page 146. The black lines in Figure C.4 represent the recall and the red lines the precision.

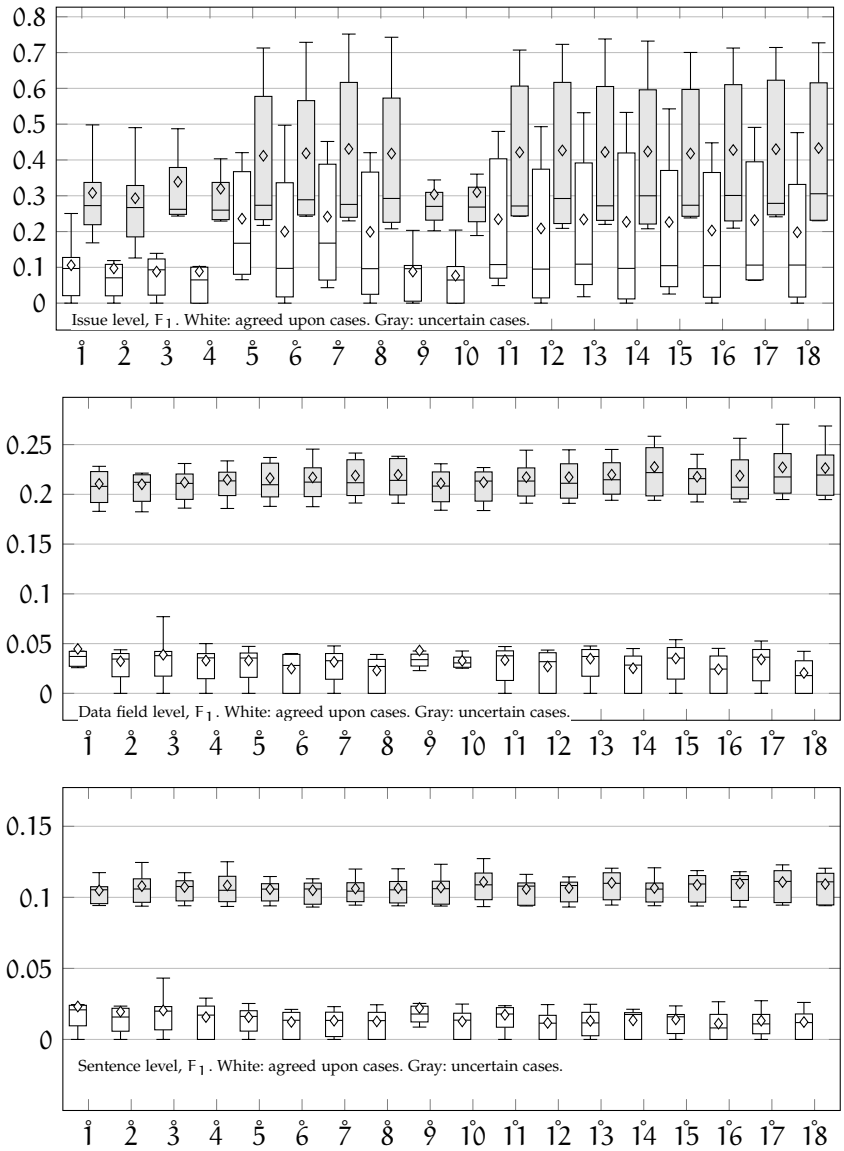


Figure C.3: F_1 Scores for *Solution* Detection: Comparison of MLF-sets for Different Scopes with Multiple Classifiers.

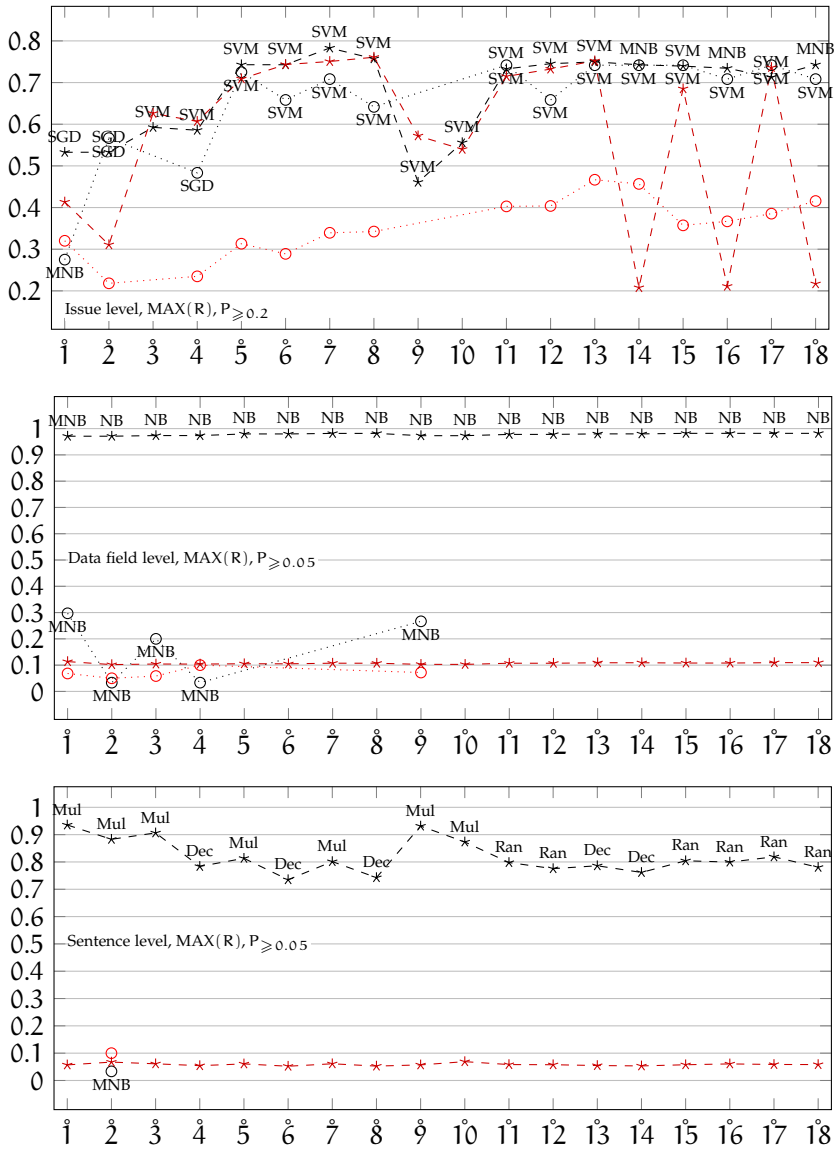


Figure C.4: MAX(R), $P_{\geqslant 0.2}$ and MAX(R), $P_{\geqslant 0.05}$ Scores for *Solution* Detection: Comparison of MLF-sets for Different Scopes with Multiple Classifiers.

C.3 CROSS TRAINING IMPACT FOR CLARIFICATION AND SOLUTION

Table C.2 and Table C.3 show the best achieved cross-training scores for *clarification* and *solution* detection, complementing *request functionliaty* from Table 15.6 on page 147.

project	level	MLF-set	algorithm	values	avg.	MLF-set	algorithm	values	avg.
		AGREED		F_1		UNCERTAIN		F_1	
cg	I	2	SGD	0.57	0.76	12	SGD	0.66	0.74
li	I	11	LR	0.83		7	SGD	0.76	
ra	I	15	SVM	0.71		7	LR	0.75	
re	I	17	SVM	0.91		13	SVM	0.79	
cg	DF	4	MNB	0.18	0.19	4	MNB	0.48	0.47
li	DF	4	MNB	0.27		18	SGD	0.48	
ra	DF	14	LR	0.17		12	SGD	0.43	
re	DF	13	LR	0.17		13	SGD	0.49	
cg	Se	4	MNB	0.09	0.19	11	MNB	0.29	0.26
li	Se	18	MNB	0.13		16	SGD	0.24	
ra	Se	13	LR	0.09		16	SGD	0.24	
re	Se	3	MNB	0.06		16	MNB	0.26	

$MAX(R), P_{\geqslant p}$		AGREED		p	R	P	UNCERTAIN		p	R	P
cg	I	1	Mul	0.2	0.6	0.23	3	NB	0.2	1.0	0.23
li	I	7	Lin	0.2	1.0	0.27	1	NB	0.2	1.0	0.23
ra	I	1	Lin	0.2	0.9	0.91	1	NB	0.2	1.0	0.21
re	I	1	Lin	0.2	0.8	0.08	1	NB	0.2	1.0	0.42
cg	DF	4	yes	0.05	0.9	0.06	4	NB	0.05	0.98	0.29
li	DF	4	yes	0.05	1.0	0.06	7	NB	0.05	1.0	0.17
ra	DF	16	SGD	0.05	0.5	0.06	7	NB	0.05	1.0	0.19
re	DF	3	Mul	0.05	0.9	0.05	1	NB	0.05	1.0	0.27
cg	Se	1	Mul	0.05	0.4	0.05	13	NB	0.05	1.0	0.14
li	Se	16	Mul	0.05	0.5	0.06	1	NB	0.05	1.0	0.09
ra	Se	12	Log	0.05	0.3	0.23	7	NB	0.05	1.0	0.09
re	Se	-	-	0.05	-	-	7	NB	0.05	1.0	0.10

with: cg=c:geo, li=Lighttpd, ra=Radiant, re=Redmine

Table C.2: *Clarification*: Best Cross-Training F_1 and $MAX(R), P_{\geqslant}$ Scores for all Levels of Detail and Projects.

project	level	MLF-set	algorithm	values		MLF-set	algorithm	values	
		AGREED		F_1	avg.	UNCERTAIN		F_1	avg.
cg	I	6	SGD	0.35	0.49	7	SGD	0.67	0.67
li	I	7	SGD	0.5		14	LR	0.67	
ra	I	7	SGD	0.5		15	SVM	0.6	
re	I	14	SGD	0.63		17	SGD	0.75	
cg	DF	7	RF	0.12	0.13	4	MNB	0.28	0.24
li	DF	16	RF	0.06		13	SGD	0.13	
ra	DF	3	MNB	0.18		16	SGD	0.19	
re	DF	4	MNB	0.17		2	MNB	0.34	
cg	Se	1	DT	0.06	0.09	3	MNB	0.12	0.11
li	Se	2	RF	0.03		9	SGD	0.06	
ra	Se	3	MNB	0.14		4	SGD	0.09	
re	Se	3	MNB	0.11		3	MNB	0.18	

$MAX(R), P_{\geqslant p}$		AGREED		p	R	P	UNCERTAIN		p	R	P
cg	I	5	SGD	0.2	0.31	0.27	9	SGD	0.2	0.77	0.4
li	I	7	SGD	0.2	0.5	0.5	7	SVM	0.2	0.92	0.24
ra	I	1	SGD	0.2	0.5	0.21	6	SGD	0.2	0.86	0.24
re	I	2	SGD	0.2	0.7	0.35	7	NB	0.2	1.0	0.31
cg	DF	5	DT	0.05	0.52	0.06	2	NB	0.05	0.95	0.13
li	DF	-	-	0.05	-	-	15	RF	0.05	0.88	0.05
ra	DF	1	MNB	0.05	0.66	0.08	13	NB	0.05	1.0	0.06
re	DF	14	SGD	0.05	0.57	0.05	7	NB	0.05	0.99	0.13
cg	Se	-	-	0.05	-	-	1	NB	0.05	0.96	0.05
li	Se	-	-	0.05	-	-	-	-	0.05	-	-
ra	Se	1	MNB	0.05	0.33	0.05	-	-	0.05	-	-
re	Se	1	MNB	0.05	0.20	0.06	13	NB	0.05	0.99	0.06

with: cg=c:geo, li=Lighttpd, ra=Radiant, re=Redmine

Table C.3: *Solution*: Best Cross-Training F_1 and $MAX(R), P_{\geqslant}$ Scores for all Levels of Detail and Projects.

D

REGULAR EXPRESSIONS TO DETECT TECHNICAL INFORMATION

All regular expressions are expressed in the Python programming language. In Python regular expressions can be extended by so called modifiers. Two modifiers are used in the following regular expressions:

- `re.DOTALL` means that the dot character matches the newline character, too. Effectively this setting spans the regular expression of multiple lines.
- `re.MULTILINE` means that the '^' character matches at the beginning of the string and at the beginning of the line and the '$' character matches at the end of the string and at the end of the line, respectively.

This regular expression detects marked-up technical information that spans multiple lines in the GitHub ITS:

```
leading_whitespace_pattern = \
  re.compile(r"^( {4,}|\t( |\t)*).*?$", re.MULTILINE)
multiline_backtick_pattern = \
  re.compile(r"```.*?```", re.DOTALL)
```

This regular expression detects inline marked-up technical information in the GitHub ITS:

```
single_line_backtick_pattern = re.compile(r"`.*?`")
```

This regular expression detects marked-up technical information that spans multiple lines in the Redmine ITS:

```
code_pattern = re.compile('<pre>.*?</pre>', re.DOTALL)
```

This regular expression detects inline marked-up technical information in the Redmine ITS:

```
at_pattern = re.compile(r"@.*?@")
```

BIBLIOGRAPHY

ANSI/IEEE (1998). *IEEE Std 829-1998 - Standard for Software Test Documentation*. Los Alamitos, California, USA (cit. on p. 14).

Ahsan, Syed Nadeem, Javed Ferzund, and Franz Wotawa (2009). "Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine." In: *4th International Conference on Software Engineering Advances (ICSEA 2009)*. Porto, Portugal: IEEE, pp. 216–221 (cit. on p. 121).

Alspaugh, Thomas a. and Walt Scacchi (2013). "Ongoing Software Development without Classical Requirements." In: *21st IEEE International Requirements Engineering Conference (RE 2013)*. Rio de Janeiro, Brazil: IEEE, pp. 165–174 (cit. on pp. 3, 81).

Amoui, Mehdi, Nilam Kaushik, Abraham Al-Dabbagh, Ladan Tahvildari, Shimin Li, and Weining Liu (2013). "Search-Based Duplicate Defect Detection: An Industrial Experience." In: *10th Working Conference on Mining Software Repositories (MSR 2013)*. San Francisco, California, USA: IEEE, pp. 173–182 (cit. on p. 123).

Angius, Elian and Rene Witte (2012). "OpenTrace: An Open Source Workbench for Automatic Software Traceability Link Recovery." In: *19th Working Conference on Reverse Engineering (WCRE 2012)*. Kingston, Ontario, Canada: IEEE, pp. 507–508 (cit. on p. 56).

Antoniol, Giuliano, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc (2008). "Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests." In: *18th Annual International Conference on Computer Science and Software Engineering (CASCON 2008)*. Richmond Hill, Ontario, Canada: ACM, p. 304 (cit. on p. 121).

Bacchelli, Alberto, Marco D'Ambros, and Michele Lanza (2010). "Extracting Source Code from E-Mails." In: *18th IEEE International Conference on Program Comprehension (ICPC 2010)*. Braga, Portugal: IEEE, pp. 24–33 (cit. on p. 120).

Bacchelli, Alberto, Marco D'Ambros, Michele Lanza, and Romain Robbes (2009). "Benchmarking Lightweight Techniques to Link E-Mails and Source Code." In: *16th Working Conference on Reverse Engineering (WCRE 2009)*. Lille, France: IEEE, pp. 205–214 (cit. on p. 120).

Bacchelli, Alberto, Anthony Cleve, Michele Lanza, and Andrea Mocci (2011). "Extracting Structured Data from Natural Language Documents with Island Parsing." In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence, Kansas, USA: IEEE, pp. 476–479 (cit. on p. 120).

Bacchelli, Alberto, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza (2012). "Content Classification of Development Emails." In: *34th International Conference on Software Engineering (ICSE 2012)*. Zurich, Switzerland: IEEE, pp. 375–385 (cit. on p. 120).

Baeza-Yates, Ricardo and Berthier Ribeiro-Neto (2011). *Modern Information Retrieval: The Concepts and Technology behind Search*. 2nd ed. Boston, Massachusetts, USA: Addison-Wesley (cit. on pp. 29, 32).

Bakar, Noor Hasrina, Zarinah M. Kasirun, and Norsaremah Salleh (2015). "Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review." In: *Journal of Systems and Software* 106, pp. 132–149 (cit. on pp. 5, 120).

Baysal, Olga, Reid Holmes, and Michael W. Godfrey (2013). "Situational Awareness: Personalizing Issue Tracking Systems." In: *35th International Conference on Software Engineering (ICSE 2013)*. San Francisco, California, USA: IEEE, pp. 1185–1188 (cit. on p. 96).

Berry, Daniel M, Khuzaima Daudjee, J. Dong, I. Fainchtein, M. A. Nelson, Torsten Nelson, and L. Ou (2004). "User's Manual as a Requirements Specification: Case Studies." In: *Requirements Engineering Journal* 9.1, pp. 67–82 (cit. on pp. 61, 65, 79–81).

Berry, Daniel, Ricardo Gacitua, Pete Sawyer, and Sri Fatimah Tjong (2012). "The Case for Dumb Requirements Engineering Tools." In: *18th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2012)*. Vol. 7195 LNCS. Essen, Germany: Springer, pp. 211–217 (cit. on pp. 41, 46, 47).

Bertram, Dane (2009). "The Social Nature of Issue Tracking in Software Engineering." M.Sc. Thesis. University of Calgary (cit. on pp. 4, 49, 118).

Bertram, Dane, Amy Voida, Saul Greenberg, and Robert Walker (2010). "Communication, Collaboration, and Bugs: The Social Nature of Issue Tracking in Small, Collocated Teams." In: *ACM Conference on Computer Supported Cooperative Work (CSCW 2010)*. Savannah, Georgia, USA: ACM, pp. 291–300 (cit. on pp. 3, 5, 97, 171).

Bettenburg, Nicolas, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim (2008). "Extracting Structural Information from Bug Reports." In: *5th Working Conference on Mining Software Repositories (MSR 2008)*. New York, New York, USA: ACM Press, p. 27 (cit. on p. 119).

Bettenburg, Nicolas, Bram Adams, Ahmed E. Hassan, and Michel Smidt (2011). "A Lightweight Approach to Uncover Technical Artifacts in Unstructured Data." In: *19th IEEE International Conference on Program Comprehension (ICPC 2011)*. Kingston, Ontario, Canada: IEEE, pp. 185–188 (cit. on p. 119).

Bird, Steven, Ewan Klein, and Edward Loper (2009). *Natural Language Processing with Python*. 1st ed. Sebastopol, California, USA: O'Reilly (cit. on pp. 24, 26, 36, 56, 138).

Bishop, Christopher M. (2009). *Pattern Recognition and Machine Learning*. 8th. New York, New York, USA: Springer, p. 738 (cit. on pp. 14, 36).

Borg, Markus, Per Runeson, and Anders Ardö (2014). "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability." In: *Empirical Software Engineering* 19.6, pp. 1565–1616 (cit. on pp. 5, 27, 123).

Borg, Markus, Per Runeson, Jens Johansson, and Mika V. Mäntylä (2014). "A replicated study on duplicate detection: Using Apache Lucene to search among Android defects." In: *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. Torino, Italy: ACM, pp. 1–4 (cit. on pp. 5, 123).

Boutkova, Ekaterina and Frank Houdek (2011). "Semi-Automatic Identification of Features in Requirement Specifications." In: *19th IEEE International Requirements Engineering Conference (RE 2011)*. Trento, Italy: IEEE, pp. 313–318 (cit. on p. 121).

Brill, Eric (1992). "A Simple Rule-Based Part of Speech Tagger." In: *Proceedings of the third conference on Applied natural language processing -*. Morristown, NJ, USA: Association for Computational Linguistics, p. 152 (cit. on pp. 22, 25).

Casamayor, Agustin, Daniela Godoy, and Marcelo Campo (2012). "Mining textual requirements to assist architectural software design: a state of the art review." In: *Artificial Intelligence Review* 38.3, pp. 173–191 (cit. on p. 121).

Cerulo, Luigi, Michele Ceccarelli, Massimiliano Di Penta, and Gerardo Canfora (2013). "A Hidden Markov Model to Detect Coded Information Islands in Free Text." In: *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013)*. Eindhoven, Netherlands: IEEE, pp. 157–166 (cit. on p. 120).

Chawla, Indu and Sandeep K. Singh (2015). "An Automated Approach for Bug Categorization Using Fuzzy Logic." In: *8th India Software Engineering Conference (ISEC 2015)*. Bangalore, India: ACM, pp. 90–99 (cit. on p. 121).

Chen, Danqi and Christopher D Manning (2014). "A Fast and Accurate Dependency Parser using Neural Networks." In: *11th Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. i. Doha, Qatar, pp. 740–750 (cit. on pp. 25, 55, 57, 138).

Cleland-Huang, Jane, Adam Czauderna, and Jane Huffman Hayes (2013). "Using TraceLab to Design, Execute, and Baseline Empirical Requirements Engineering Experiments." In: *21st IEEE International Requirements Engineering Conference (RE 2013)*. Rio de Janeiro, Brazil: IEEE, pp. 338–339 (cit. on p. 56).

Cleland-Huang, Jane, Raffaella Settimi, Xuchang Zou, and P. Solc (2006). "The Detection and Classification of Non-Functional Requirements with Application to Early Aspects." In: *14th IEEE International Conference on Requirements Engineering (RE 2006)*. Minneapolis - St. Paul, Minnesota, USA: IEEE, pp. 39–48 (cit. on p. 121).

Cleland-Huang, Jane, Patrick Mader, Mehdi Mirakhorli, and Sorawit Amornborvornwong (2012). "Breaking the Big-Bang Practice of Traceability: Pushing Timely Trace Recommendations to Project Stakeholders." In: *20th IEEE International Requirements Engineering Conference (RE 2012)*. Chicago, Illinois, USA: IEEE, pp. 231–240 (cit. on p. 165).

Cockburn, Alistair (2001). *Writing Effective Use Cases*. 1st ed. Boston, Massachusetts, USA: Addison-Wesley, p. 304 (cit. on p. 65).

Cohen, Jacob (1960). "A Coefficient of Agreement for Nominal Scales." In: *Educational and Psychological Measurement* 20.1, pp. 37–46 (cit. on pp. 54, 136).

Cuddeback, David, Alex Dekhtyar, and Jane Hayes (2010). "Automated Requirements Traceability: The Study of Human Analysts." In: *18th IEEE International Requirements Engineering Conference (RE 2010)*. c. Sydney, Australia: IEEE, pp. 231–240 (cit. on pp. 54, 124, 166).

Cunningham, Hamish (1999). "A definition and short history of Language Engineering." In: *Natural Language Engineering* 5.1, pp. 1–16 (cit. on p. 21).

Cunningham, Hamish et al. (2016). *Text Processing with GATE (Version 8)*. Sheffield, England, United Kingdom: University of Sheffield Department of Computer Science, p. 588 (cit. on pp. 22, 24, 55).

Dang, Yingnong, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel (2012). "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity." In: *34th International Conference on Software Engineering (ICSE 2012)*. Zurich, Switzerland: IEEE, pp. 1084–1093 (cit. on pp. 30, 123).

De Marneffe, Marie-Catherine, Bill MacCartney, and Christopher D. Manning (2006). "Generating Typed Dependency Parses from Phrase Structure Parses." In: *5th International Conference on Language Resources and Evaluation (LREC 2006)*. Genoa, Italy: European Language Resources Association (ELRA) (cit. on p. 87).

Demeyer, Serge (2011). "Research methods in computer science." In: *27th IEEE International Conference on Software Maintenance (ICSM 2011)*. Williamsburg, Virginia, USA: IEEE, pp. 600–600 (cit. on p. 45).

Dit, Bogdan, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk (2013). "Feature location in source code: a taxonomy and survey." In: *Journal of Software: Evolution and Process* 25.1, pp. 53–95 (cit. on pp. 5, 79, 122).

Domingos, Pedro (2012). "A Few Useful Things to Know about Machine Learning." In: *Communications of the ACM* 55.10, pp. 78–86 (cit. on pp. 14, 34).

Duan, Chuan, Paula Laurent, Jane Cleland-Huang, and Charles Kwiatkowski (2009). "Towards automated requirements prioritization and triage." In: *Requirements Engineering* 14.2, pp. 73–89 (cit. on p. 121).

Easterbrook, Steve, Janice Singer, Margaret-Anne Storey, and Daniela Damian (2008). "Selecting Empirical Methods for Software Engineering Research." In: *Guide to Advanced Empirical Software Engineering*. London: Springer London, pp. 285–311 (cit. on p. 45).

Ernst, Neil A. and Gail C. Murphy (2012). "Case Studies in Just-In-Time Requirements Analysis." In: *2nd IEEE International Workshop on Empirical Requirements Engineering (EmpiRE 2012)*. IEEE, pp. 25–32 (cit. on pp. 3, 49).

Fantechi, Alessandro and Emilio Spinicci (2005). "A Content Analysis Technique for Inconsistency Detection in Software Requirements Documents." In: *8th Workshop em Engenharia de Requisitos (WER 2005)*. Porto, Portugal, pp. 245–256 (cit. on p. 138).

Feldman, Ronen and James Sanger (2006). *The Text Mining Handbook*. New York, New York, USA: Cambridge University Press (cit. on pp. 21, 22, 24, 26, 33, 125, 140).

Finkelstein, A and I Sommerville (1996). "The Viewpoints FAQ." In: *Software Engineering Journal* 11.1, pp. 2–4 (cit. on p. 96).

Fitzgerald, Camilo, Emmanuel Letier, and Anthony Finkelstein (2012). "Early failure prediction in feature request management systems: an extended study." In: *Requirements Engineering* 17.2, pp. 117–132 (cit. on pp. 49, 85, 138).

Flyvbjerg, Bent (2006). "Five Misunderstandings About Case-Study Research." In: *Qualitative Inquiry* 12.2, pp. 219–245 (cit. on p. 153).

Fowler, M, K Beck, J Brant, W Opdyke, and D Roberts (2012). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Reading, Massachusetts, USA: Addison-Wesley (cit. on p. 176).

Fricker, Samuel and Susanne Schumacher (2012). "Release Planning with Feature Trees: Industrial Case." In: *18th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2012)*. Vol. LNCS 7195. Essen, Germany: Springer, pp. 288–305 (cit. on p. 3).

Furnas, George W., Scott Deerwester, Susan T. Dumais, Thomas K. Landauer, Richard a. Harshman, Lynn a. Streeter, and Karen E. Lochbaum (1988). "Information retrieval using a singular value decomposition model of latent semantic structure." In: *11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1988)*. New York, New York, USA: ACM Press, pp. 465–480 (cit. on p. 32).

Gervasi, Vincenzo and Didar Zowghi (2011). "Mining Requirements Links." In: *17th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2011)*. Ed. by D. Berry and X. Franch. Vol. 6606 LNCS. Essen, Germany: Springer, pp. 196–201 (cit. on p. 124).

— (2014). "Supporting Traceability through Affinity Mining." In: *22nd IEEE International Requirements Engineering Conference (RE 2014)*. Karlskrona, Sweden: IEEE, pp. 143–152 (cit. on pp. 27, 124).

Ghazarian, Arbi (2012). "Characterization of Functional Software Requirements Space: The law of Requirements Taxonomic Growth." In: *20th IEEE International Requirements Engineering Conference (RE 2012)*. Chicago, Illinois, USA: IEEE, pp. 241–250 (cit. on p. 81).

Ghosh, Shalini, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner (2016). "ARSENAL: Automatic Requirements Specification Extraction from Natural Language." In: *8th International Symposium on NASA Formal Methods (NFM 2006)*. Berlin/Heidelberg, Germany: Springer, pp. 41–46 (cit. on p. 122).

Glinz, Martin (2012). *A Glossary of Requirements Engineering Terminology*. Tech. rep. Version 1.4, September 2012. Zurich, Switzerland: International Requirements Engineering Board (IREB) e.V. (cit. on p. 15).

Gorschek, Tony and Claes Wohlin (2006). "Requirements Abstraction Model." In: *Requirements Engineering Journal* 11.1, pp. 79–101 (cit. on pp. 14, 15, 65).

Gotel, Olli, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grunbacher, and Giuliano Antoniol (2012). "The Quest for Ubiquity: A Roadmap for Software and Systems Traceability Research." In: *20th IEEE International Requirements Engineering Conference (RE 2012)*. Chicago, Illinois, USA: IEEE, pp. 71–80 (cit. on pp. 5, 123).

Gotel, Orlena C Z and Anthony C W Finkelstein (1994). "An Analysis of the Requirements Traceability Problem." In: *1st International Conference on Requirements Engineering (RE 1994)*. Colorado Springs, Colorado, USA: IEEE, pp. 94–101 (cit. on p. 30).

Guo, Jin, Jane Cleland-Huang, and Brian Berenbach (2013). "Foundations for an Expert System in Domain-Specific Traceability." In: *21st IEEE International Requirements Engineering Conference (RE 2013)*. 978. Rio de Janeiro, Brazil: IEEE, pp. 42–51 (cit. on p. 124).

Guyon, Isabelle and André Elisseeff (2003). "An Introduction to Variable and Feature Selection." In: *Journal of Machine Learning Research* 3, pp. 1157–1182 (cit. on pp. 34, 149, 175).

Guzman, Emitza and Walid Maalej (2014). "How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews." In: *22nd IEEE International Requirements Engineering Conference (RE 2014)*. Karlskrona, Sweden: IEEE, pp. 153–162 (cit. on p. 137).

Han, Jiawei, Micheline Kamber, and Jian Pei (2012). *Data Mining: Concepts and Techniques*. 3rd. Amsterdam, The Netherlands: Morgan Kaufmann (cit. on pp. 14, 21, 23, 24, 33–36).

Hausser, Roland (1999). *Foundations of Computational Linguistics*. 3rd. Berlin/Heidelberg, Germany: Springer (cit. on pp. 14, 25).

Heck, Petra and Andy Zaidman (2014). "Horizontal Traceability for Just-In-Time Requirements: The Case for Open Source Feature Requests." In: *Journal of Software: Evolution and Process* 26.12, pp. 1280–1296 (cit. on pp. 123, 154, 159).

Hemetsberger, A. (2006). "Learning and Knowledge-building in Open-source Communities: A Social-experiential Approach." In: *Management Learning* 37.2, pp. 187–214 (cit. on p. 3).

Hemmati, Hadi, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey (2013). "The MSR Cookbook: Mining a Decade of Research." In: *10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*. San Francisco, California, USA: IEEE, pp. 343–352 (cit. on p. 121).

Herzig, Kim, Sascha Just, and Andreas Zeller (2013). "It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction." In: *35th International Conference on Software Engineering (ICSE 2013)*. IEEE, pp. 392–401 (cit. on pp. 3, 4, 85, 94, 97, 122, 123).

Hindle, Abram, Neil a. Ernst, Michael W. Godfrey, and John Mylopoulos (2012). "Automated topic naming." In: *Empirical Software Engineering* (cit. on p. 175).

Hohpe, Gregor and Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, Massachusetts, USA: Addison Wesley (cit. on p. 55).

Huffman Hayes, Jane, Alex Dekhtyar, and Senthil Karthikeyan Sundaram (2006). "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods." In: *IEEE Transactions on Software Engineering* 32.1, pp. 4–19 (cit. on p. 39).

Hull, David A. (1996). "Stemming Algorithms: A Case Study for Detailed Evaluation." In: *Journal of the American Society for Information Science* 47.1, pp. 70–84 (cit. on p. 25).

Just, Sascha, Rahul Premraj, and Thomas Zimmermann (2008). "Towards the Next Generation of Bug Tracking Systems." In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2008)*. Herrsching, Germany: IEEE, pp. 82–85 (cit. on pp. 96, 176).

Kagdi, Huzefa, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad (2012). "Assigning change requests to software developers." In: *Journal of Software: Evolution and Process* 24.1, pp. 3–33 (cit. on p. 121).

Kang, Kyo Chul, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson (1990). *Feasibility Study Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. November.

Pittsburgh, Pennsylvania, USA: Software Engineering Institute, Carnegie Mellon University (cit. on p. 3).

Kaushik, Nilam and Ladan Tahvildari (2012). "A Comparative Study of the Performance of IR Models on Duplicate Bug Detection." In: *16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*. Szeged, Hungary: IEEE, pp. 159–168 (cit. on pp. 5, 123).

Kim, Yoon (2014). "Convolutional Neural Networks for Sentence Classification." In: *11th Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. Doha, Qatar: Association for Computational Linguistics, pp. 1746–1751 (cit. on p. 175).

Kitchenham, B A, S L Pfleeger, L M Pickard, P W Jones, D C Hoaglin, K El Emam, and J Rosenberg (2002). "Preliminary guidelines for empirical research in software engineering." In: *IEEE Transactions on Software Engineering* 28.8, pp. 721–734 (cit. on p. 45).

Kitchenham, Barbara A., Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang (1999). "Towards an Ontology of Software Maintenance." In: *Journal of Software Maintenance: Research and Practice* 11.6, pp. 365–389 (cit. on p. 97).

Kitchenham, Barbara and S Charters (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. Durham, England, United Kingdom: EBSE Technical Report 2007-01, Keele University (cit. on p. 119).

Knauss, Eric and Daniel Ott (2014). "(Semi-) automatic Categorization of Natural Language Requirements." In: *20th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2014)*. Ed. by Camille Salinesi and Inge van de Weerd. Vol. 8396. C. Essen, Germany: Springer, pp. 39–54 (cit. on p. 121).

Ko, Andrew J. and Parmit K. Chilana (2011). "Design, discussion, and dissent in open bug reports." In: *6th iConference (iConference 2011)*. Seattle, Washington, USA: ACM Press, pp. 106–113 (cit. on pp. 4, 17, 85, 97, 171).

Krämer, Daniel (2014). "Automatische Erkennung von Traces zwischen textuellen Artefakten im Kontext von Issue-Tracking-Systemen [Automatic Detection of Traces Between Text Artifacts in the Context of Issue Tracking Systems]." MSc Thesis. Hochschule Bonn-Rhein-Sieg, p. 129 (cit. on p. 56).

Kruchten, Philippe (2003). *The Rational Unified Process: An Introduction*. Third. Amsterdam, Netherlands: Addison-Wesley, p. 310 (cit. on p. 30).

Kunz, Werner and Horst W. J. Rittel (1970). *Issues as Elements of Information Systems*. Tech. rep. 131. Berkeley, California, USA: Institute

of Urban and Regional Development, University of California (cit. on pp. 16, 96, 99, 103, 173).

Lamkanfi, Ahmed, Serge Demeyer, Quinten David Soetens, and Tim Verdonck (2011). "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug." In: *15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*. Oldenburg, Germany: IEEE, pp. 249–258 (cit. on p. 121).

Lotufo, Rafael and Krzysztof Czarnecki (2012). *Improving Bug Report Comprehension*. Tech. rep. Waterloo, Ontario, Canada: Generative Software Development Lab, University of Waterloo (cit. on p. 96).

Lotufo, Rafael, Leonardo Passos, and Krzysztof Czarnecki (2012). "Towards Improving Bug Tracking Systems with Game Mechanisms Rafael." In: *9th IEEE Working Conference on Mining Software Repositories (MSR 2012)*. Zurich, Switzerland: IEEE, pp. 2–11 (cit. on pp. 4, 96, 176).

Lucia, Andrea De, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora (2007). "Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods." In: *ACM Transactions on Software Engineering and Methodology* 16.4, pp. 13–63 (cit. on p. 123).

Lv, Yuanhua and ChengXiang Zhai (2011a). "Lower-Bounding Term Frequency Normalization." In: *20th ACM Conference on Information and Knowledge Management (CIKM 2011)*. Glasgow, Scotland, United Kingdom: ACM, pp. 7–16 (cit. on p. 32).

— (2011b). "When Documents Are Very Long, BM25 Fails!" In: *34th International ACM SIGIR Conference on Research and development in Information (SIGIR 2011)*. Beijing, China: ACM, pp. 1103–1104 (cit. on p. 32).

Maalej, Walid and Hadeer Nabil (2015). "Bug Report, Feature Request, or Simply Praise? On Automatically Classifying App Reviews." In: *23rd IEEE International Requirements Engineering Conference (RE 2015)*. IEEE, pp. 116–125 (cit. on pp. 115, 122, 138, 151).

Maalej, Walid and Martin P. Robillard (2013). "Patterns of Knowledge in API Reference Documentation." In: *IEEE Transactions on Software Engineering* 39.9, pp. 1264–1282 (cit. on p. 84).

MacDonell, Stephen, Martin Shepperd, Barbara Kitchenham, and Emilia Mendes (2010). "How Reliable Are Systematic Reviews in Empirical Software Engineering?" In: *IEEE Transactions on Software Engineering* 36.5, pp. 676–687 (cit. on p. 119).

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze (2008). *An Introduction to Information Retrieval*. 1st ed. Cambridge, England: Cambridge University Press (cit. on pp. 14, 21, 32, 41).

Manning, Christopher D. and Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing*. 1st ed. Cambridge, Massachusetts, USA: The MIT Press (cit. on pp. 14, 24, 25, 110, 112, 140, 150).

Marciniak, John J, ed. (2001). *Encyclopedia of Software Engineering*. New York, New York, USA: Wiley (cit. on p. 15).

Matter, Dominique, Adrian Kuhn, and Oscar Nierstrasz (2009). "Assigning bug reports using a vocabulary-based expertise model of developers." English. In: *6th IEEE International Working Conference on Mining Software Repositories (MSR 2009)*. Vancouver, British Columbia, Canada: IEEE, pp. 131–140 (cit. on p. 121).

Medem, Amelie, Marc-Ismael Akodjenou, and Renata Teixeira (2009). "TroubleMiner: Mining network trouble tickets." In: *2009 IFIP/IEEE International Symposium on Integrated Network Management-Workshops*. New York, New York, USA: IEEE, pp. 113–119 (cit. on p. 175).

Merten, Thorsten and Barbara Paech (2013). "Research Preview: Automatic Data Categorization in Issue Tracking Systems." In: *43. Jahrestagung der Gesellschaft für Informatik e.V. (INFORMATIK 2013)*. Koblenz, Germany: Lecture Notes in Informatics, pp. 128–130 (cit. on p. 7).

Merten, Thorsten, Bastian Mager, Simone Bürsner, and Barbara Paech (2014). "Classifying Unstructured Data into Natural Language Text and Technical Information." In: *11th Working Conference on Mining Software Repositories (MSR 2014)*. Hyderabad, India: ACM Press, pp. 300–303 (cit. on pp. 9, 125).

Merten, Thorsten, Bastian Mager, Paul Hübner, Thomas Quirchmayr, Simone Bürsner, and Barbara Paech (2015). "Requirements Communication in Issue Tracking Systems in Four Open-Source Projects." In: *6th Workshop on Requirements Prioritization and Communication (RePriCo 2015)*. Essen, Germany: CEUR Workshop Proceedings, pp. 114–125 (cit. on pp. 3, 9, 26, 83, 86).

Merten, Thorsten, Daniel Krämer, Bastian Mager, Paul Schell, Simone Bürsner, and Barbara Paech (2016a). "Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data?" In: *22nd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016)*. Vol. 9619. Gothenburg, Sweden: Springer, pp. 45–62 (cit. on pp. 3, 9, 155).

Merten, Thorsten, Matús Falis, Paul Hübner, Thomas Quirchmayr, Simone Bürsner, and Barbara Paech (2016b). "Software Feature Request Detection in Issue Tracking Systems." In: *24th IEEE International Requirements Engineering Conference (RE 2016)*. Beijing, China: IEEE, pp. 166–175 (cit. on p. 9).

Moonen, Leon (2001). "Generating Robust Parsers Using Island Grammars." In: *8th Working Conference on Reverse Engineering (WCRE 2001)*. Los Alamitos, California: IEEE, pp. 13–22 (cit. on p. 119).

Mukherjee, Debdoot and Malika Garg (2013). "Which Work-Item Updates Need Your Response?" In: *10th Working Conference on Min-*

ing Software Repositories (MSR 2013). San Francisco, California, USA: IEEE, pp. 12–21 (cit. on p. 121).

Nagwani, Naresh Kumar and Shrish Verma (2012). "Predicting Expert Developers for Newly Reported Bugs Using Frequent Terms Similarities of Bug Attributes." In: *9th International Conference on ICT and Knowledge Engineering (ICT & Knowledge Engineering 2011)*. Bangkok, Thailand: IEEE, pp. 113–117 (cit. on p. 121).

Natt och Dag, Johan and Vincenzo Gervasi (2005). "Managing Large Repositories of Natural Language Requirements." In: *Engineering and Managing Software Requirements*. Berlin/Heidelberg, Germany: Springer, pp. 219–244 (cit. on p. 18).

Neelofar, Muhammad Younus Javed, and Mosin Hufsa (2012). "An Automated Approach for Software Bug Classification." In: *6th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS 2012)*. Palermo, Italy: IEEE, pp. 414–419 (cit. on p. 121).

Neuendorf, Kimberly A. (2002). *The Content Analysis Guidebook*. 1st ed. Thousand Oaks, California, USA: SAGE Publications (cit. on pp. 53, 136, 151).

Ng, Andrew Y. and Michael I. Jordan (2001). "On Discriminative vs. Generative Classifiers: A comparison of Logistic Regression and Naive Bayes." In: *Advances in Neural Information Processing Systems 14 (NIPS 2001)*. Boston, Massachusetts, USA: MIT Press, pp. 841–848 (cit. on p. 143).

Nguyen Duc, Anh, Daniela S. Cruzes, Claudia Ayala, and Reidar Conradi (2011). "Impact of Stakeholder Type and Collaboration on Issue Resolution Time in OSS Projects." In: *7th International Conference on Open Source Systems: Grounding Research (OSS 2011)*. Vol. 365. Salvador, Brazil: Springer, pp. 1–16 (cit. on pp. 3, 124).

Nguyen, Anh Tuan, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen (2012). "Multi-layered Approach for Recovering Links between Bug Reports and Fixes." In: *20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*. Cary, North Carolina, USA: ACM, p. 1 (cit. on p. 124).

Niu, Nan and Anas Mahmoud (2012). "Enhancing Candidate Link Generation for Requirements Tracing: The Cluster Hypothesis Revisited." In: *20th IEEE International Requirements Engineering Conference (RE 2012)*. Chicago, Illinois, USA: IEEE, pp. 81–90 (cit. on p. 124).

Nivre, Joakim et al. (2015). *Universal Dependencies 1.2*. Tech. rep. Prague, Czech Republic: LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University (cit. on p. 26).

Noll, John and Wei-Ming Liu (2010). "Requirements Elicitation in Open Source Software Development: A Case Study." In: *3rd International Workshop on Emerging Trends in Free/Libre/Open Source*

Software Research and Development (FLOSS 2010). New York, New York, USA: ACM Press, pp. 35–40 (cit. on p. 81).

Oliveto, Rocco, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia (2010). "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery." In: *18th IEEE International Conference on Program Comprehension*. Braga, Portugal: IEEE, pp. 68–71 (cit. on p. 124).

Ott, Daniel (2013). "Automatic Requirement Categorization of Large Natural Language Specifications at Mercedes-Benz for Review Improvements." In: *19th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2013)*. Essen, Germany, pp. 50–64 (cit. on p. 121).

Paech, Barbara, Paul Hübner, and Thorsten Merten (2014). "What are the Features of this Software?" In: *9th International Conference on Software Engineering Advances (ICSEA 2014)*. Nice, France: IARIA XPS Press, pp. 97–106 (cit. on pp. 3, 7, 9, 61, 65, 182).

Parnas, David L. (2010). "Precise Documentation: The Key to Better Software." In: *The Future of Software Engineering (FOSE) Symposium*. Ed. by Sebastian Nanz. Zürich, Switzerland: Springer, pp. 125–148 (cit. on p. 61).

Parnas, David L. and Paul C. Clements (1986). "A Rational Design Process: How and Why to Fake It." In: *IEEE Transactions of Software Engineering* 12.2, pp. 251–257 (cit. on p. 45).

Pedregosa, Fabian et al. (2011). "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on pp. 35, 36, 56).

Pohl, Klaus (2010). *Requirements Engineering : Fundamentals, Principles, and Techniques*. 1st ed. Berlin, Germany: Springer (cit. on p. 30).

Porter, Martin .F. (1980). "An algorithm for suffix stripping." In: *Program: electronic library and information systems* 14.3, pp. 130–137 (cit. on p. 26).

Rahimi, Mona, Mehdi Mirakhorli, and Jane Cleland-Huang (2014). "Automated Extraction and Visualization of Quality Concerns from Requirements Specification." In: *22nd IEEE International Requirements Engineering Conference (RE 2014)*. Karlskrona, Sweden: IEEE, pp. 253–262 (cit. on p. 121).

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin (2016). ""Why Should I Trust You?" Explaining the Predictions of Any Classifier." In: *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016)*. San Francisco, California, USA: ACM, pp. 1135–1144 (cit. on p. 175).

Robertson, Stephen E, Steve Walker, Micheline Hancock-Beaulieu, Aarron Gull, and Marianna Lau (1992). "Okapi at {TREC}." In: *1st Text REtrieval Conference, (TREC 1992)*. Ed. by Donna K Harman. Vol. Special Pu. Rockville, Maryland, USA: National Institute of Standards and Technology {(NIST)}, pp. 21–30 (cit. on p. 32).

Robson, Colin and Kieran McCartan (2015). *Real World Research*. 4th. Chichester, United Kingdom: Wiley, p. 560 (cit. on p. 53).

Rubin, Julia and Marsha Chechik (2013). "A Survey of Feature Location Techniques." In: *Domain Engineering*. Ed. by Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin. Berlin/Heidelberg, Germany: Springer, pp. 29–58 (cit. on p. 122).

Runeson, Per, Magnus Alexandersson, and Oskar Nyholm (2007). "Detection of Duplicate Defect Reports Using Natural Language Processing." In: *29th International Conference on Software Engineering (ICSE 2007)*. Minneapolis, Minnesota, USA: IEEE, pp. 499–510 (cit. on pp. 123, 154).

Runeson, Per and Martin Höst (2009). "Guidelines for conducting and reporting case study research in software engineering." In: *Empirical Software Engineering* 14.2, pp. 131–164 (cit. on pp. 46, 47).

Ryan, Kevin (1993). "The Role of Natural Language in Requirements Engineering." In: *IEEE International Symposium on Requirements Engineering*. San Diego, California, USA: IEEE, pp. 240–242 (cit. on pp. 120, 133).

Salton, G., A. Wong, and C. S. Yang (1975). "A vector space model for automatic indexing." In: *Communications of the ACM* 18.11, pp. 613–620 (cit. on p. 31).

Santorini, Beatrice (1990). *Part of Speech Tagging Guidelines for the Penn Treebank Project*. Tech. rep. Pennsylvania: University of Pennsylvania (cit. on p. 25).

Scacchi, Walt (2009). "Understanding Requirements for Open Source Software." In: *Design Requirements Workshop (2009)*. Cleveland, Ohio, USA: Springer, pp. 467–494 (cit. on p. 17).

Schneider, Kurt (2009). *Experience and Knowledge Management in Software Engineering*. 1st ed. Berlin/Heidelberg, Germany: Springer (cit. on p. 84).

Shokripour, Ramin and John Anvik (2013). "Why So Complicated? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation." In: *10th Working Conference on Mining Software Repositories (MSR 2013)*. San Francisco, California, USA, pp. 2–11 (cit. on p. 121).

Skerrett, Ian and The Eclipse Foundation (2011). *The Eclipse Community Survey 2011*. Tech. rep. Ottawa, Ontario, Canada: The Eclipse Foundation (cit. on pp. 3, 19).

Sommerville, Ian (2011). *Software Engineering*. 9th ed. Boston, Massachusetts, USA: Pearson (cit. on p. 15).

Sultanov, Hakim and Jane Huffman Hayes (2013). "Application of Reinforcement Learning to Requirements Engineering: Requirements Tracing." In: *21st IEEE International Requirements Engineering Conference (RE 2013)*. Rio de Janeiro, Brazil: IEEE, pp. 52–61 (cit. on p. 124).

Sureka, Ashish and Pankaj Jalote (2010). "Detecting Duplicate Bug Report Using Character N-Gram-Based Features." English. In: *Asia Pacific Software Engineering Conference (ASPEC 2010)*. Sydney, Australia: IEEE, pp. 366–374 (cit. on p. 123).

Tian, Yuan, David Lo, and Chengnian Sun (2012). "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction." In: *19th Working Conference on Reverse Engineering (WCRE 2012)*. Kingston, Ontario, Canada: IEEE, pp. 215–224 (cit. on p. 121).

Tian, Yuan, Chengnian Sun, and David Lo (2012). "Improved Duplicate Bug Report Identification." English. In: *16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*. Szeged, Hungary: IEEE, pp. 385–390 (cit. on pp. 5, 123).

Toutanova, Kristina, Dan Klein, Christopher D Manning, and Yoram Singer (2003). "Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network." In: *Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL 2003)*. Vol. 1. Edmonton, Canada: Association for Computational Linguistics, pp. 173–180 (cit. on p. 25).

Tran, Ha Manh, Christoph Lange, Georgi Chulkov, Jürgen Schönwälder, and Michael Kohlhase (2009). *Applying Semantic Techniques to Search and Analyze Bug Tracking Data* (cit. on pp. 4, 51).

Trumble, William R (2007). *Shorter Oxford English Dictionary*. 6th. Oxford, England, United Kingdom: Oxford University Press (cit. on p. 14).

Uysal, Alper Kursat and Serkan Gunal (2014). "The impact of preprocessing on text classification." In: *Information Processing & Management* 50.1, pp. 104–112 (cit. on p. 21).

Vlas, Radu E and William N Robinson (2012). "Two Rule-Based Natural Language Strategies for Requirements Discovery and Classification in Open Source Software Development Projects." In: *Journal of Management Information Systems* 28.4, pp. 11–38 (cit. on pp. 100, 113, 122, 138, 150).

Vlas, Radu and William N. Robinson (2013). "Applying a Rule-Based Natural Language Classifier to Open Source Requirements: a Demonstration of Theory Exploration." In: *46th Hawaii International Conference on System Sciences (HICSS 2013)*. Hawaii, USA: IEEE, pp. 3158–3167 (cit. on pp. 22, 100, 113, 122).

Wang, Xiaoyin, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun (2008). "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information." In: *13th International Conference on Software Engineering (ICSE 2008)*. Leipzig, Germany: ACM Press, p. 461 (cit. on pp. 123, 154).

Wieringa, Roel J (2014). *Design Science Methodology for Information Systems and Software Engineering*. 1st ed. Berlin/Heidelberg, Germany: Springer (cit. on p. 45).

Witten, Ian, Eibe Frank, and Mark A. Hall (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd ed. Burlington, Massachusetts, USA: Morgan Kauffmann (cit. on pp. 14, 24, 33–36).

Wohlin, Claes, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén (2012). *Experimentation in Software Engineering*. Vol. 44. Berlin/Heidelberg, Germany: Springer (cit. on pp. 45, 46).

Yin, Robert K (2013). *Case Study Research: Design and Methods*. 5th. Applied Social Research Methods. Thousand Oaks, California, USA: SAGE Publications, p. 312 (cit. on pp. 45, 46).

Zhang, Tao and Byungjeong Lee (2011). "A Bug Rule Based Technique with Feedback for Classifying Bug Reports." In: *2011 IEEE 11th International Conference on Computer and Information Technology*. Paphos, Cyprus: IEEE, pp. 336–343 (cit. on p. 121).

— (2013). "A Hybrid Bug Triage Algorithm for Developer Recommendation." In: *28th ACM Symposium on Applied Computing (SAC 2013)*. Coimbra, Portugal: ACM, p. 1088 (cit. on p. 121).

Zhang, Tong (2004). "Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms." In: *21st International Conference on Machine Learning (ICML 2004)*. Vol. 6. Banff, Alberta, Canada: ACM Press, p. 116 (cit. on p. 36).

Zimmermann, Thomas, Rahul Premraj, Jonathan Sillito, and Silvia Breu (2009). "Improving Bug Tracking Systems." In: *31st International Conference on Software Engineering (ICSE 2009)*. Vancouver, Canada: IEEE, pp. 247–250 (cit. on pp. 4, 96, 176).

Zou, Weiqin, Yan Hu, Jifeng Xuan, and He Jiang (2011). "Towards Training Set Reduction for Bug Triage." English. In: *35th IEEE Computer Software and Applications Conference (COMPSAC 2011)*. Munich, Germany: IEEE, pp. 576–581 (cit. on p. 121).