USI

Holistic Recommender Systems for Software Engineering

Luca Ponzanelli

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera italiana

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Supervised by

Prof. Michele Lanza

Co-supervised by

Dr. Andrea Mocci

i

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Luca Ponzanelli
Lugano, 16 March 2017

To whoever backed me up
throughout this long journey.

Ever tried. Ever failed.
No matter.
Try again. Fail again. Fail better.

Samuel Beckett

Abstract

The knowledge possessed by developers is often not sufficient to overcome a programming problem. Short of talking to teammates, when available, developers often gather additional knowledge from development artifacts (*e.g.,* project documentation), as well as online resources. The web has become an essential component in the modern developer's daily life, providing a plethora of information from sources like forums, tutorials, Q&A websites, API documentation, and even video tutorials.

Recommender Systems for Software Engineering (RSSE) provide developers with assistance to navigate the information space, automatically suggest useful items, and reduce the time required to locate the needed information.

Current RSSEs consider development artifacts as containers of homogeneous information in form of pure text. However, text is a means to represent heterogeneous information provided by, for example, natural language, source code, interchange formats (*e.g.,* XML, JSON), and stack traces. Interpreting the information from a pure textual point of view misses the intrinsic heterogeneity of the artifacts, thus leading to a reductionist approach.

We propose the concept of Holistic Recommender Systems for Software Engineering (H-RSSE), *i.e.,* RSSEs that go beyond the textual interpretation of the information contained in development artifacts. Our thesis is that modeling and aggregating information in a holistic fashion enables novel and advanced analyses of development artifacts.

To validate our thesis we developed a framework to extract, model and analyze information contained in development artifacts in a reusable meta-information model. We show how RSSEs benefit from a meta-information model, since it enables customized and novel analyses built on top of our framework. The information can be thus reinterpreted from an holistic point of view, preserving its multi-dimensionality, and opening the path towards the concept of *holistic* recommender systems for software engineering.

Acknowledgments

4 years, 6 months, and 16 days. This is the exact amount of life I invested to achieve this dissertation and the consequent Ph.D. degree. Beside the academic results, this long journey allowed me to meet people from all over the world, getting to know different cultures, thus hopefully understanding the world in a better way than I used to. All of this has been shared with travel companions who had different, yet prominent, roles along the way.

First of all, I want to thank my advisor, Prof. Michele Lanza, for having given me such an opportunity. Michele did more than just advising: He supported and pushed me beyond what I believed to be my boundaries. Even though our views on some things might be completely orthogonal, he has always made me aware of the value of my work in spite of my harshest self-criticism. No advisor is required to do so, yet he did.

A heartfelt thank goes to Dr. Andrea Mocci with whom I literally worked side by side for almost four years. Andrea has just been one of the best mentor I ever had. The "second in charge" was the one responsible for the growth of several aspects of my research from both a technical and philosophical point of view. I learned a lot from him, including the meaning of "method", a fundamental one. Without all the support, in particular psychological, provided by Andrea, I could have not achieved the results of this dissertation.

A special acknowledgment is due to the committee of this dissertation: Prof. Carlo Ghezzi, Prof. Mehdi Jazayeri, Prof. Harald Gall, and Prof. Andrian Marcus. I had the pleasure of being a student of half of this committee, Carlo and Mehdi, during my master degree, and to meet the other half, Harald and Andrian, "on the field". Thank you for having accepted to be part of my committee, and thank you for the time spent reading, and understanding my thesis. Your feedback on my dissertation was really valuable and helpful, and made me proud of my work.

A shoutout goes to all the members of the REVEAL research group. Many people came and went during my Ph.D., but some endured until the very end: Roberto Minelli and Tommaso Dal Sasso. Both of you are complementary to what I am, yet it was a pleasure to travel and share the office(s) with you. I wish you all the best, and I hope you will conclude your Ph.D. journey soon. On the contrary, I wish a "safe and sound" Ph.D. journey to the youngest members: Bin Lin and Emad Aghajani. You are at the very beginning but I believe you have what it takes to do a great job. Keep up the good work.

The achievements I obtained during my Ph.D. are strictly related to the term "collaboration". For this reason, I would like to thank all the members of the Molise team: Prof. Gabriele Bavota, Prof. Massimiliano Di Penta, and Prof. Rocco Oliveto. This thesis would not have been possible without all the effort we made together. I took the best out of each of you and I learned things that I would not have learned otherwise. In particular the meaning of the term "empirical". A special consideration goes to Gabriele. In each location you were, from Benevento to Bolzano, and now in Lugano, your role of "sprint partner" allowed me to achieve outstanding results with respect to the time invested for each sprint. It was a pleasure to work with you, even when I had to bike for seven kilometers a day.

To conclude the academic section, I want to thank all the members of the Dean's office, Elisa Larghi, Janine Caggiano, and Danijela Milicevic, who performed an outstanding job in managing bureaucracy, reimbursements, and travel requests, sometimes with tight deadlines. If I manage to travel during my Ph.D is also because of your work.

Alongside the people encountered in the academic world, pursuing a Ph.D. requires support from the people you love and with whom you share some parts of this experience. This group of people is wide and diverse, thus involving closest friends and family. I want to start from the extended family, whose hard core is still in our home country, Italy, while others are spread throughout Europe. I cannot name all of you, but I want to thank you all. Even in the worst situations I always had all the needed support. You are just outstanding. An *ad personam* consideration is due to Adamà Faye. Without your support, even before the Ph.D back in 2010, I would have had harder times in achieving these results. We have been friends for more than 10 years now, and you were always there physically or remotely. Thank you.

Above all, I must thank all the members of my family. First and foremost, I want to thank my brother Claudio. We are really different and we chose different paths that hopefully will give us satisfaction on the way. I am proud of you. A special thanks to Elio Casiraghi, whom I am glad to have as part of my family. Since you joined us, you brought a kind of silent balance from which I hope to learn more and more. There is no right word to thank my mother, Gabriella Bianchi. You were my front-line supporter ever since I left Italy. You pushed me to pursue my objectives and goals to fulfill myself and build the person I am today. "Thank you" is just a reductive sentence. I also would like to thank my beloved father Ernesto Ponzanelli, and I do it in the simplest possible way: I wish you were here. What else can I say?

Last, but definitely most prominently, I thank Federica Oleda for being the person she is. You shared with me the last year of this long journey, probably getting the worse and more stressful parts. You supported me even when my choices clashed with what you desired the most. I consider myself lucky to have encountered you. I love you.

Luca Ponzanelli
March 2016

Contents

List of Figures

List of Tables

Part I

Prologue

1

Introduction

Developers often need to go beyond the knowledge they already possess to succeed in the completion of a programming task. Programming problems such as API understanding or bugs, are always behind the corner and developers often need to retrieve additional information, generally resorting to asking team mates [KDV07, HP00, LVD06] or consulting web artifacts such as forums, blogs, questions and answers (Q&A) websites, and API documentation [USL08].

The amount of resources at disposal of developers is vast, and the information they provide is getting a prominent role in the way developers perform a task. For example, both novice and experienced developers build applications by iteratively searching for, modifying, and combining examples whose presence in online resources like code repositories, documentation, blogs and forums has increased significantly [BGL+09a, BDWK10]. However, the process of searching the right piece of information that solves a programming problem at hand is time-consuming and requires considerable effort. Imagine a developer who wants to use a library without having knowledge of its functionalities, and resorts to the web to find out the solution: The developer types a query and retrieves at least ten different documents from the first page of a web-search result. Then, the developer needs to assess each document, or at least the title of the document, or to spot keywords from the textual summaries provided with the result (*i.e.,* as in a Google[1] search). If the developer has the feeling that a document could provide the needed information, she opens the related link and starts skimming the contents, gets the relevant parts, and then moves on to the next document, until she collects enough information to accomplish what she needs.

This search process has three clear and distinguished phases:

1. Query formulation and identification of a set of candidate documents;

2. Extraction of relevant parts of the information contained in different artifacts;

3. Assembly of all the relevant information units to obtain the desired solution.

Any improvement in the search process would result in time saved for the developers. For example, to help developers in the first phase, queries can be automatically generated [HBM+13] so that development artifacts can be retrieved according to a specific code entity, and notified to the developer from within the Integrated Development Environment (IDE). Recommender systems represent one possible implementation to provide developers with this technological support in the IDE.

---

[1] http://www.google.com

According to Robillard *et al.*, a Recommender System for Software Engineering (RSSE) is a *"software application that provides information items estimated to be valuable for a software engineering task in a given context"* [RWZ10]. Such *virtual* assistants, as RSSEs should be, implement approaches to discover and evaluate the pieces of information needed by a developer during a programming task.

The search process concerning online resources is not the only one performed by a developer. In a real situation additional resources are available to the developer, in particular in-project resources such as documentation, mailing lists, and bug tracking systems. All of them might contribute to the information needed by the developer, and should be considered by RSSEs as well. Several seminal approaches [CM03, HB08, ZWDZ04] used recommender systems to suggest these artifacts for a specific context in the IDE, but they consider the artifacts as sources of homogeneous information. In reality, such artifacts are heterogeneous, thus mixing narrative, source code snippets, and ad-hoc meta data that bring additional and valuable information that can only be analyzed by going beyond their textual representation.

1.1 The Backlash of Ockham's Razor

> *Pluralitas non est ponenda sine necessitate*
> (*Plurality should not be posited without necessity*)
> — William of Ockham (1285 – 1347)

The concept of recommender system does not come from software engineering but is the result of cross-fertilization between the fields of information retrieval (IR) and artificial intelligence (AI). In these fields, entities are represented through their constitutive elements. For example, to classify an item, the approach involves the extraction of features describing the item that are then put together in a feature vector, used as training set for a classification algorithm. A similar feature vector can be built for textual artifacts, where each feature corresponds to the frequency of a word in the text, and all the frequencies are put together in a *term frequency* vector [MRS08]. Alternatively, analysis on language models [MRS08] or topic modeling [BNJ03] focus on the probability distribution of the words in the text. The constitutive elements used in this case correspond to words and their interactions like n-grams or co-appearances in the same artifact.

Following the principle of Ockham's razor, such "imported" approaches are used as-is by considering development artifacts as purely textual. For example, Latent Semantic Analysis (LSI) [Dum04] and vector space models [MRS08] are used to recover traceability links between documentation and source code [ACCL00, MM03], and several algorithms for textual summarization have been used to summarize development artifacts like bug reports [MCSD12, LMC12, RMM14a].

These applications rely on the assumption that an artifact is a homogeneous container of information, made of text, where the constitutive elements (*i.e.,* words) are extracted from text by splitting the stream of characters on spaces and punctuation.

If this assumption holds in general, it does not when it comes to the contents of development artifacts. This type of artifact is unlikely to contain pure narrative, but instead, it is made of heterogeneous elements where the narrative is just one of the possible elements composing the contents. The constitutive elements for this type of artifacts change, and words become insufficient to describe all the types of information contained.

Conceiving development artifacts as containers of homogeneous information is reductive. Non-textual elements have their own structural information even though they are immersed in

natural language, and such information should be preserved for later use. Prominent examples of such type of artifacts are Stack Overflow[2] discussions, where natural language and source code live side by side with additional and heterogeneous elements like interchange formats (*e.g.,* XML, JSON), logs, images, and meta-data like reputation of users. In this case, the information provided by code snippets (*e.g.,* types, method invocations) extracted from an artifact is richer than, for example, a topic analysis (*e.g.,* LDA) on the same snippets.

In other words, Ockham's razor principle is overused, favoring simplicity over structure, and legitimating the absence of a *model.* This lack can be overcome by reusing an IR model (*e.g.,* vector space model), or by delegating the creation of the model to a machine learning algorithm (*e.g.,* neural networks). Since it is not possible to recover the original structure or to describe the interactions among constitutive elements, this reductive approach backfires when information needs to be manipulated or abstracted to build novel analyses.

The *modus operandi* described so far drifts away from the usual modeling approaches in software engineering. For example, when it comes to code analysis, approaches like srcML [CKM03] or the FAMIX meta-model [DDT99, DTD01] tackled the need of modeling source code to increase the level of abstraction and allowed, for example, to build general and language independent approaches to refactor and reengineering systems [TDDN00, DGLD05].

Why are development artifacts treated any different?

The set of artifacts perused by developers like bug reports, development emails, Stack Overflow discussions, source code, and even non-textual artifacts like video tutorials, all provide heterogeneous information that can be organized and modeled in the same way. For example, a development email and a video tutorial are different in their nature but they share a common structure for the information. For example, in both artifacts it is possible to establish what type of code elements (*e.g.,* types, identifiers) are used, which words are part of the narrative, which code elements are immersed in the narrative, or even what sentiment is expressed by the narrative (*e.g.,* positive or negative). Clearly these artifacts require two different processes, yet the outcome might be represented with a unified meta-information model, thus abstracting the nature of the artifact themselves.

As well as for code artifacts, a meta-model of the contents should be devised. The heterogeneity of the development artifacts should be preserved, such that each element composing the contents expresses a specific type of meta-information that together give a *holistic* view of the information. The concept of *holism*[3] is based upon the idea that *the whole is more than the sum of its parts* [Smu26]. In other words, novel factors can emerge when the entity is considered as a whole, without reducing it to its constitutive elements.

Following a *holistic* principle, if two heterogeneous entities like source code and narrative coexist in the same artifact, they preserve relationships that can emerge when the information is analyzed from several points of view. For example, narrative contains explicit references to source code and, at the same time, source code can be textually similar to narrative. Each of these entities describe a dimension of the information that needs to be preserved.

Analysis based on this idea would reach another level of abstraction where different entities preserve their nature and their own dimension in the overall information.

RSSEs would benefit from a meta-information model, since it enables customized and novel analyses built on top of it. The information can be thus managed and analyzed from a holistic point of view, where the information preserves its multi-dimensionality, opening the path towards the concept of holistic recommender systems for software engineering.

---

[2]`http://www.stackoverflow.com`
[3]from Greek (holos) "all, whole, entire"

1.2 Our Thesis

We formulate our thesis as follows:

> *Modeling information in a holistic fashion enables novel and advanced analyses of development artifacts, favoring reusability, and providing the foundations for holistic recommender systems for software engineering.*

To validate our thesis, we devised a framework to extract and model the heterogeneous contents of development artifacts, and we developed a set of applications built on top of its meta-information model, aimed at analyzing development artifacts by considering the information from a holistic point of view. The additional abstraction on the information enables novel analysis, and captures relationships among different types of information within an artifact, thus making novel and latent information emerge.

All of this can leveraged to revise current textual-based approaches in software engineering, and revamp the current approaches for RSSEs, thus laying the foundations of a novel type of Holistic Recommender Systems for Software Engineering (H-RSSE).

1.3 Contributions

The contributions of this dissertation can be classified in two categories. The first one concerns the modeling and analysis phase, while the second one concerns the tools based on such analyses. Except for the evaluation of CODETUBE and PROMPTER, all the work behind the core of every approach described in this thesis and listed as contribution, including their tool-based implementation, has been performed by the author of this thesis.

Modeling and Analysis

The contributions of this thesis include approaches aimed at modeling and analyzing of data coming from software development artifacts:

- We present an approach [PBL13a] based on off-the-shelf technologies imported from the information retrieval field (see Chapter 3).

- We devise a ranking model [PBD+14a, PBD+16] that evaluates the relevance of a Stack Overflow discussion, given a code context in the IDE, by considering code, conceptual and community aspects (see Chapter 4).

- We present a quality model to identify low quality post at creation time in Stack Overflow [PMBL14, PMB+14] by leveraging information concerning textual features of the post, and social aspects of the users to estimate the quality of a post (see Chapter 5).

- We devise a multi-lingual island parser capable of extracting constructs of interest like Java, XML, JSON, and stack traces within natural language, and modeling as Heterogeneous Abstract Syntax Tree (H-AST) (see Chapter 6).

- We devise a model of Stack Overflow discussions that, together with its original structure, models several types of information including natural language, readability metrics, code information (*e.g.,* types, invocations) in a novel meta-information model.

- We perform a study [MPM⁺15] to find and analyze usages of the undocumented Java class `sun.misc.Unsafe` in Stack Overflow (see Chapter 7).

- We propose a novel approach [PBM⁺16a] which mines video tutorials found on the web, splits them into coherent fragments, and complements them with information from additional sources, such as Stack Overflow discussions (see Chapter 8).

- We devise HoliRank, an extension of PageRank that runs on a similarity graph built by using our meta-information model (see Chapter 9).

- We present an navigation model [PSB⁺17] that leverages our meta-information model to create an informational context of the developer by leveraging HoliRank (see Chapter 10).

Tools and Artifacts

Another set of contributions of this thesis includes all the tools and artifacts implementing all the analyses presented:

- We present SeaHawk [PBL13a, PBL13b], an Eclipse plug-in that automatically formulates queries from the current context in the IDE, and presents a ranked and interactive list of results (see Chapter 3).

- We present Prompter [PBD⁺14b], an Eclipse plug-in that automatically searches and suggests Stack Overflow discussions if the a threshold set by the developer is surpassed in the IDE.

- We describe StORMeD[4], a publicly available dataset modeling more than 800k Stack Overflow discussions concerning Java, and an example of reusability of our H-AST and meta-information model. StORMeD is also available as free public island parsing service. (see Chapter 7).

- We present an automatic retagging tool which tags, by leveraging the StORMeD service, untagged code elements in Stack Overflow discussions (see Chapter 7).

- We present a holistic summarizer for Stack Overflow which employs HoliRank and StORMeD to implement a holistic summarizer for Stack Overflow discussion [PML15] (see Chapter 7).

- We present CodeTube [PBM⁺16b] a publicly available[5] search engine that enables developers to query the contents of video tutorials, and retrieve pertinent fragments (see Chapter 8).

- We present Libra [PSB⁺17], a holistic recommender system that provides support in navigating the information by augmenting the Google web page with a dedicated navigation chart (see Chapter 9).

1.4   Structure of the Thesis

Chapter 2 presents an overview of the state of the art. The chapter addresses different topics concerning this dissertation, including approaches to mine unstructured data, recommender systems for software engineering, and analysis of multimedia contents.

---

[4]`http://stormed.inf.usi.ch`
[5]`http://codetube.inf.usi.ch`

Part II: Reductionist RSSEs

In the second part of this dissertation, we present three approaches for recommender systems, which are built on top of what is currently available in the state of the art.

Chapter 3 describes SEAHAWK, an approach to automatically retrieve and link Stack Overflow discussion in the IDE. SEAHAWK takes advantage of off-the-shelf IR approaches based on pure textual analysis.

Chapter 4 describes PROMPTER, an approach that automatically searches for Stack Overflow discussions on the web, estimates their relevance compared to a code context in the IDE, and fires notifications if and only if a confidence threshold is surpassed. PROMPTER employs a model that considers textual information, user reputation, and code information (*e.g.,* API type names), all obtained with off-the-shelf tools.

Chapter 5 describes a classification approach to automatically identify low quality posts in Stack Overflow that should be put under review. The approach uses heterogeneous information not strictly related to code (*i.e.,* readability, textual features, popularity metrics). The work reported in this chapter is the result of an industrial collaboration with Stack Exchange Inc, New York.

Part III: Parsing and modeling unstructured data

In the third part of this dissertation we introduce an approach to parse and model heterogeneous artifacts, and we present a set of approaches built on top of it.

Chapter 6 describes a multi-lingual island parser approach capable of creating a Heterogeneous Abstract Syntax Tree (H-AST) of a textual development artifact. We take advantage of this approach to develop a model for Stack Overflow discussions which features a meta-information model carrying several types of information.

Chapter 7 describes STORMED, a publicly available dataset modeling more than 800k Stack Overflow discussions concerning Java, and an example of reusability of our H-AST and meta-information model. We take advantage of STORMED to find and analyze usages of the undocumented Java class `sun.misc.Unsafe` in Stack Overflow, and we also present an automatic retagging tool for Stack Overflow.

Chapter 8 presents CODETUBE, an approach to mine and analyze the contents of video tutorials. Our approach mixes several techniques of image analysis, and supports them with analysis on the H-AST provided by our multi-lingual island parser.

Part IV: Holistic RSSEs

In the fourth part of this dissertation we present two approaches built on top of the meta-information framework devised previously, which enable the first holistic analyses of development artifacts.

Chapter 9 shows how textual summarization algorithms like LEXRANK can be revisited and extended to support holistic analyses. We describe HOLIRANK, a customized version of PAGERANK leveraging the meta-information model of STORMED to revisit the concept of similarity between two heterogeneous units of development artifacts.

Chapter 10 presents LIBRA, a Holistic Recommender System for Software Engineering (H-RSSE) that helps developers in searching and navigating the information needed by constructing a holistic meta-information model of the resources perused by a developer, analyzing their semantic relationships, and augmenting the web browser with a dedicated interactive navigation chart.

Part V: Epilogue

Chapter 11 concludes the dissertation by discussing and reflecting on the contributions of this work, the challenges addressed, and the ones left for future research.

2

State of the Art for RSSEs

In this chapter we present the genesis of RSSEs, the role of recommenders in the development chain, and the motivations that brought them on the scene. We present an overview of the types of analysis proposed by the most prominent approaches in literature, and we conclude by outlining their current limitations.

2.1 A Genesis of RSSEs

Program (Software) Evolution is a term used by Lehman for the first time in the mid seventies [Leh78]. The term comes from a study performed on the evolution of the OS/360 system at IBM [Leh69]. From that experience, Lehmann devised eight laws for software evolution [LB85], describing the implications derived by the fact that software systems grow and change over time. According to Lehman, software systems are continuously evolving over time, as a result of an intrinsic need of adaptation due to a drift between the software and its operational domain (law 1), as a system evolves its complexity increases (law 2), its quality decreases (law 7), and the system needs to meet new functional requirements to maintain user satisfaction over time (law 6). As Lehman pointed out, performing constant maintenance on software systems is due to their evolution and increasing complexity. As a result, software maintenance accounts for 50 to 90% of the overall system costs [LS80, Erl00].

Another aspect to consider when looking at the impact of software maintenance concerns the time needed to understand systems by developers. Several studies estimate program comprehension to take more than half the time spent on maintenance [ZSG79, MML15]. These two factors combined together highlight that getting an understanding of systems impacts for 30 to 50% of the overall costs [FH82].

The quality of documentation is one of the problematics affecting the understanding of the system at hand. Indeed, the documentation of software systems is often inadequate [LVD06], or even unavailable, thus lowering the overall support for developers. Navigating source code is one way to recover knowledge about the system [KDV07], but it might be not enough. To overcome this, developers consult other, potentially more experienced, teammates [KMCA06] to retrieve the desired information.

Providing developers with better support to understand the software system they are working on has always been a challenge. Corbi pointed out that *"programmers have become part historian, part detective, and part clairvoyant"* [Cor89]. Understanding systems at hand, maintaining them, or meeting new user requirements, requires a deep knowledge about the systems themselves. Corbi also suggested that *"software renewal tools are needed to reduce the costs of modifying and*

*maintaining large programming systems, to improve our understanding of programs so that we
can continue to extend their life and restructure them as needed"* [Cor89].

The need for tools to improve understanding is even more justified nowadays. Developers do
not limit themselves to teammates or documentation, but they often extend their search to online
resources such as tutorials and Q&A websites [USL08], spending around 20% of their development
time consulting them [BGL+09b]. The advent of internet, and crowdsourcing platforms like Stack
Overflow opened the path to a vast amount of information, and inevitably changed the developers
habits in retrieving the needed information. For example, as a direct consequence, venues for
knowledge exchange moved from mailing lists to Q&A websites like Stack Overflow [VSDF14].

Tools to help developers gather information among the available resources would ideally
reduce the amount of time spent by developers in understanding the system and researching
the desired information. Similar tools to suggest items of interest already exists outside the
context of software engineering. For example, e-commerce websites like Amazon[1] or Ebay[2] take
advantage of the history of the purchases, items recently viewed, and past searches to suggest
items of interest to customers. Another example concerns streaming services like Netflix[3] or
Hulu[4], where users receive suggestions aimed at supporting them in deciding what movie or TV
series to watch depending on what is trending among other users, what the user has already
watched, or the preferences the user explicitly expressed.

These tools are named *recommender systems*, and their aim is to provide suggestions for
useful items to a user [RRSK10], thus supporting users in their decision-making processes while
interacting with a large information space. The items suggested by recommender systems are
generally specific and tailored to the user's profile, usage, or preferences.

In the context of software engineering, recommender systems aim at supporting the informa-
tion seeking process [RWZ10]. In other words, a Recommender System for Software Engineering
(RSSE) supports developers in the navigation of the information space at their disposal by sug-
gesting items from several and diverse source of information either in-project or online. Items
can be API documentation, code samples, experts, bug reports, Q&A websites *etc.* that can
speed up both maintenance and forward development. RSSEs can thus play an important role
in the developer toolset, and help getting a better understanding of the system at hand.

2.2   Recovering Traceability Links

Recovering links between code and documentation [ACCL00] and enhancing API documentation
are common tasks for RSSEs. Several types of resources can be used for these approaches.
For example, Petrosyan *et al.* [PRM15] presented an approach to discover tutorial sections
explaining how to use a given API type, by analyzing the API types mentioned in the sections.
These tutorial sections can be leveraged to enrich API documentation (*i.e.,* Javadoc), by linking
them directly. Another approach is the one by Robillard and Chhetri [RC15], which aims at
distinguishing "reference" (*i.e.,* indispensable, or at least valuable) parts of API documentation,
from less valuable details. To achieve this result, their approach leverages NLP processing, and
relies on a set of heuristics (*e.g.,* camel-case notation) to identify API types in the text within
`<code>` tags. These heuristics are implemented in KREC, a plugin for the Eclipse IDE[5] that links
relevant API documentation parts to the code at hand in the editor.

---

[1] http://amazon.com
[2] http://ebay.com
[3] http://netflix.com
[4] http://hulu.com
[5] http://eclipse.org

Traceability links are spread across several types of artifact and include more than API documentation and Stack Overflow discussions. Indeed, in-project knowledge is another example resource where approaches for RSSEs focused on. This knowledge coming from within a project includes development emails, bug reports, general project documentation, and even knowledgable or expert developers. Approaches focusing on this type of information often require to analyze the history of a software system, and mine data from its repository and related sources (*e.g.,* mailing lists, bug trackers) to extract relevant information. The seminal work concerning recommender systems harnessing data from repositories lies in EROSE, an approach proposed by Zimmerman *et al.* [ZWDZ04]. EROSE mines a repository to suggest code co-edits to be performed according to the history of the system. Another prominent and seminal work is HIPIKAT [CMSB05], a tool that builds a project memory from artifacts created during a software development project (*e.g.,* source code, documentation, emails), and recommends such artifacts if they result relevant to the task performed. HIPIKAT uses a set of heuristics to establish relationships among activity and artifacts that includes analysis of logs, activity, CVS revisions, and textual similarity. Holmes and Begel [HB08] tackled the problem of the information overload due to multiple type of information and artifacts. DEEPINTELLISENSE summarizes and displays historical information (*e.g.,* modifications, people involved) about source code, to help developers build a cohesive mental model of the rationale behind the code.

One important aspect of recovering traceability links concerns the discovery of code elements within artifacts. Several approaches rely on textual analysis techniques, delegating the discovery to third party algorithms establishing a sort of "semantic" link between code and description. Other approaches tackled this problem in a more systematic way, trying to isolate the code elements within narrative, and then match such elements in source code to establish a link. A first attempt has been performed by Bacchelli *et al.* [BLR10]. Their approach uses regular expressions to identify code elements (*e.g.,* classes, packages) in development emails and establish a link with source code. On top of this approach, Bacchelli *et al.* developed REMAIL [BBL11], a plugin for the Eclipse IDE that suggests development emails concerning the code at hand. Following a similar approach, Rigby and Robillard [RR13] used a set of regular expressions to identify essential code elements in development artifacts like type names, method invocations, annotations, packages, and import statements.

Other approaches rely on grammars to analyze artifacts. For example, Dagenais and Robillard [DR12] leveraged Partial Program Analysis (PPA)[DH08] to cope with partial programs, perform partial type inference, and recover the declared type of expressions. This approach identifies API related code elements (*e.g.,* types) in documentation, and establish a link with source code. Instead, Bacchelli *et al.* [BCLM11] devised an island grammar capable of isolating Java code elements and stack traces from natural language. In another work, Bacchelli *et al.* [BSDL12] leveraged the island parser and machine learning (*i.e.,* Naïve Bayes [FGG97]) to classify different contents (*e.g.,* stack traces, patches, Java code, garbage) in development emails.

A third way to identify code elements within artifacts concerns the pure usage of machine learning approaches, thus avoiding the definition of ad-hoc grammars and regular expressions to match code elements. For example, Cerulo *et al.* [CPB+15] leverage characters distribution and features to build a Hidden Markov Models [BP66] that manages different constructs (*e.g.,* stack traces, XML, Java code) without needing regular expressions or grammars.

2.3   Web Resources

In the previous section we gave an overview of RSSEs harnessing data mostly built in the context of a project. However, the modern developer has a vast amount of resources at disposal by accessing and retrieving information from the web, one of the main targets of developers while searching for information [USL08]. Recommending relevant web artifacts to help solve coding problems, or to automate web searches according to the contextual information available in the IDE, is another type of task that well suits RSSEs.

The importance of web resources as one of the main targets to scavenge information highlights two different, yet complementary, tools used by developers: the IDE and the web browser. To access web resources, developers need to leave the IDE, thus switch the context they are working in, and search for interesting artifacts on the web by using a search engine.

Reducing the context switch is thus the goal of RSSEs. The goal can be achieved by enhancing the IDE, the web browser, or both of them. When RSSEs focus on the IDE side, the goal is to prevent developers to switch to the web browser by providing web resources in the IDE. For example, Brandt *et al.* [BDWK10] presented Blueprint, a plug-in for Adobe Flex Builder[6] that automatically augments queries with code context, extracts code examples from Web pages, and composes them in the code editor. This approach leverages the HTML structure of web pages to separate code from text, and verifies the text tagged as code by using heuristics based on characters features assumed to be unique to code such as curly braces, language keywords, and lines ending with semi-colons.

On the same line, Sawadsky and Murphy developed Fishtail [SM11], a tool to discover code examples and documentation on the web relevant to the current task. Fishtail leverages the history of programmer interactions with the source code to automatically determine relevant web resources, and notify them in the IDE. Fishtail relies on Mylyn [KM06] to identify the code elements with the highest degree-of-interest (DOI) in the IDE, and triggers web searches whenever the element with the highest DOI changes. Queries are built by using the signature of a code element, and then used to perform a Google search.

Another way to reduce context switch is to enhance the web browser side by providing ad-hoc search engines or by augmenting actual web search results. For instance, Hoffmann *et al.* proposed Assieme [HFW07], a web search interface that finds and resolves implicit references to Java packages, types, and members within samples of code on the Web. Assieme collects web pages that are likely to contain code samples, and uses Google to find pages with keywords which frequently appear in Java code (*e.g.,* import, class, interface), and provides an interface tailored to navigate API information. The code extraction is performed by using the Eclipse JDT.

Similarly, Stylos and Myers [SM06] developed Mica, a search tool that augments Google search results by providing a description of the desired functionality and help programmers find examples when they already know which methods to use. Mica allows the developer refine the search results by selecting keywords from a set displayed aside.

Another prominent example, focused on code search on the web, is ParseWeb [TX07], a tool that retrieves and analyzes code from several open source repositories on the web, and builds code samples aimed at reusing existing frameworks or libraries.

A third and last way to reduce context switch consists in breaking the boundaries between IDE and web browser and leverage them to gather more contextual information. For instance, Goldman and Miller presented Codetrail [GM09], a tool that uses a communication channel and shared data model between the IDE and the web browser to combine information gathered

---

[6]http://www.adobe.com/products/flex.html

from the two (*i.e.,* editing history, code contents, and recent browsing). CODETRAIL makes the web browser an additional IDE view that enables additional features like bookmarking of web resources in the resources code from both the IDE and the web browser, and automatic documentation browsing in the web browser by synching with the current element under the cursor in the IDE. A similar approach is proposed by Hartmann and Dhillon [HDC11] in HYPERSOURCE, an augmented IDE that associates browsing histories with source code edits. Their approach takes advantage of both the web browser and the IDE to track the history of visited pages and code edits, establish a link between the two, and mark the web resources directly in the code editor where the code change has been performed. These marks are interactive and allow the developer to review the browsing history concerning a specific change.

Also Sawadsky *et al.* [SMJ13] followed a similar idea and developed REVERB, a tool that extends both the Eclipse IDE and the Chrome[7] web browser. Their approach monitors the web pages visited in the web browser, and indexes them by using Apache Lucene[8], and also tracks interactions in the IDE to understand the element currently displayed. When the element displayed changes, REVERB queries the Lucene index, retrieves a list of visited pages, and shows them directly in the IDE.

2.4 Stack Overflow as Source of Information

Among the available online resources, Q&A services provide developers with the infrastructure to exchange knowledge in the form of questions and answers [AZBA08]. Developers ask questions and receive answers regarding issues from people that are not part of the same project, performing what is defined as *crowd sourcing* a task. Even though researchers pointed out that Q&A services could not provide high level technical answers [NAA09, MMM+11], these services are filling *"archives with millions of entries that contribute to the body of knowledge in software development"* and they often become the substitute of the official project documentation [TBS11].

The impact of Stack Overflow on the way developers exchange and transfer knowledge [VSDF14] captured the attention of researchers who raised a set of questions concerning the impact of this Q&A website on software engineering practices and the tools [STvDC10]. For example, Storey *et al.* [SSC+14] found that while traditional channels (*i.e.,* mailing lists, face-to-face communication) are still considered crucial, social media like Stack Overflow *"have led to yet another paradigm shift in software development, with highly tuned participatory development cultures contributing to crowdsourced content"*.

Stack Overflow relies on a very active community asking and discussing a considerable amount of questions daily, providing an answer rate above 90%, and a median answer time of only 11 minutes [MMM+11]. At the time of writing, by querying the Stack Exchange Data Explorer[9], Stack Overflow accounts for more than 6 million users who asked more than 12 million questions. This critical mass of information makes Stack Overflow an ideal resource for RSSEs.

Stack Overflow can be leveraged as a source of information for RSSEs to automatize the identification of relevant help online. For example, Cordeiro *et al.* [CAG12] proposed to process the contextual information of stack traces to retrieve pertinent Stack Overflow discussions to help developers in the IDE when a runtime error happens. Their approach relies on a the HTML tagging of the discussions to identify code blocks, and analyzes the code with a combination of regular expressions and Eclipse JDT aimed at identifying stack traces or Java code respec-

---

[7]https://www.google.com/chrome/
[8]http://lucene.apache.org/
[9]https://data.stackexchange.com/stackoverflow/query/new

tively. Another example is DORA [KDSH12], a tool integrated into the Visual Studio IDE[10] that automatically queries and analyzes online discussions (*e.g.,* Stack Overflow, Codeguru, Bytes, Daniweb, Dev Shed) to locate relevant solutions to programming problems. DORA searches for discussions by using the search engine provided by a website like Stack Overflow, and evaluates the quality of the retrieved discussions by relying on a model based on community-related features (*e.g.,* number of replies, resolved answer).

Mixing community features in Stack Overflow and textual features to retrieve relevant help is a goal targeted by several approaches when building their own search engine, without reusing existing ones. For example, Campos *et al.* [CdSdAM16] devised a retrieval approach that takes into account pairs composed by a question and an answer, and evaluates them by composing three aspects: (1) the score of Apache Lucene, (2) the score of received by the Stack Overflow community, and a score that determines the *How-to* nature of a pair. Similarly, Zagalsky *et al.* [ZBY12], presented EXAMPLE OVERFLOW[11], a search engine for Javascript code samples that allows the developer to retrieve samples for the JQuery[12] library according to a textual query. EXAMPLE OVERFLOW uses a score function to estimate the overall quality of code samples that mixes the score given by Apache Lucene, and the community scores assigned to each part of the discussion (*i.e.,* title, question, answers, code) from which the sample is taken.

Stack Overflow discussions are also used as resource to enrich current documentation. For instance, Subramanian *et al.* [SIH14] presented BAKER, a tool that augments API documentation (*i.e.,* Javadoc) with code samples taken from Stack Overflow, and viceversa. Their approach employs the Eclipse JDT parser to reconstruct a partial AST of the code sample found between `<code>` tags to identify fully qualified names that pertain to API usages. The fully qualified names are then used to dynamically inject code samples in the API documentation and link the documentation within the corresponding Stack Overflow discussion, thus favoring navigation between the two resources. A similar approach was devised by Treude and Robillard [TR16]. Their approach enriches current API documentation with summarized information taken from Stack Overflow discussions to describe usages of classes and methods. They employ regular expression to identify code elements within HTML, and select relevant sentences by using SISE (Supervised Insight Sentence Extractor), a summarization approach that considers several factors concerning community aspects (*e.g.,* user reputation, score, favorites, views), textual aspects (*e.g.,* part-of-speech tags), and code related aspects (*e.g.,* API elements in the sentence), to build an extractive summary. The summary can be thus injected in the API documentation to provide additional information concerning real usages. Differently from the previous ones, Wong *et al.* [WYT13] enrich software documentation by automatically generating comments for source code. Their approach, called AUTOCOMMENT, leverages and refines the code description found in Stack Overflow, identifies, by using code clone detection techniques, related code elements in source code, and generates comments accordingly.

RSSEs for Stack Overflow were also designed to assess the quality of posts. Indeed, the quality of Stack Overflow posts has many implications for developers. On the one hand, the Stack Overflow community aims at keeping a certain level of quality in their posts, and lets the crowd judge and filter out low quality posts. On the other hand, an automated approach can help the crowd in identifying such low quality posts faster, and developer in dodging them when looking for help in Stack Overflow. For example, Correa and Sureka [CS13, CS14] devised an approach to automatically identify questions to be deleted or closed. Their predictive models detect the quality of a question at the creation time, and use a set of features concerning user

---

[10]`https://www.visualstudio.com`
[11]`http://www.exampleoverflow.net`
[12]`https://jquery.com`

profile (*e.g.,* badges count), community aspects (*e.g.,* score, favorites), question contents (*e.g.,* number of URLs), and syntactic style (*e.g.,* number of uppercase and lowercase letters).

2.5   Reflections on the State of the Art

All the approaches discussed in this section represent the state of the art for RSSEs. An aspect concerning these approaches is the way the information is extracted and modeled from text. In many cases, as described in Chapter 1, current RSSEs delegate the extraction phase to machine learning approaches or to naïve text processing with regular expressions. For instance, the isolation of code from natural language might be performed by analyzing the distribution of features of characters (*e.g.,* [HFW07]) like the frequency of brackets and semi-colons, and then use this information to build a machine learning approach (*e.g.,* [CPB$^+$15]) instead of adopting a systematic approach based on structure and grammars. Even when the identification of code elements is performed with regular expressions (*e.g.,* [RR13]) and grammars (*e.g.,* [BCLM11]), where code constructs are indeed identified and isolated, the final outcome remains in a plain textual form, still not modeling the contents.

All the information concerning the structure of the code elements identified in the artifacts is lost, hindering the manipulation and aggregation of the information afterwards. To overcome this problem, several approaches analyze these identified elements with parsers capable of handling errors such as the Eclipse JDT. In the state of the art, the usage of the Eclipse JDT to isolate code within narrative is considered a normal procedure. Even though the "best effort" approach implemented in the Eclipse JDT reconstructs minor errors (*e.g.,* missing semicolons), it is definitely not designed and implemented to cope with narrative. Indeed, the sentence *"REs are not Turing complete, so they cannot be used for island parsing"* is parsed by the Eclipse JDT as `REs are; Turing complete, so; be used; island parsing;` when it tries to reconstruct valid Java code.

Let us assume for a moment that the Eclipse JDT works for such applications, and produces a correct AST of Java code snippets described in an artifact. *Is Java enough to describe the contents of development artifacts?* The assumption latently lying in the current approaches concerns the general existence of one single language: Java. Clearly, reality is different. For example, the "unique language" assumption falls apart whenever an Android application is analyzed. In this case, Java has the lead in building the application, but several parts are delegated to other languages. User Interfaces are one prominent example, where XML is the language defining the composition of various buttons, bars, and labels interacting with the user. A user interface written in XML establishes explicit links with Java code (*e.g.,* callbacks and delegates) in the attributes of entries. Correctly capturing this information requires an XML parser. Services and the distributed nature of modern applications is another example. Generally modern applications rely on external REST services. The interchange formats used to communicate with these services can be either XML or JSON and, once again, they describe part of the behavior of the application that might be lost. One could state that three off-the-shelf parsers for Java, XML, and JSON are enough, and they can be used subsequently on the specific parts. Even if the final result might be achieved, it would require several passes, resulting in three different ASTs among each other, with no immediate possibility of manipulation or analyses.

As already stated in Chapter 1, such approaches do not provide any proper modeling for the contents of development artifacts. The absence of a common model for the contents highlights another problem concerning the sources managed by RSSEs. Most of the approaches presented are tailored to handle one specific type of artifact (*e.g.,* Stack Overflow discussion, bug report,

clean source code), or one type of information (*e.g.,* edit history). Developers need information from more than one source at time, motivating the need of a multi-source RSSE. Unfortunately, this aspect is currently underestimated. Indeed, these sources are treated as separated entities without a common model for the information.

A consequence of multi-source RSSEs concerns the information overload due to the amount of information that several items suggested together may achieve. This problem is partially tackled in the state of the art. Several approaches allow to navigate different types of information mined from different sources (*e.g.,* [HB08]), as well as history information from different applications like the IDE and the browser (*e.g.,* [HDC11]). The common problem is that RSSEs stop acting once the information is displayed, and do not really provide a guidance through the information provided by several artifacts. For example, a web search provides a set of artifacts which may provide a considerable amount of information. Current approaches try to synchronize IDE and web browser [GM09], or enhance the search results with code information. Even though these features help the developer, a real guidance in selecting the right result, or avoiding reading results providing information already acquired, is needed to reduce the time needed for understanding.

In this dissertation we cope with the modeling limitation of the approaches proposed in the literature by describing a journey starting within the current state of the art described so far. We begin by leveraging off-the-shelf tools to build cutting edge RSSEs. We expose the limitations imposed by such tools, and we describe our effort towards modeling development artifacts and their information. We propose a common framework to model the heterogeneous contents of an artifact, and the information provided by it with the final aim, at the end of this journey, of devising the first holistic recommender system for software engineering.

Part II

Developing RSSEs with
off-the-shelf tools

3

Leveraging Crowd Knowledge for Software Comprehension and Development

As described in Chapter 1, developers often need to gather additional information, beyond what they already possess, to complete the programming task at hand. RSSEs support developers in the retrieval of such information by recommending items of interest pertinent to the current context. As discussed in Chapter 2, actual approaches rely on reductionist recommendation engines, which in turn rely on a reductionist analysis of the artifacts. The underlying approaches used to analyze artifacts generally exhibit the backlash of the Ockham's Razor principle, since they treat an artifact as a bag of words without caring about the unstructured nature of the contents, thus relying on off-the-shelf tools borrowed from the information retrieval field.

In this chapter we describe an approach that follows this *plug 'n play* attitude of reusing off-the-shelf tools from information retrieval in Software Engineering. We present SEAHAWK[1], a recommender2 system in the form of a plugin for the Eclipse IDE to harness the crowd knowledge of Stack Overflow from within the IDE. SEAHAWK mines the Stack Overflow knowledge base and creates an index of the discussions by relying on its textual representation.

SEAHAWK allows the developer to search for Stack Overflow discussions directly in the IDE, to link discussions to code entities, and to import code snippets. It also offers the possibility to automatically generate queries by extracting keywords from the code entities given in the IDE. To evaluate the approach implemented in SEAHAWK, we present and discuss a series of experiments aimed at understanding the quality of the recommendations.

Structure of the Chapter

In Section 3.1 we present SEAHAWK and the approach relying on a pure information retrieval approach. In Section 3.2 we illustrate its usage with a scenario, and present an evaluation in Section 3.3. In Section 3.4 we present a summary of the chapter and we draw our conclusions.

3.1 SeaHawk

Figure 3.1 depicts the user interface of SEAHAWK. Users are provided with four main components to interact with SEAHAWK: (1) *Document Navigator View*, where the user can type in a query and navigate the returned documents, (2) *Suggested Document View*, where SEAHAWK suggests documents that are linked by the annotations in the code, (3) *Document's Content View*, where

---

[1] http://seahawk.inf.usi.ch
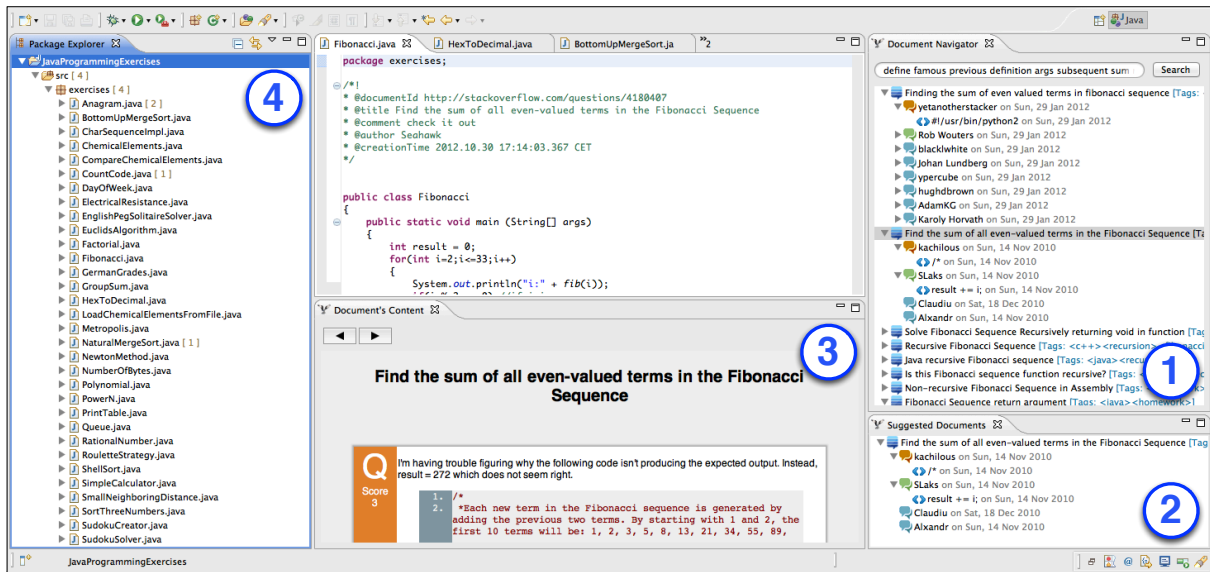
Figure 3.1. SeaHawk User Interface

the content of the current document is presented to the user, and (4) a notification system inside the package explorer to notify developers of new linked documents.

3.1.1   The Architecture

In the following we present the architecture of SeaHawk according to the structure of a recommendation system defined by Robillard *et al.* [RWZ10]: *A data-collection mechanism*, a *recommendation engine* and a *user interface*.
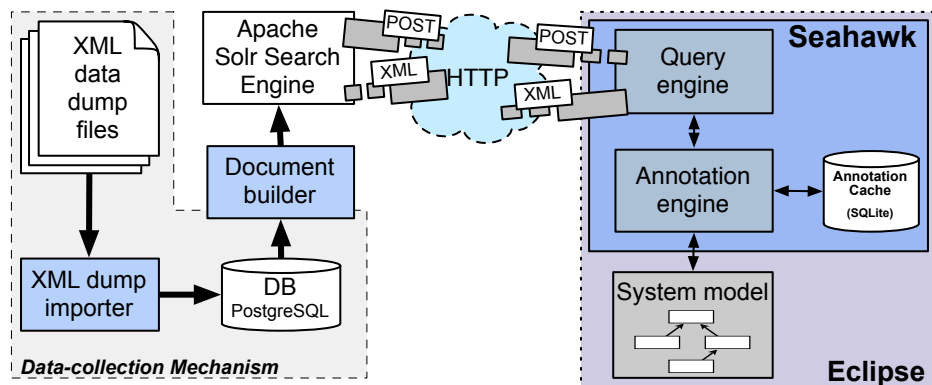
Figure 3.2. SeaHawk's Architecture

Figure 3.2 depicts SeaHawk's architecture. The first component in SeaHawk is the data collection mechanism, which is responsible for gathering Q&A data from Stack Overflow. We import Stack Overflow documents from a public data dump provided as local XML files. The data is extracted through a *XML dump importer* and stored in a relational database for performance

reasons. We built a tool to query the database and to build a JSON representation of each document (thus making it available for any language). This representation is then included in an additional document schema required by the Solr[2] search engine. When documents are indexed by the search engine, they become available and can be retrieved by querying the service. Solr rovides a RESTful interface to perform searches by means of GET and POST requests and it replies with XML data with the relevant documents.

According to Robillard *et al.*, interaction with recommendation systems can be both manual (*i.e.,* a query is inserted by the user) and automatic (*i.e.,* the recommendation engine generates the query) [RWZ10]. SeaHawk supports both: users can manually write queries to retrieve documents, or SeaHawk can extract keywords from code entities, build a query, and suggest documents. SeaHawk also provides an annotation system that allows developers to link Stack Overflow documents to the source code.

3.1.2   Data Collection Mechanism

Stack Exchange offers a public RESTful API[3] to access the Stack Overflow content. Since the API is limited in usage and search capabilities, we decided to use the public data dump[4] provided by Stack Exchange, which comprises several XML files that represent the database of each website. We limit the files needed to the ones representing the data (*i.e., posts.xml, users.xml, comments.xml*), discarding the files regarding the evolution of the website (*e.g., posthistory.xml, badges.xml, votes.xml*), since we are not interested in data regarding the interactions of the users with the community.

On the left hand side of Figure 3.2, we depict the process to import and manipulate the data to reconstruct documents indexed by the search engine. We consider three XML files: *posts.xml, users.xml, comments.xml*. The total amount of entries in *posts.xml* sums up to more than 7 millions. To recreate a document, we need to gather a question and all the related answers from *posts.xml*. For the opening question and all answers, we extract information regarding users (*users.xml*), comments (*comments.xml*) and authors (*users.xml*). Since performing these operations by manipulating data directly from the XML files is resource intensive, we import everything in a database (thus also easing the document extraction). We chose to represent documents in JSON format to make them portable. To build documents, we implemented an importer that queries the database. Those documents are then included in an additional document representation and indexing required by Solr.

The Search Engine

The search engine indexes documents when extracted and reconstructed from the database, and makes them available for queries. We take advantage of Solr which stores and indexes documents in a vector space model, relying on Apache Lucene[5] as core engine. The weighting algorithm used by Apache Lucene, and thus by Solr, is a variation of *tf-idf* [MHG10]. We configured Solr to remove stop words, filter out possessive words, stem words, trim white spaces, filter synonyms and lower the case at both query and indexing time.

Once the indexing phase is complete, the Solr engine can be queried via HTTP in a RESTful fashion. SeaHawk can thus query the search engine to get relevant documents in XML format

---

[2]http://lucene.apache.org/solr/

[3]https://api.stackexchange.com

[4]http://www.clearbits.net/creators/146-stack-exchange-data-dump

[5]http://lucene.apache.org/

that contain the JSON representation of the original ones. The documents are then deserialized and shown in the Eclipse IDE.

3.1.3   The Recommendation Engine

The recommendation engine of SeaHawk provides both manual and automatic interactions. The core is composed of a query engine and an annotation engine.

The Query Engine

SeaHawk's Eclipse plugin makes the Q&A crowd knowledge available in the IDE. Users can interact with this knowledge in ways that the website normally does not allow, such as directly manipulating code snippets. The main goal of the query engine is to communicate with Solr by creating a query given an input string. Being Q&A documents the target of such queries, it is likely to have some information also in the title, that is, the question itself. We assign more weight to the document's title to exploit possible keywords that can be relevant for the target search. Let us assume that a developer wants to query the search engine with the following query: "change label color in Java". The query engine tokenizes the string inserted by the developer. The engine builds the query, according to Solr syntax, in a way that every token must be present in the document field or at least one of those is contained in the title field.

In the query, the overall relevance of a document is determined by the relevance of the body of the document and its title. Documents whose title is interesting for the given query (*e.g.,* titles containing words such as *label* or *color*) are retrieved even if the document's body does not match any of the tokens.

Automation of Queries

The query engine also provides an automatic keyword extraction feature to build queries. The first technical issue to overcome regards the code written by developers. Developers need to understand their code even though it does not compile. Dealing with code that does not compile has drawbacks: Compiling code can provide a full Abstract Syntax Tree (AST), but with compilation errors the AST can be partial or even absent. Moreover, the partial AST is the representation of the code until the compilation failed, thus discarding any additional information that comes after. This also applies for Eclipse when it is asked to produce an AST for a Java program.

To overcome this problem we use island parsing [Moo01]. It copes with code that does not compile, to identify structural information of code entities (*i.e.,* class and methods) and discard the uncompilable parts. The Eclipse IDE provides a framework to apply similar parsing approaches to Java code: It identifies classes, methods and fields in a source file even though the compilation fails. We employed this framework to parse the code with the single constraint of being Java-dependent for this feature.

Since we do not have complete AST information for the identified code entities, code entities are treated as text and analyzed as natural language. The target entity is defined by the cursor position in the text editor, the nearest entity is picked as target entity.

When an entity is selected, the query is built by merging the keywords obtained in two ways:

1. *Processing the entity's body.* We apply basic information retrieval techniques to extract the ten most frequent keywords in the body. We tokenize the entity's body on white spaces. For every token, we split it on case change, digits and symbols. We lower the case and

remove stop words. The set of tokens we obtain is ordered by frequency and the first ten become part of the query. To this set of keywords we add the entity's name. This is done because of Java interfaces. If the entity is a method, including the name would enhance the research. Being immutable, the method's name of a Java interface in a library, or a framework, is always the same. A Stack Overflow document would contain this method's name if one of the code snippets is tackling the implementation of a specific interface. For instance, a developer can invoke SEAHAWK on the method *decorate* implementing the *ILightweightDecorator* interface in the Eclipse API, an interface used to perform a custom decoration of the package explorer. In this situation, documents containing code snippets that implement the interface are enhanced because the term "decorate" is used. Moreover, the same term could be also used in the title of the document when questions regard the method.

2. *Analyzing the import statements.* We take all the import statements in the source file and remove the ones not used by the target entity. Since we do not have any information from the AST, we identify the used imports by applying a naïve matching on the class name: If the class name is contained in the entity's body, we consider this import or we discard it otherwise. This approach can lead to false positives in case two classes have the same name, but they reside in different packages, and are used by the same entity. However, such situations rarely happen. Once the imports are identified, we tokenize each statement on the "." character, and by defining a set of unique tokens that become part of the query. For example, assuming we have the following import statements used by an entity:

```
import java.util.List;
import java.util.ArrayList;
```

The resulting set of tokens would be [*java, util, List, ArrayList*].

The Annotation Engine

The recommendation engine allows the creation of links between source code and documents. To this aim, we implemented an annotation engine to let developers put annotations in the code. We want to allow developers to collaborate by means of the crowd knowledge itself. Differently from the query engine, which provides automated query generation, the annotation engine implements the second aspect of the manual interaction in the SEAHAWK recommendation system.

There are two main purposes in the annotation engine: creating and parsing annotations. The annotation structure must be flexible. To be language independent, we wanted to achieve this flexibility by embedding annotations in multi-line comments. Doxygen[6] follows a similar approach to integrate documentation in, for example, C++ and Java code and in Blueprint [BDWK10] to link code examples to code. Both of them enclose meta-information between multi-line comment delimiters (*i.e.,*/* and */) and define fields by putting @ as prefix character.

SEAHAWK's approach gives users more flexibility. Developers can define custom delimiters (that need to match the target language syntax for comments), and to avoid conflicts with Doxygen or JavaDoc annotations, we decided to put an exclamation mark as last character for the opening delimiter (*e.g.,* in Java the opening delimiter would become /*! instead of /* while in XML it would become <!-! instead of <!-). Listing 3.1 presents an example of SEAHAWK's annotation.

---

[6]http://www.doxygen.org/

Listing 3.1. Example of SeaHawk's annotation

```
/*!
 * @documentId <Document's Id>
 * @title <Document's title>
 * @comment <Author's comment>
 * @author <Autor's name>
 * @creationTime <creation date>
 */
```

Whenever an annotation is created, SEAHAWK reports the id of the document, its title, a comment put by the developer (the author of the annotation), and the creation time. The id of the document identifies the target document to be suggested. The other fields are used to implement the basis of the support for collaboration. SEAHAWK does not explicitly provide collaborative functionalities, but relies on the fact that a versioning system (*e.g.,* Git, SVN) is used in the development phase. Putting annotations in the code is enough to keep track of the document suggested by developers, thus linking documents to a specific revision of the source code.

The whole collaborative process is embedded in the normal development phase: whenever a developer commits, the annotations are committed too. Whenever a developer updates the repository, the new annotations are updated together with the comment explaining the purpose of the linked document. The role of the *comment*, *author* and *creationTime* fields guarantees that annotations are unique. The *comment* field is mainly used to allow developer to communicate with each other through the annotation system.

The annotation engine provides also a notification system to keep track of the annotations already seen by developers. For that reason we use two different ways of parsing code:

1. We implemented our own parser for annotations.

2. We took advantage of Eclipse's partitioning system.

The partitioning system identifies code blocks (partitions) that match specific delimiters in the code editor (*e.g.,* comment, classes, methods *etc.*). We ask it to match the annotation's delimiter and notify in case of changes. Whenever a source file is opened or modified in the code editor, the partitioning system notifies the view showing the suggested documents and storing the annotations in the cache, thus tracking the annotations that the developers have already seen. The latter relies on our implementation of the parser that works in the background. Whenever a project is updated, it parses all the updated files of the project and extracts annotations. If the annotations are not present in the annotation cache, they are considered as new annotations to be notified to the developer.

3.1.4   The User Interface

In Section 3.1 we presented an overview of the user interface of SEAHAWK. In this section we present each UI element and the functionalities provided to developers.

Document Navigator View

The first view of SEAHAWK is the one implementing the manual interaction of the recommendation engine. Through this view, developers can compose a query, send it to the search engine,

and retrieve Stack Overflow's documents. Users are provided with a tree navigation system that allow them to explore single nodes (*i.e.,* questions or answers) of a single discussion. We reach the granularity of the code snippets in case they are available. By means of *drag and drop* (D&D) interactions, developers can drag a document or a code snippet into the code editor. Whenever a document is dropped in the editor, SEAHAWK shows a dialog (see Figure 3.3) where the user can put a comment to explain the link between the document and the code, and then it generates the annotation in the code editor.
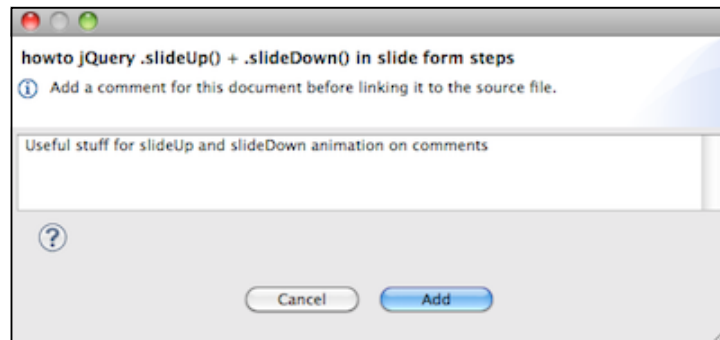
Figure 3.3. SeaHawk dialog for annotation's comment

Suggested Documents View

Figure 3.1 (2) depicts a tree view similar to the one previously presented (1). which is used by SEAHAWK to show document linked to the code. Instead of presenting documents retrieved from a query, this view tightly works with the annotation engine. Whenever an editor become active, the annotation engine parses the file, extracts all of SEAHAWK's annotations, and notifies the view. The set of documents linked by the annotations is then retrieved from the search engine and displayed. Differently from the document navigator view, the user cannot drag documents in the code editor to create annotations. Allowing this feature would create redundancies in the annotations for documents that are already present in the code editor. Through a contextual menu, users can modify the comment of an annotation or delete the annotation as well. Annotation data is accessible by a tool tip that appears on top of the document when the mouse pointer is over it.

Since there is no mechanism to ensure consistency in the annotations, a linked document could have been removed in the search engine. In this situation, the document is shown anyway but the message "[Not Available]" is put in front of the document's title and it becomes not traversable (see Figure 3.4).
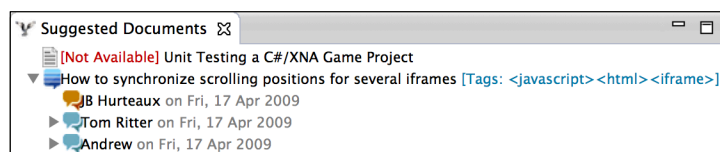
Figure 3.4. Document not available in SeaHawk's View.

Document's Contents View

When a document, or a document's node, is selected in one of the other views, the content is displayed in this view. To display the content, we use a custom layout and we take advantage of a web-browser widget embedded in the view. The web-browser widget allows the developer to navigate the links contained in the document and getting additional information. We use a Javascript library[7] to highlight the syntax of the text contained in the `<code>` tags, without having to care about the programming language. Questions are orange, the accepted answer is green and the other answers are blue.

Notification System

To rapidly spot new annotations in the project, we implemented notification system in the package explorer (Figure 3.1 (4)). Whenever a project is refreshed, the annotation engine parses the files and creates a list of annotations. Subsequently, it counts the number of annotations not seen and decorates the package explorer with the number of new annotations between square brackets. Whenever the developer opens one of the compilation units, the annotation engine parses the file and puts the annotations in the cache before the number shown in the package explorer is updated, thus reducing the count of the annotations.

3.2 A Use Case Scenario

By means of a simple scenario, we illustrate how SEAHAWK can help developers to solve programming problems by leveraging Stack Overflow from within the Eclipse IDE. Alice is required to build a simple *echo server* in Java. The echo server must handle one client at a time and it must terminate itself whenever a client sends the "quit" string. Alice opens up the Eclipse IDE, with the SEAHAWK plugin installed, and begins creating the class `EchoServer`. She starts by creating a socket by using the `Socket` class:

Listing 3.2. Initial Implementation of an Echo Server

```java
public class EchoServer{
    public static void main(String[] args){
        Socket server;
        server = new Socket("localhost",8000);
    }
}
```

Alice starts looking at the methods trying to find out a way to accept incoming connections. Since she does not find any method to accomplish this task, she invokes SEAHAWK through the contextual menu in the code editor.

SEAHAWK analyzes the existing code, builds a query, and retrieves a set of documents related to what is written in the `EchoServer` class (Figure 3.5). Among the documents, Alice finds out a question whose title is "Problems trying to implement Java Sockets". She reads the document and finds an accepted answer that proposes the implementation of a simple echo server. She realizes that the right class to be used instead of `Socket` is `ServerSocket`. Thanks to the document navigation system of SEAHAWK, she locates the code snippet and drags it into the code editor, importing it (Figure 3.5). Subsequently, Alice can start modifying the code in the editor to

---

[7]`http://code.google.com/p/google-code-prettify/`
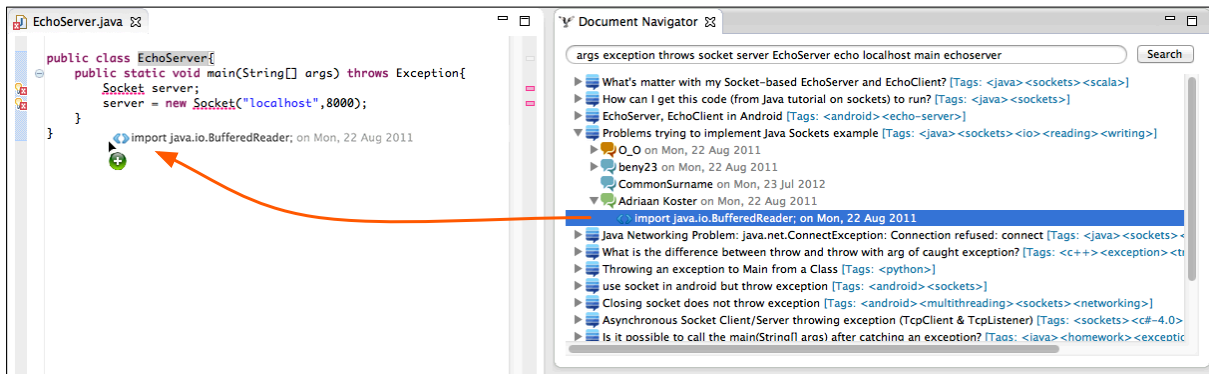
Figure 3.5. Alice imports the code snippet in the code editor.
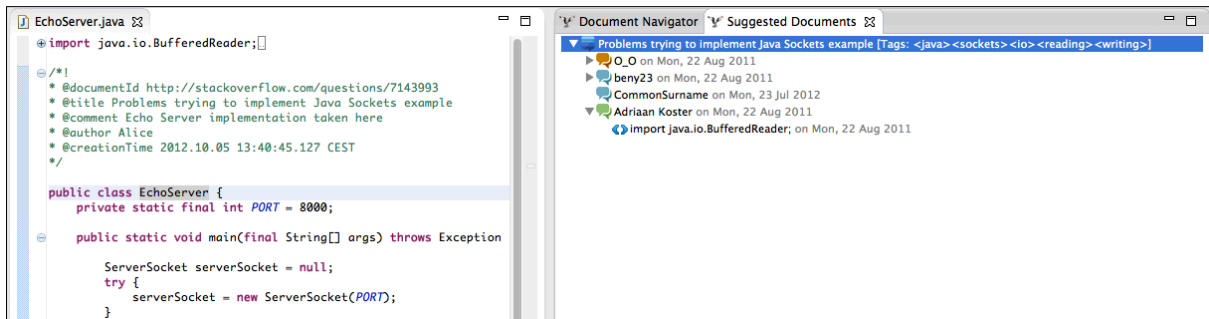
Figure 3.6. Notification of the linked document in the Suggested Documents View.

achieve the desired outcome. With minor modifications she adapts the imported snippet and makes the server able to be terminated when receiving a *quit* string from a connected client. In the end, Alice wants to bookmark the original solution directly in the code. Thus, she drags the document in the editor. SEAHAWK creates an annotation to link this specific Stack Overflow document and asks her to put a comment by means of a dialog box. Alice types the comment and confirms the creation of the annotation that becomes visible in the code editor. By doing so, every other person opening the file with the SEAHAWK plugin installed will be notified about the bookmark in an ad-hoc view (Figure 3.6).

3.3 Evaluation

The previous scenario, while being a real example of SEAHAWK in action, does not provide any evidence in terms of usefulness and usability. To address the question whether SEAHAWK can actually help developers in their tasks, we present an evaluation composed of three different experiments.

3.3.1 Experiment I: Java Programming Exercises

We want to assess to what extent SEAHAWK can deal with plain text. We use a set of exercises taken from Java programming courses[8][9] to evaluate the relevance of the documents retrieved from Stack Overflow by extracting keywords from the text of the exercises.

SEAHAWK is not designed to directly deal with plain text taken from exercises. We thus had to recreate the right conditions to allow SEAHAWK to extract keywords from the text of the exercises: Since it needs at least a Java entity, we manually create a class stub with a name that summarizes the topic, and put the entire text of the exercise as a comment before, or inside, the class body, as depicted in Listing 3.3.

Listing 3.3. Example of Java exercise prepared for the test

```
/* Write a class that implements the CharSequence inter-
face found in the java.lang package. Your implementation
should return the string backwards. Select one of the
sentences from this book to use as the data. Write a
small main method to test your class. Make sure to call
all four methods.*/

public class CharSequenceImpl { }
```

With this approach we tested SEAHAWK on 35 exercises. For every exercise, we created a class similar to the one presented in the previous example, we generated keywords from it, and we queried the search engine. From the result returned, we considered the first 15 documents. We decided to use such a threshold because it is the smallest number of documents retrieved by Stack Overflow. We manually inspected and evaluated every document. With the term "relevant", we mean that the discussion can lead to a solution of the exercise either through the discussed topic or the code snippets. For example, the exercise in Listing 3.3 could lead to discussions tackling the implementation of `CharSequence` interface that could be partially relevant as well. For this reason, a binary notion of relevance is not enough. Thus, we defined *five* levels of relevance, ranging from 0 to 4. To have a numerical assessment of this experiment, we refer to the *normalized discounted cumulative gain* (NDCG), which is generally used to evaluate ranked retrieval results from search engines, using a multi-valued notion of relevance [MRS08]:

$$NDCG(Q,k) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)} - 1}{log_2(1+m)} \tag{3.1}$$

k is the size of the result set; Q is the set of queries performed; $R(j,d)$ is the relevance score gave to document d for query j; and Z_{kj} is the normalization factor calculated such that NDCG is equal to 1.0 in the ideal scenario (*i.e.,* all the documents have the maximum level of relevance). In our experiment, $k = 15$, $|Q| = 35$ and the normalization factor we calculated is $Z_{kj} \sim 0.011$.

Experiment I: Results

The result we obtained from the NDCG index is 9.07%, thus meaning that one in ten of the documents retrieved was relevant to the Java exercises we used. In Table 3.1 we present only a subset of the results, the ones we discuss afterwards; for the results for all the 35 exercises

---

[8]http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/java.html
[9]http://codingbat.com/java

Table 3.1. Experiment I Results *(0 = Not Relevant, 4 = Highly Relevant).*

Exercise	D 1	D 2	D 3	D 4	D 5	D 6	D 7	D 8	D 9	D 10	D 11	D 12	D 13	D 14	D 15
Electrical Resistance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fibonacci	2	3	3	0	0	0	2	3	3	3	3	3	3	3	4
Metropolis	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Natural Merge Sort	3	3	4	0	3	0	4	3	0	0	0	3	2	2	2
Roulette Strategy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sudoku Solver	3	4	3	2	0	0	0	0	0	0	0	0	0	0	1
Roulette Strategy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

we refer to [Pon12]. Although the *NDCG* value is low, there are some considerations to make. Our approach fails on very simple exercises: Exercises like *ElectricalResistance* or *WindSpeed* (where the student is asked to write a simple function to calculate the value of the resistance and the wind speed value) provide very little information, thus the documents returned were unrelated. Sometimes the topic of the exercise was a subset of a more complex one. For instance, *RouletteStrategy* requires to calculate the number of turns required to lose all by betting only on red or black at roulette. The retrieved documents were discussing the same topic but at a higher level of difficulty (*e.g.,* machine learning approach), making them not relevant.

A reason to justify the irrelevance of the documents could reside in the absence of information in the Stack Overflow's crowd knowledge. Even though Stack Overflow archives many discussions on homework, the requirements of the exercises were not specific enough. Just in one case the exercise was in one of the document returned. Moreover, exercises requiring the implementation of data-classes (*e.g., Metropolis*) returned unrelated documents. However, when an exercise tackles a well known topic (*i.e., Fibonacci, NaturalMergeSort* and *SudokuSolver*), the relevance of the documents increases: We were able to find solutions or even the full implementations in pseudocode, Java or similar languages that could be easily adapted and used to solve the exercise.

Table 3.2. Experiment II Selected Methods

Name	Method	Class (method)
M1	Interface	`EnumerationImpl (hasMoreElements)`
M2	Interface	`IntegerList (addAll)`
M3	Interface	`MarkerInitActionDelegate (selectionChanged)`
M4	Interface	`PreferencePaneMbox (createFieldEditors)`
M5	Interface	`REmailLightweightDecorator (decorate)`
M6	Regular	`CopyPaste (copy)`
M7	Regular	`MarkerInitActionDelegate (prepareSQLite)`
M8	Regular	`Parser (parseFunction)`
M9	Regular	`SpreadsheetReader (loadFile)`
M10	Regular	`SpreadsheetReader (removeDoubleQuotes)`

3.3.2 Experiment II: Method Stubs

In this experiment we wanted to assess the impact of SEAHAWK when dealing with method stubs. The scenario concerns a developer who starts to implement a method, does not know how to

Table 3.3. Experiment II Results *(0 = Not Relevant, 4 = Highly Relevant)*.

Method Name	D 1	D 2	D 3	D 4	D 5	D 6	D 7	D 8	D 9	D 10	D 11	D 12	D 13	D 14	D 15
M1	0	0	3	4	0	0	4	0	0	0	0	0	0	0	0
M2	2	2	2	3	1	0	2	0	0	1	0	0	1	2	2
M3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M4	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0
M5	4	2	4	0	0	0	0	0	0	0	0	0	0	0	0
M6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M9	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
M10	0	2	1	1	0	1	0	0	0	1	0	0	0	0	0

proceed, and asks SEAHAWK for help. We selected eight different methods from student projects, and two exercises taken from a Java programming course, reaching a total of ten methods. As reported in Table 3.2, half of the methods were implementing part of a Java interface, the remaining half were regular methods. In doing so, we want to see if the behavior, in case of interfaces, changes with respect to regular methods. We removed the bodies from the methods to obtain stubs, leaving everything else unchanged.

Experiment II: Results

In Table 3.3 we report the results for the stubs we tested. When dealing with method stubs, SEAHAWK performs well with interface methods, but not with regular methods. Our conjecture is on the one hand that interface methods are less volatile than general methods, and on the other hand there is a higher probability that they have been discussed on Stack Overflow because interfaces are used by potentially many clients. For instance, for the *REmailLightweightDecorator (decorate)* method, SEAHAWK retrieves useful documents with the right code examples to implement a fully working decorator. Regular methods, on the other hand, have a lower probability to be discussed on Stack Overflow because they pertain to the specific domain of a system. The exception (see the results for the last two stubs) is when the "theme" of a method signature is of general interest (*e.g.,* loading a file, removing quotes). In the case of *Spreadsheet-tReader (removeDoubleQuotes)* the retrieved documents lead to a better solution than the one implemented in the original method. This is an argument in favor of having appropriate names for methods (*i.e.,* if removeDoubleQuotes would have been called rDQ SEAHAWK would have performed poorly).

3.3.3　Experiment III: Method Bodies

In the third experiment we want to assess the behavior of SEAHAWK when dealing with fully implemented methods. The scenario pertains to program comprehension: A developer invokes SEAHAWK to understand an existing and fully implemented method. As reported in Table 3.4, we selected seven fully implemented methods, one of which was implementing an interface. We left all methods unchanged, including comments. We wanted to see if the documents retrieved by SEAHAWK would help a developer achieving the same goal of the original implementation, thus helping her in getting a better understanding of the code.

Table 3.4. Experiment III Selected Methods

Name	Method	Class (method)
M1	Interface	`REmailLightweightDecorator (decorate)`
M2	Regular	`MapEditor (buildMenu)`
M3	Regular	`MapEditor (buildWest)`
M4	Regular	`MarkerInitActionDelegate (prepareSQLite)`
M5	Regular	`Parser (parseFunction)`
M6	Regular	`SpreadsheetReader (loadFile)`
M7	Regular	`SpreadsheetReader (removeDoubleQuotes)`

Table 3.5. Experiment III Results *(0 = Not Relevant, 4 = Highly Relevant)*.

Method Name	D 1	D 2	D 3	D 4	D 5	D 6	D 7	D 8	D 9	D 10	D 11	D 12	D 13	D 14	D 15
M1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M2	1	0	4	4	4	4	0	0	0	0	0	0	1	4	1
M3	2	0	2	0	2	2	1	3	3	2	0	0	0	2	0
M4	2	3	0	2	0	2	0	0	0	0	0	0	0	0	0
M5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M6	2	1	2	0	0	0	4	4	2	2	0	0	0	4	0
M7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Experiment III: Results

In Table 3.5 we report the results for the tested methods. SEAHAWK performs well in this scenario. On the one hand this is influenced by the higher amount of useful information that can be fed to the query engine. On the other hand the fact that methods have (or should have) a single responsibility limits the scope in a positive way, leading to more useful retrieved documents. Moreover, the library or framework used in the implementation of a method can refine the scope of the research. For example, many useful documents were retrieved for the methods *buildMenu* and *buildWest* of the class *MapEditor*. In this case, both methods largely used the *Swing* framework, restricting the search scope. The contrary happens when the topic of a method is too general. Moreover, as in the case of *parseFunction* of the class *Parser*, SEAHAWK is not able to retrieve relevant documents when the implementation of a method totally relies on code belonging to the project (*i.e.,* no particular standard libraries or well-known frameworks are used). A last scenario in which SEAHAWK performs poorly is when it is confronted to badly implemented methods. Tying this back to the program comprehension scenario: Sometimes understanding a method is not easy because the method is badly implemented. This also applies for cases where the signature of a method does not correspond to its implementation.

3.4 Conclusions

We presented an approach completely based on standard information retrieval approaches. We implemented our approach in SEAHAWK, a plugin for the Eclipse IDE which let users interact with Stack Overflow documents, to import code snippets and create links between documents and source code by means of language-independent annotations so that developers can use annotations to take advantage of the versioning system to collaborate and suggest documents to

teammates. SEAHAWK automatically generates queries from code entities, by dealing with import statements and uncompilable code to extract keywords from Java code entities. Finally, we presented an evaluation of SEAHAWK and a discussion of the results we obtained.

Reflections

This chapter presented a reductionist way of treating the information provided by both code and artifacts, reducing them to words, without preserving any information about the contents. The whole pipeline of SEAHAWK is designed by ignoring the multi-dimensionality of the information. Indeed, the process takes a code context, projects it to a textual query, and retrieves discussions according to the query by leveraging a standard IR approach based on term frequencies. As shown in the evaluation part, some of the side effects are clearly visible in the evaluation of this approach, where it performs well only with known topic (*e.g.,* fibonacci).

We believe that recommendation systems like SEAHAWK have the potential to help developers in program comprehension and software development activities. However, to be effective, the engines of RSSEs like SEAHAWK need to go beyond the pure textual representation of the artifacts, and use analyses which consider the different types of information provided by the artifacts (*e.g.,* code information, meta data) as first class entities.

In the next chapter we follow this path, and we abandon the idea that off-the-shelf information retrieval analysis tools might be used to build effective RSSEs. Instead, we focus on building customized tools and models considering several aspects of the information.

4

Turning the IDE into a Self-confident Programming Assistant

In the previous chapter we discussed an approach for RSSEs using off-the-shelf tools for information retrieval, considering words, in the narrative sense, as the minimal unit to describe information in development artifacts and source code.

In this chapter we tackle two aspects of RSSEs. The first aspect concerns the data analysis, where we aim at analyzing the semantic link between source code in the IDE and a development artifact by weighting different aspects of the information.

The second aspect concerns the interaction between RSSEs and developers, the (almost) total absence of pro-activeness of the current approaches, which leads to manual intervention by the developers.

We describe PROMPTER, a fully automated RSSE that retrieves and recommends, through push notifications, relevant Stack Overflow discussions to the developer. PROMPTER makes the IDE a programming prompter that silently observes and analyzes the code context in the IDE, searches for Stack Overflow discussions on the Web, evaluates their relevance by taking into consideration several aspects of the information like *code aspects* (*e.g.,* code clones, type matching), *conceptual aspects* (*e.g.,* textual similarity), and Stack Overflow *community aspects* (*e.g.,* user reputation) to decide when to suggest discussions.

Structure of the chapter

In Section 4.1 we highlight a limitation of the interaction of current RSSEs. In Section 4.2 we present our approach and its implementation in PROMPTER. In Section 4.3 we present the results of the ranking model evaluation of PROMPTER (*Study I*). In Section 4.4 we present the evaluation of PROMPTER with developers (*Study II*), while the replication of *Study I* one year later is described in Section 4.5. We discuss threats to validity in Section 4.6, and Section 4.7 concludes the chapter.

4.1 On the pro-activeness of RSSEs

Seminal tools, such as EROSE [ZWDZ04], HIPIKAT [CM03] and DEEPINTELLISENSE [HB08], suggest project artifacts in the IDE to provide developers with additional information on specific parts of the system. They come however with a caveat: the developer must proactively invoke them, and, once invoked, they continuously display information, which may defeat their purpose,

as they augment the complexity of what is displayed in the IDE. *Ideally, a recommender system should behave like a prompter in a theatre: ready to provide suggestions whenever the actor needs them, and ready to autonomously give suggestions if it feels something is going wrong.*

The interaction between the theatre prompter and the actor is similar to the interaction between two developers doing pair programming, working side by side to write code. These developers have different roles, *i.e.,* the driver, who is in charge of writing code, and the observer, who observes the work of the driver [Wil01], tries to understand the context, and, if she has enough confidence, interrupts the driver by giving suggestions. In addition, the driver can consult the observer whenever she needs it, making the observer the programming prompter of the programming actor.

By following a metaphor similar to the interaction between the theatre prompter and the actor, we propose PROMPTER.

4.2   Prompter

We first introduce PROMPTER's user interface and architecture. We then discuss and describe the approach implemented in PROMPTER, especially focusing on the ranking model, its features, and the techniques that enable its self-confidence.
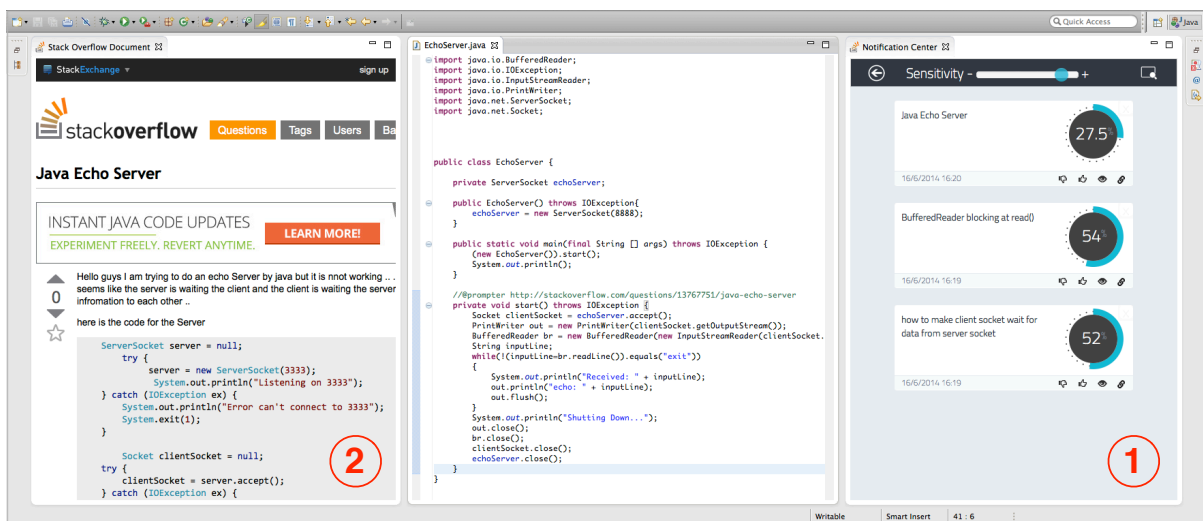
Figure 4.1. The Prompter User Interface.

4.2.1   User Interface

Figure 4.1 shows the user interface of PROMPTER. It provides two views through which the user can (i) receive and track notifications, and (ii) read the suggested Stack Overflow discussions. The notification center (1) is the main view of PROMPTER and it is used to notify the developer whenever a relevant result is available. When PROMPTER considers a discussion as relevant for the current context (*i.e.,* for the code opened in the IDE), it opens the notification center and plays a sound. If a Stack Overflow discussion is notified more than once, it is pushed to the top of the list for visibility.
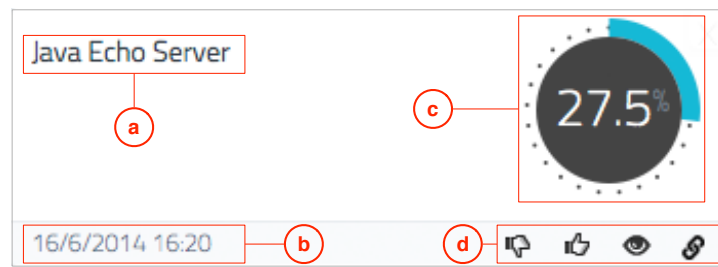
Figure 4.2. Prompter notification details.

Figure 4.2 shows an example of notification. The developer is provided with some information regarding (a) the title of the Stack Overflow discussion, (b) the notification date and time, and (c) the confidence level of PROMPTER on the Stack Overflow discussion against the related code context,

Moreover, PROMPTER provides feedback, tracking and linking functionalities in the bottom-right corner. By clicking on the thumb up (down) icon, the developer can rate the discussion as useful (useless) with respect to the coding activity she is performing in the IDE. Currently, the feedback provided by PROMPTER's users are only stored in a database for possible future usages, including: (i) a better tuning of the PROMPTER ranking model, and (ii) the possibility to gather indications on the goodness of the PROMPTER's recommendations during case studies.

The other icons on the notification allow the developer of backtracking the code entity associated with a specific notification (eye icon), or to link the suggested discussion to its code entity (chain icon). If the developer clicks on the former, PROMPTER opens up a code editor and highlights the portion of code related to the notification. If the developer clicks on the latter, a simple annotation reporting the URL of the discussions is created in the code in form of a comment.

Whenever a developer clicks on a notification, a Stack Overflow document view (point 2 in Figure 4.1) is opened, which shows the contents of the Stack Overflow discussion. At the top of the notification center, the developer can change the sensitivity of the notification system (point 2 in Figure 4.3(a)): by sliding to the right PROMPTER is more talkative and produces more notifications, by sliding to the left it becomes more taciturn and requires a higher level of confidence to notify the developer. Moreover, by clicking on the arrow in the top-left corner (point 1 in Figure 4.3(a)), the developer can access the full result set of Stack Overflow discussions related to the last notification (*i.e.,* the other Stack Overflow discussions retrieved by PROMPTER for the same code context but not pushed as having a lower confidence level).

Explicit Query Writing

Sometimes PROMPTER is not able to point out the right Stack Overflow discussion or probably it has not enough information to generate a notification. A similar situation could happen at the very beginning of the development, when there are just few lines of code (*e.g.,* a class stub) written in the IDE's code editor. To overcome these situations, we implemented an additional manual interaction where we provide the developer with the capability to perform manual searches. Whenever the developer wants to search for Stack Overflow discussions on her own, she can click on the manual search button at the top right corner (point 3 in Figure 4.3(a)). The notification center disappears and a manual search bar becomes available (Figure 4.3(b)). There, the developer can manually type a query (point 4 in Figure 4.3(b)) and search for Stack Overflow discussions. As it will be clearer later, the first version of PROMPTER did not implement the

(a) PROMPTER sensitivity bar

(b) PROMPTER manual search bar
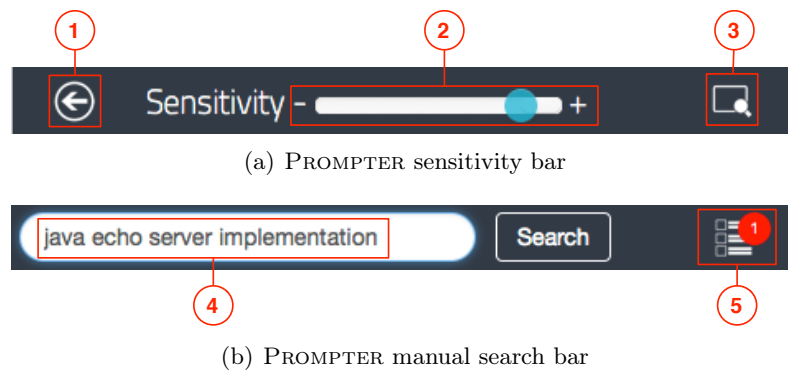
Figure 4.3. Notification center bars of Prompter.

search bar for manual query formulation. The need for such a feature has been highlighted by participants of our second study (see Section 4.4).

The results are presented in form of notification, where each one presents a confidence value according to the code context obtained from the code editor on top, that is, the active code editor. While the developer is interacting with the manual search view, she can continue modifying and writing code. If PROMPTER pushes a discussion in the meanwhile, the developer is notified anyway: a counter of the unseen notifications will pop up on top of the notification center icon—point 5 in Figure 4.3(b)—and it resets as soon as the developer accesses the notification center by clicking on the icon.

Explicit Invocation

A prompter in a theatre not only prompts the right sentence to the actors on the stage, but also provides support on demand. Indeed, an actor can always ask the prompter for a cue in order to go on with the show. In PROMPTER we implemented the same interaction: the developer can always ask PROMPTER to perform a search on a specific code entity (*i.e.,* method or class), by accessing the contextual menu in the code editor, or on the package explorer. In the first case, PROMPTER searches discussions for the code entity identified by cursor in the editor, while in the second case it searches according to the code entity selected (see Figure 4.4).

4.2.2   Architecture and Control Flow

Figure 4.5 depicts the interactions among all the components of PROMPTER when it searches, evaluates, and triggers a new notification to the developer.

PROMPTER tracks code contexts every time a change in the source code occurs. The extracted code context—code elements to formulate the query—is sent to the *Query Generation Service*, which formulates a query starting from the code context. It extracts a query and, according to a set of parameters described later, determines if a new search can be triggered. This information is sent back, with the query and the context, to the plug-in. Since the query is the basis of every search triggered by PROMPTER, the plug-in also considers the query when deciding to trigger a new search. PROMPTER submits a new search only if the query differs from the last one. The query and code context are sent to the *Search Service*, which acts as a proxy between the plug-in, the search engines to which the query is sent, and the *Stack Overflow API*. The query is sent to search engines (Google, Bing) to perform a Web search on the Stack Overflow website. The first 100 Stack Overflow discussions retrieved by each of the two search engines (a retrieved URL
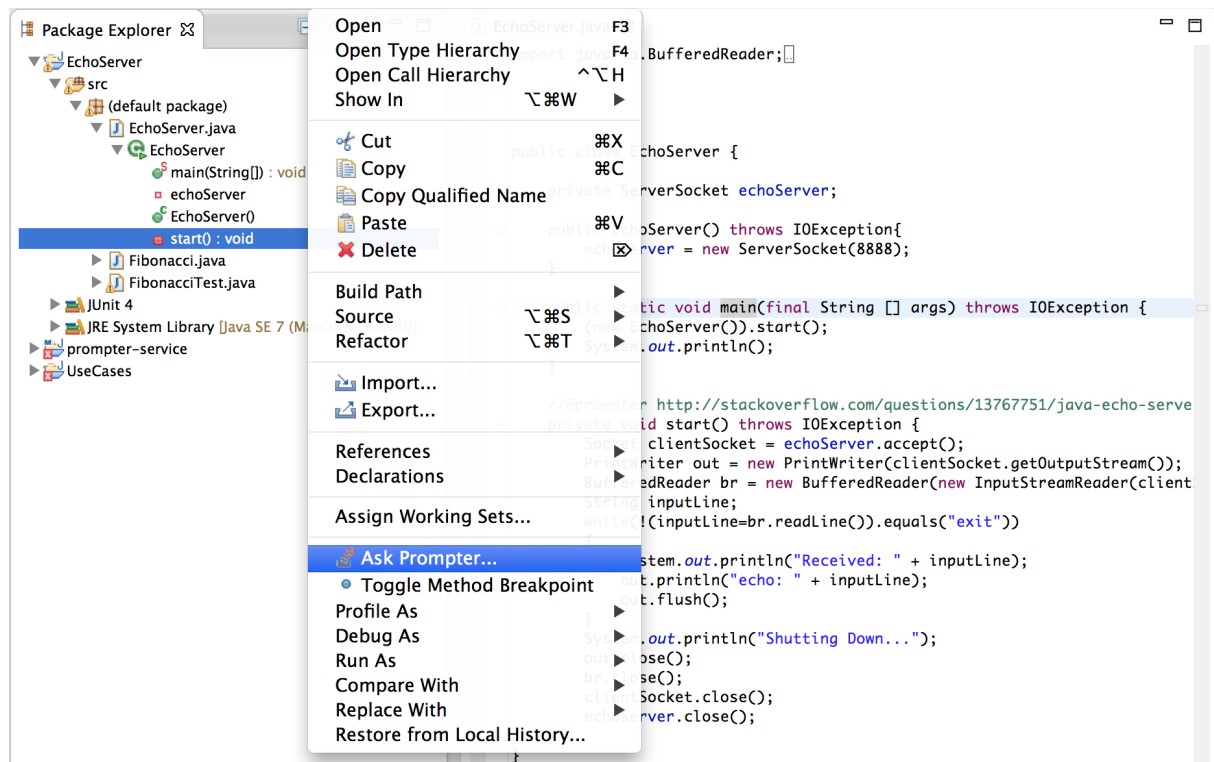
Figure 4.4. Explicit invocation of Prompter via contextual menu.

refers to a question from Stack Overflow if it matches the form *stackoverflow.com/questions/*
<id>/<title>) are collected and merged in a single set, where duplicates are removed. Note
that this set of retrieved Stack Overflow's discussions is not ranked in any way (*i.e.,* we ignore
the ranking made by the search engine) since PROMPTER will evaluate the relevance of each of
these discussions to the code context by using its own ranking model. The search service uses
the Stack Overflow question ID to retrieve the discussion via the Stack Overflow API. Every
discussion is ranked according to the *Ranking Model* (see Section 4.2.3), that takes into account
the developer's code context. The ranked list of URLs, along with the related confidence values
given by the model, is sent back to PROMPTER. The plug-in takes the top-ranked discussion and
evaluates its confidence level against the threshold set by the developer. In case the confidence
surpasses the threshold, the top-ranked discussion is notified to the developer.

4.2.3   Retrieval Approach

Our approach is capable of (i) connecting different aspects of the code written by developers to
the information contained either in the text or in the code of Stack Overflow discussions, and (ii)
taking into consideration information about the quality of the discussions that Stack Overflow
has available (*e.g.,* user reputation and questions/answers score). In Chapter 3 we only used text
similarity to retrieve Stack Overflow discussions related to the actual code. This led to errors in
the identification of relevant discussions.
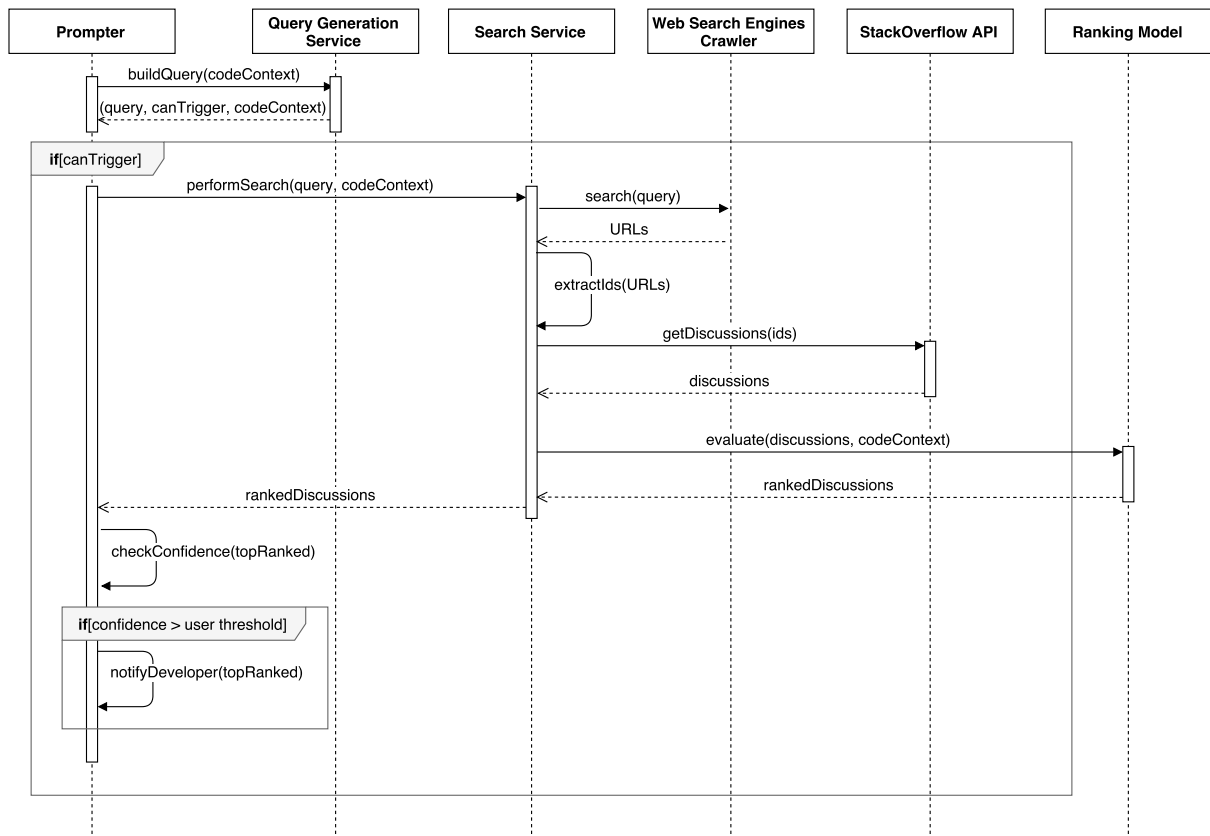
Figure 4.5. The UML sequence diagram representing the background search phase performed by Prompter whenever the developer modifies a code entity.

Tracking Code Contexts in the IDE

PROMPTER is meant to be a silent observer "looking" at what a developer writes, with the aim of suggesting relevant Stack Overflow discussions. Whenever the developer types, PROMPTER waits until the developer stops writing for at least s seconds[1], identifies the current code element (*i.e.,* method or class) that has been modified, and extracts the current context, which consists of: (i) a fully qualified name identifying the code element[2]; (ii) the source code of the modified element (*i.e.,* class or method); (iii) the types of the used API, taking into account only types outside the analyzed Eclipse project (*i.e.,* declared in external libraries or in the JDK); and (iv) the names of methods invoked in the API, again considering only external libraries and JDK only. The extracted information (*i.e.,* the context) is sent to the *Query Generation Service* (see Figure 4.5) to generate a query.

Generating Queries From Code Context

Since we want to automatize the triggering of searches for discussions on Stack Overflow, we have to devise a strategy to build a query describing the current code context in the IDE. A naïve approach (see Chapter 3) is to treat the code as a bag of words by: (i) splitting identifiers

---

[1]The s threshold is customizable. By default it is set to 5.

[2]Classes are identified by the unique id `projectName.packageName.ClassName`, methods are identified by `projectName.packageName.ClassName.methodSignature`

and removing stop words; (ii) ranking the obtained terms according to their frequency; and (iii) selecting the top-n most frequent terms. Using only the frequency value is not highly discriminating in selecting terms that appropriately describe the context: Words like *run* or *exception*, even if very frequent in source code, have a too general meaning in programming to discriminate the programming context. Our solution is to also consider the entropy of a given term t in Stack Overflow—previously used in the context of quality assessment and reformulation of queries for text retrieval in software engineering [HBO$^+$12a, HBO$^+$12b, HBM$^+$13]—and computed as:

$$E_t = -\sum_{d \in D_t} p(d) \cdot \log_\mu p(d) \tag{4.1}$$

where D_t is the set of discussions in Stack Overflow containing the term t, μ is the number of discussions in Stack Overflow, and $p(d)$ represents the probability that the random variable (term) t is in the state (discussion) d. Such a probability is computed as the ratio between the number of occurrences of the term t in the discussion d over the total number of occurrences of the term t in all the discussions in Stack Overflow. The entropy has a value in the interval of $[0, 1]$. The higher the value, the lower the discriminating power of the term. We computed the entropy of all 105,439 terms present in Stack Overflow discussions by using the data dump of June 2013[3]. Frequent terms exhibit high levels of entropy (*e.g.,* for *run* the entropy was 0.75) compared to less frequent and more discriminative terms (*e.g.,* for *swt* the entropy was 0.25). Therefore, term entropy can be used to lower the prominence of frequent terms that do not sufficiently discriminate the context.

It is important to point out that the interpretation of term entropy is more similar to the interpretation of Gibbs' entropy from thermodynamics than to the Shannon's entropy [Sha48]. That is, words that are highly diffused across documents have high entropy, much alike particles in a gas, whereas words occurring only in few documents have a low entropy, much alike particles in a solid. Our definition of entropy may still be interpreted as a Shannon entropy, similarly to what done by Hassan [Has09] to change entropy. That is, if a word is scattered in many files, you need more bits to keep track of where it is located (*e.g.,* the inverted index representation would allocate more memory for that word) than if it appears in few documents.

Last, but not least, it is important to point out that, while E_t converges to a *idf* (*i.e.,* Inverse Document Frequency) when a term is diffused, strictly speaking the definition of E_t is different from the *idf* definition. Indeed, previous studies that compared the *idf* and the entropy, concluded that "*despite the - log(P) form of the traditional IDF measure, any strong relationship between it and the ideas of Shannon's information theory is elusive*" [Rob04].

The *Query Generation Service* ranks the terms in the context based on a term quality index (**TQI**):

$$\mathbf{TQI}_t = \nu_t \cdot (1 - E_t) \tag{4.2}$$

where t is the term, ν_t is frequency in the context, and E_t is its entropy value measured as described before.

Once the ranking is complete, the *Query Generation Service* selects the top n terms to devise the query, plus the word *java*. The query can exceed n terms in case two or more terms exhibit the same **TQI** value. To better understand this process, we show an example of query creation. Listing 4.1 shows a Java method from which the *Query Service* has to extract a query. The method is making use of a library applying the *Snowball Stemmer*[4] on a set of tokens. By treating

---

[3] http://www.clearbits.net/torrents/2141-jun-2013
[4] http://snowball.tartarus.org/

the code as bag of words, we tokenize the text on white spaces, split on case-change, symbols and numbers, lower the case, and remove English stop-words and Java keywords. Table 4.1 shows the resulting tokens with the respective frequency and entropy value. Tokens in bold are the one selected for the query.

Listing 4.1. Example code entity from which the Query Service extracts a query

```java
@Override
public List<String> filter(final List<String> tokens) {
    final List<String> stemmed = new ArrayList<String>();
    for(final String t : tokens){
        SnowballStemmer stemmer = new englishStemmer();
        stemmer.setCurrent(t);
        stemmer.stem();
        stemmed.add(stemmer.getCurrent());
    }
    return stemmed;
}
```

We can notice how the entropy acts as dumping factor for the frequency, making high entropy value terms lose power. An example is the term *tokens* that has more priority than the term *list* even though it has half the frequency values of the other term. However, the term entropy approach has one drawback. We observed that terms with a very low entropy (thus good candidates to be part of a query) may be terms containing typos (*e.g.,* for *override* the entropy was 0.63, and for *overide* it was 0.05). They are present in very few Stack Overflow discussions and thus have a low entropy. To overcome this problem, before selecting the n terms to create the query, we use the *Levenshtein distance* [Lev66] to check for terms with a very high textual similarity. If we detect two terms (say t_i and t_j) having *Levenshtein distance* $= 1$, the term having the lower frequency in the context (say t_i) is discarded and considered as a likely typo, and its frequency is added to the frequency of t_j. If the two terms have the same frequency, we discard the lower entropy term as a likely typo.

Table 4.1. Selected terms for the code entity in Listing 4.1.

Term	Frequency	Entropy	TQI	Term	Frequency	Entropy	TQI
stemmer	**6**	**0.15**	**5.10**	english	1	0.51	0.49
stemmed	**3**	**0.15**	**2.55**	filter	1	0.58	0.42
tokens	**2**	**0.45**	**1.10**	array	1	0.72	0.28
list	**4**	**0.74**	**1.04**	set	1	0.80	0.20
snowball	**1**	**0.11**	**0.89**	add	1	0.84	0.16
stem	1	0.25	0.75				

4.2.4 Prompter Ranking Model

The goal of the ranking model is to rank the retrieved Stack Overflow discussions, and assign them a value that measures their relevance to the query. It relies on 8 different features that capture relations between Stack Overflow discussions and source code.

Textual Similarity: The similarity of the code in the IDE to the textual part of a Stack Overflow discussion without code samples. The goal is to assess the similarity between the topics of

the code and the topics of the discussion. We use Apache Lucene to create the index and preprocess the contents, by removing English stop words and Java language keywords, by splitting compound identifiers/token based on case change and presence of digits, and by applying the Snowball stemming. Finally, we compute the cosine similarity among the *tf-idf* vectors[MRS08].

Code Similarity: The percentage of lines of code in the IDE that are cloned in the Stack Overflow discussion. We use DuDe [WM05], a fast and lightweight line-based textual clone detector, to identify cloned statements among code and documents.

API Types Similarity: The percentage of API types used in the code that are also present in the Stack Overflow discussion. These are types that are not declared in the project, but in external libraries or in the JDK. The higher the usage of the same types in both discussions and code, the more the potential usefulness of the discussions. To identify the API types, we parse every code sample in the discussion with the Eclipse JDT parser. We are able to resolve types among different samples in the discussion as long as the fully qualified name (*e.g.,* imports) of the type is used in one them, or if the identified type is part of the standard JDK. In case of unresolved types, we match the identified simple name of the class with the simple name of the types used in the code.

API Methods Similarity: The percentage of API method invocations in the code present in the Stack Overflow discussion. Higher values suggest a similarity in API usage. We use the Eclipse JDT parser to identify method invocations that respect the Java grammar even if the type is not resolved. Since we can only identify the name and number of parameters without any signature, we only consider the name of the invoked method, which helps matching overloaded methods.

Question Score: The quality of the score of the question in the Stack Overflow discussion. Since the score is not bounded, we normalize the value in the range [0,1] using a sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{(\bar{x} - x)}} \qquad (4.3)$$

where x is the score and $\bar{x}$ is the average of the scores of all the questions in Stack Overflow according to the data dump of June 2013. This index indicates the quality of the question, according to the Stack Overflow community.

Accepted Answer Score: The quality of the score of the accepted answers in the Stack Overflow discussion. In case no accepted answer is present, the score is set to zero. The score is normalized like the question score, using the related average score. This index indicates the quality of the accepted answer, according to the Stack Overflow community.

User Reputation: The level of reputation of the person who posted the question. The value is normalized like the two previous features, using the related average value. Differently from the two previous indexes, this index evaluates the reliability of the person who asked the question on the Stack Overflow community.

Tags Similarity: The percentage of tags covered by keywords extracted from imports. Tags gets split on number and symbols to remove versions, and tokens are matched against the tokens obtained by splitting imports on dots and lowering the case, without considering the case change. For example the tag *apache-httpclient-4.x* becomes *[apache, httpclient]* and

the import statement *org.apache.http.client.HttpClient* becomes *[org,* **apache***, http, client,* **httpclient***]* . In this case, there is a 100% coverage of the tags. The idea is to identify the topics or libraries used in the discussions even if there is no code in the discussion.

Ranking model definition

These 8 features are linearly combined to define the ranking model. Each feature is assigned a weight that defines the impact of this specific feature on the overall score:

$$S = \sum_{i=1}^{n} w_i \cdot f_i \qquad \text{having} \qquad \sum_{i=1}^{n} w_i = 1 \qquad (4.4)$$

where $f_i \in [0,1]$ is a feature value and $w_i \in [0,1]$ is the assigned weight. In doing so, the score S ranges in the interval [0,1] as well. The next step is to calibrate the weights of the PROMPTER features in equation 4.4.

Calibration of the Ranking Model

We need a way to objectively measure the recommendation accuracy of a given PROMPTER configuration, a "gold standard" composed of code contexts each of which is linked to a set of "relevant" (useful to a developer working on a specific context) Stack Overflow discussions. With such a dataset, the recommendation accuracy of a specific PROMPTER configuration can be measured as the number of code contexts for which PROMPTER is able to retrieve a relevant Stack Overflow discussion in the first position. Since PROMPTER recommends only the top ranked document, we only need to evaluate the accuracy for that document.

To identify the best configuration we used an exhaustive combinatorial search. We measured the performance of all configurations obtained varying each weight between 0 and 1 with step size 0.01 where the weights total 1, as defined in equation (4.4). Although time-consuming, this avoids that a possible sub-optimal calibration affects the study results. If a faster calibration were required, a search-based approach could be used, as done for information retrieval [LAZCH13, PDO+13] or clone detectors [WHJK13].

Such a calibration process might be highly biased by the choice of the dataset, *i.e.,* of the set of code contexts. We mitigate this threat by maximizing the dataset diversity, and its representativeness of various programming problems developers could encounter: We collected a set of problems encountered by industrial developers and Master and Bachelor students during laboratory and project activities. For each problem, we asked the subjects to provide a description and the code they produced before requesting or searching for solutions.

We collected 74 code contexts, 48 from academic contexts and 26 from industry. We randomly sampled half of them (37) for the calibration, and used the remaining 37 for the first evaluation of PROMPTER described in Section 4.3. For each of the 37 contexts used for the calibration, we browsed Stack Overflow with the aim of finding pertinent, helpful, discussions. More than one discussion could be identified in this phase. The set of relevant documents manually identified represents our "gold standard" to measure the suggestion accuracy of a specific PROMPTER's configuration.

Table 4.2 reports the configuration that provides the best recommendation accuracy. The indices with value **0.00** have been discarded from the model after completing the calibration. We have used this configuration for the two evaluation studies. Having 74 code contexts available (along with manually identified relevant documents), and having calibrated the model using only 37 of them, we could have used the other 37 contexts as a test set to automatically evaluate the

Table 4.2. Prompter Ranking Model: Best Configuration.

Index	Weight	Index	Weight
Textual Similarity	0.32	Question Score	0.07
Code Similarity	0.00	Accepted Answer Score	0.00
API Types Similarity	0.00	User Reputation	0.13
API Methods Similarity	0.30	Tags Similarity	0.18

performance of the ranking model. However, such an evaluation would have been biased by our manual validation of the links between contexts and relevant documents. We do not have such a threat in our studies, because the relevance was evaluated by external participants (**Study I**), or where participants used PROMPTER in maintenance and development tasks (**Study II**).

4.2.5 Putting It Together

The result of the PROMPTER ranking model is not sufficient to determine if a discussion is to be recommended or not. As we discussed in Section 4.2.1, the user can define the sensitivity of PROMPTER in notifying new discussions, and we showed how the *Query Service* determines if a new search is to be triggered or not. Triggering a new search and notifying a discussion relies on two thresholds: (i) *Query Entropy Threshold* and (ii) *Minimum Confidence Threshold*. The former is sent to the *Query Service* and defines the entropy level that should not be exceeded by the median (or mean, depending on the user preferences) of the terms of the query. If the value is below the threshold, a new search is triggered. The latter defines the minimum confidence level needed for a discussion to be recommended.

Both thresholds range in the interval $[0, 1]$. We limited the interval to $[0.1, 0.9]$ to prevent PROMPTER from not being able to submit new searches or notify new discussions. Whenever one uses the sensitivity slider, these values are modified in an inverse proportional way. A complete slide to the right means a high-sensitive configuration with *Query Entropy Threshold* at 0.9 and *Minimum Confidence Threshold* at 0.1, and the opposite otherwise.

4.3 Study I: Evaluating Prompter's Recommendation Accuracy

The goal of our first empirical study (*Study I*) is to evaluate, from a developer's *perspective*, the relevance of the Stack Overflow discussions identified by PROMPTER, *i.e.,* we are interested in understanding to what extent the retrieved discussion provides useful information to a developer working on a particular code snippet.

4.3.1 Study Design and Planning

The *context* of the study consists of *participants*, *i.e.,* various kinds of developers, among professionals and students, and *objects*, *i.e.,* source code snippets and its related Stack Overflow discussion as identified by PROMPTER. This study aims at answering the following research question:

> **RQ₁:** *To what extent are the Stack Overflow discussions identified by* PROMPTER *relevant?*

We asked 55 people (industrial developers, academics, and students) to complete a questionnaire aimed at evaluating the relevance of the Stack Overflow discussions identified by
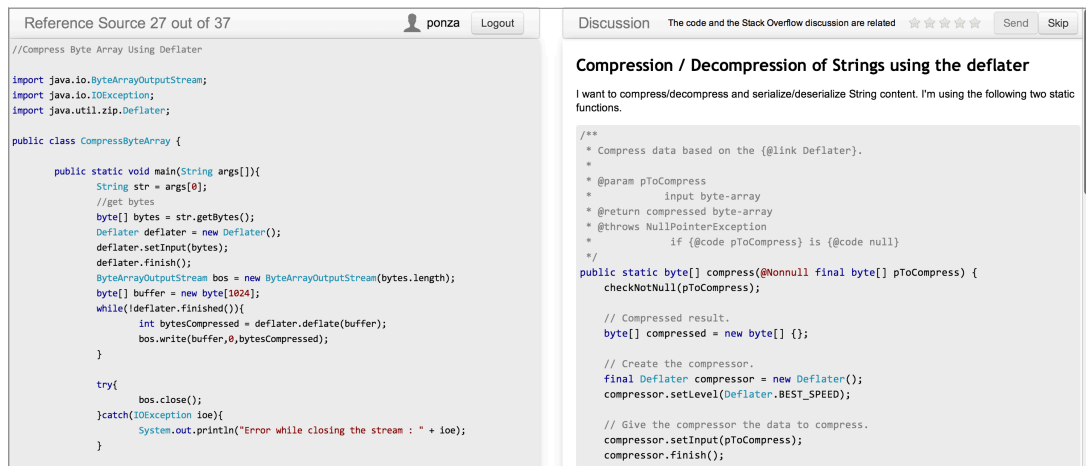
PROMPTER, by analyzing a specific code snippet. 33 participants filled in the questionnaire by answering the questionnaire through a Web application. They received the URL of the questionnaire, along with instructions, via email. Before accessing the questionnaire, participants were required to create an account, with login credentials, and to fill in a pre-questionnaire aimed at gathering information on their background. The answers to this pre-questionnaire are reported in Table 4.3.

Table 4.3. Study I Answers Questionnaire Summary. Percentages for Q3 and Q4 are calculated on the total number for subjects.

Question	Answer	Total	Percentage
Job	Industrial Developers	13	39%
	PhD Students	9	27%
	Master Students	7	21%
	Bachelor Students	2	6%
	Faculty	2	6%
Q1 : Have you ever worked in industry?	< 3 years	21	64%
If yes, how long?	3-5 years	3	9%
	> 5 years	2	6%
	Never	7	21%
Q2 : How long have you been programming in Java?	< 3 years	5	15%
	3-5 years	5	15%
	> 5 years	22	67%
	Never	1	3%
Q3 : What kind of traditional documentation do you usually use?	Javadoc	22	67%
	Official API Documentation	28	85%
	Books	9	27%
Q4 : What kind of additional resources do you usually use?	StackOverflow	26	79%
	Forums	24	73%
	Mailing List	7	21%
	Others	7	21%

The majority of the participants are industrial developers (39%) while 79% of participants declared to have spent some years in industry. Only 18% of participants have less than three years of experience in Java programming while 67% have more than five years. Most of participants use Javadoc and API Documentation as traditional documentation, while they mostly rely Stack Overflow and Forums as additional resources. Note that the different background of participants is a *requirement* for this study, since PROMPTER should be able to support developers having different skills, programming knowledge, and experience.

Once the participants answered the pre-questionnaire, they had to perform (up to) 37 tasks where the Web application showed a Java class and a discussion from Stack Overflow that PROMPTER suggested as top-1 ranked discussion among the results retrieved when analyzing that class. Even though participants had the chance of skipping tasks, we obtained at least 30 answers for each task. In the context of this study, we used the remaining 37 code snippets manually collected as explained in the previous section. Participants expressed their level of

Figure 4.6. An Example Question from the Questionnaire Assessing Discussions Retrieved by Prompter.

agreement to the claim *"The code and the Stack Overflow discussion are related"*, providing a score on a five points Likert scale [Opp92]: 1 (strongly disagree), 2 (disagree), 3 (neutral), 4 (agree), and 5 (strongly agree). In other words, the participants had to indicate to what extent the discussion could help them in completing the implementation task in the showed class.

Figure 4.6 shows an example of task from our survey. After submitting the score, participants were asked to write an optional comment to explain the rationale for their evaluation. We gave participants four weeks to complete the questionnaire. The participants were neither aware of the experimented technique (*i.e.,* PROMPTER) nor how the Stack Overflow discussions were selected. The Web questionnaire was also designed to (i) show the 37 tasks to participants in random order to limit learning and tiredness effects, and (ii) measure the time spent by each subject in answering each question. Response time was collected to detect participants who provided answers in less than 10 seconds, *i.e.,* without carefully reading code and the Stack Overflow discussion. This was not the case for any participant.

4.3.2   Analysis of the Results

We quantitatively analyzed participants' answers through violin-plots [HN98] to assess the ability of PROMPTER in identifying relevant Stack Overflow discussions given a piece of code. Violin plots combine box plots and kernel density functions, thus providing a better indication of the shape of a distribution. The dot inside a violin plot represents the median. A thick line is drawn between the lower and upper quartiles, while a thin line is drawn between the lower and upper tails.

Figure 4.7 shows the violin-plots of scores provided by participants of our experiment to each of the 37 questions composing our questionnaire (*i.e.,* their level of agreement to the claim *"the code and the Stack Overflow discussion are related"*). To understand whether PROMPTER excels in particular domains, we grouped the 37 tasks based on the topic/piece of technology they are related to, instead of ordering the tasks by their number.

Overall, the analyzed Stack Overflow discussions have been considered related to the showed Java code snippet. Specifically, 28 out of the 37 analyzed discussions (76%) received a median score greater or equal than 4. This indicates that participants agreed or strongly agreed to the above reported claim. Among the remaining 9 discussions, 5 (14%) achieved 3 as median, meaning that participants were generally undecided about their relevance to the code context,
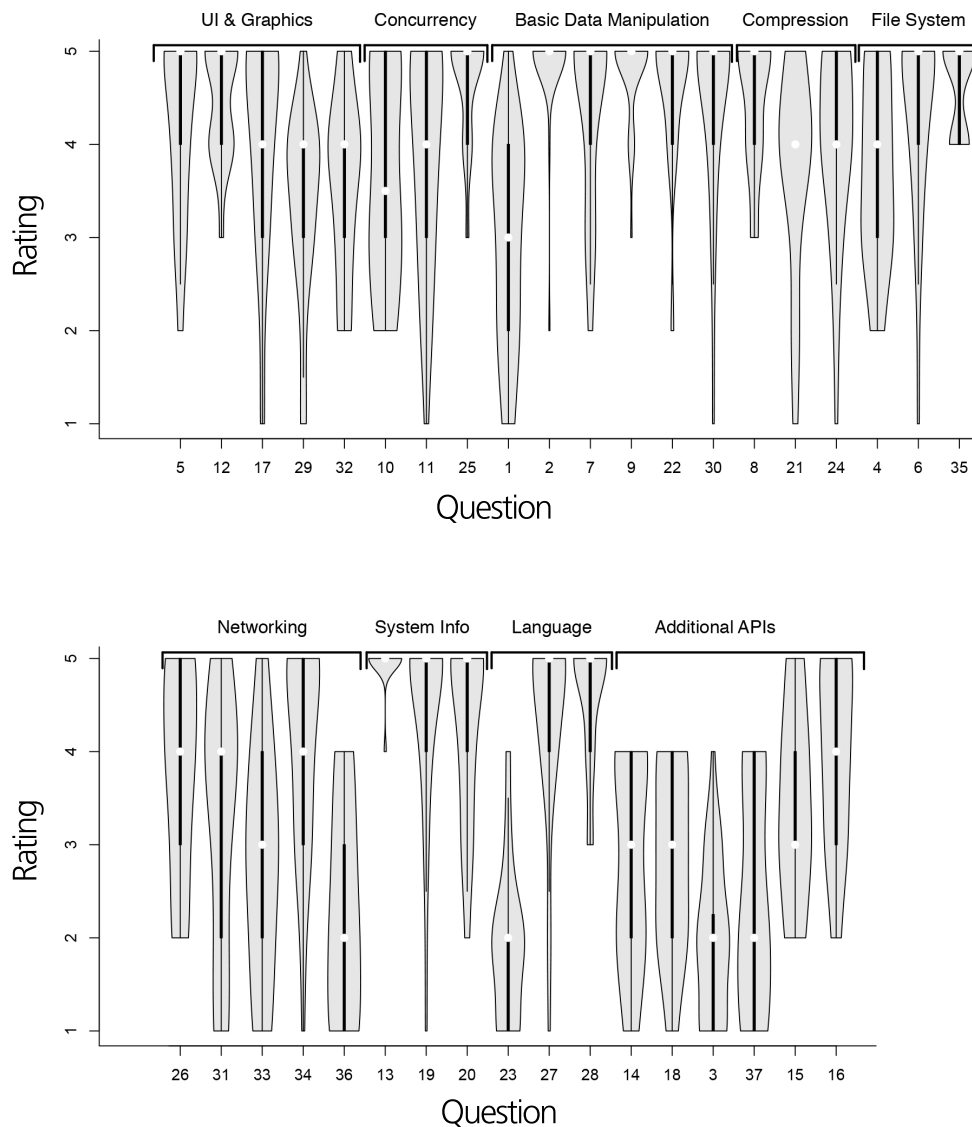
Figure 4.7. Violin Plots of Scores Assigned by Participants to the Evaluated Stack Overflow Discussions.

and four (10%) were mostly marked as not relevant achieving a median score of 2 (*i.e.,* disagree). In the following, we discuss two examples in which PROMPTER performed well, and a scenario in which we show its limitations.

Example I

The question reported in Figure 4.6 (question 8 in Figure 4.7) is an example where the achieved median score is 5. The class `CompressByteArray`—implementing the compression of a `byte` array using the `Deflater` class—has been linked by PROMPTER to the Stack Overflow discussion **Compression/Decompression of Strings using the deflater**[5]. Among the comments left by developers to their votes, one explained her "strongly agree" vote with the following sentence: *it is a good discussion if working on the* `CompressByteArray` *class, since it talks about compression with deflater, decompression, but also about problems that could be experienced and possible solutions.*

---

[5]`http://stackoverflow.com/questions/9542987`

Example II

Another Stack Overflow discussion felt by developers as strongly related to the companion Java class was the one entitled **Java regex email**[6] (question 9 in Figure 4.7) and associated to the `Utility` class in Listing 4.2:

Listing 4.2. Utility class

```java
import java.util.regex.*;

public class Utility {

  public static boolean isValidEmailAddress(String email) {
  //regex to match an e-mail address
    String EMAIL_REGEX = "^[\\w-_\\.+]*[\\w-_\\.]\@([\\w]+\\.)+[\\w]";
    Pattern emailPattern = Pattern.compile(EMAIL_REGEX);
    Matcher emailMatcher = emailPattern.matcher(email);
    return emailMatcher.matches();
  }
}
```

The `Utility` class emulates a developer experiencing troubles in writing the method `isValidEmailAddress`, aimed at validating through the Java `regex` mechanism an email address provided as parameter. The regular expression stored in variable `EMAIL_REGEX` is wrong, and for this reason `isValidEmailAddress` is incorrect.

In the Stack Overflow discussion retrieved by PROMPTER as the most related one to the `Utility` class, a user is asking help since she is experiencing a similar problem when trying to validate an email address using Java `regex`. The top answer in this discussion contains the solution to the problem in method `isValidEmailAddress`, *i.e.,* the correct regular expression to validate email addresses. This explains why almost all subjects involved in our study (26 out of 32) assigned a score equal to 5 to this discussion.

Example III

Developers did not consider particularly useful the discussion **Invoke only a method of a servlet class not the whole servlet**[7] related to the `ShoppingCartViewerCookie` servlet class (question 36 in Figure 4.7). The reason why PROMPTER linked `ShoppingCartViewerCookie` to this discussion is because it is about servlets, but not about the particular problem the developer wants to solve (*i.e.,* managing cookies). Instead, the discussion explains how to invoke a single method of a servlet. This was also confirmed by one of the participants: *"the SO discussion does not mention how to use cookies".* This example shows the limits of PROMPTER: It correctly captures the general context of the code (a developer is working on a servlet class), but it fails to identify the problem she is experiencing when trying to implement a specific feature. The same happened in the few cases where our approach obtained low scores (questions 3, 23, and 37 in Figure 4.7).

---

[6]http://stackoverflow.com/questions/8204680
[7]http://stackoverflow.com/questions/13509291

Summary of RQ$_1$. *The Stack Overflow discussions identified by* PROMPTER *are, from a developer's point-of-view, generally considered related to the source code. 76% of the discussions where considered related (median 4) or strongly related (median 5) by developers, while only 10% was considered as unrelated.*

4.4 Study II: Evaluating Prompter with Developers

The *goal* of this study is to evaluate to what extent the use of PROMPTER can be useful to developers during a development or maintenance task. The *quality focus* is the completeness (and correctness) of the task a developer can perform in a limited time frame, *e.g.,* because of a hard deadline. The *context* consists of *objects*, *i.e.,* participants have to perform two tasks with/without the availability of PROMPTER. We had 12 *participants* (3 BSc and 3 MSc CS students, and 6 industrial developers). Before the study, we screened the participants by using a pre-study questionnaire, asking them about their experience in programming and Java (the study tasks were in Java). The experience was measured in terms of (i) the number of years of Java programming, and (ii) a self-assessment based on a five-points Likert scale [Opp92] going from 1 (very low experience) to 5 (very high experience). Also, we asked participants which sources of documentation they generally exploit when programming.

Subjects analysis. All participants have at least 3 years of experience in programming, with a maximum of 12 reached by an industrial developer and a median of 6.5. They have a median of 4 years of Java programming experience. Participants claimed to have a good experience in programming and Java programming with a median of four (high experience) in both cases. Only two BSc students assessed their experience at 3 (medium), while all the others declared a high experience (4). The sources of information mostly exploited by participants when programming are: Stack Overflow (10 participants), Forums (8), Javadoc (8), and Books (6).

Tasks. The tasks participants have to perform are one maintenance task and one greenfield development task (*i.e.,* from scratch). The choice of tasks was performed taking into account that, being the study executed within a lab, the tasks could not be too long nor complicated. On the other side, the tasks could not be too simple, to avoid a "ceiling" effect, *i.e.,* that all participants correctly completed the tasks without problems, regardless of the use of PROMPTER.

Maintenance Task (MT). This task required the implementation of new features in a Java 2D arcade game, where the player controls a spaceship to destroy an attacking alien enemy fleet. In its original implementation, the game directly starts when its `Main` class is executed. We asked participants to perform the following changes:

- **Change 1.** When starting the game, present to the player a home screen containing two buttons named "Start Game" and "Show Best Scores".

- **Change 2.** By clicking on "Start Game", the player can fill in a form composed of a text box labeled with "Specify a Nickname" and a "Go!" button, that allows the user to start the game.

- **Change 3.** When the game is over, the score (*i.e.,* the number of aliens destroyed) must be stored together with the user's nickname in a file named `scores.xml`.

- **Change 4.** By clicking on "Show Best Scores", the player can view the ranking of the top 10 scores achieved ever. This data must be loaded from the previously described file `scores.xml`.

Development Task (DT). For this task the participants had to create from scratch a Java program that, given the URL of a Web page and an e-mail address, converts the HTML page into a PDF and then send it via email to the specified address. The task consisted in three sub-tasks aimed at implementing the following features:

- **Feature 1.** The program shows a form with the following input fields: (i) the URL of an HTML Web page, (ii) an e-mail address, and (iii) the "object field" for the e-mail to be sent.

- **Feature 2.** The program converts the HTML web page at the provided address in PDF, using the three following conversion rules: (i) the content of the HTML tag `<title>` must become the PDF title, using Arial font with a 16 pt size; (ii) the images in the HTML tags `<img>` must be shown center-aligned in the PDF; and (iii) the content of the HTML tag `<p>` must become the PDF body, by adopting as font Arial 12 pt.

- **Feature 3.** Once the PDF is created, it has to be sent as attachment in an e-mail to the specified address, with the specified object.

We did not provide to participants any indication about the strategy to follow in the implementation of the two tasks.

4.4.1 Research Questions and Variables

The study aims at addressing the following research question:

> **RQ$_2$:** *Does* Prompter *help developers to complete their task correctly?*

We investigate if the use of Prompter helps developers when performing coding activities and in particular to what extent—within the available time frame, and when working with or without Prompter—participants are able to correctly complete the task (or part of it).

The dependent variable aimed at addressing *RQ$_2$* is the task completeness. Since it is difficult to automatically evaluate task completeness, we asked two independent industrial developers to act as "evaluators" and to assess task completeness by performing code review on each task implemented by participants. The evaluators did not know the goal of the study nor which tasks were performed with (without) Prompter's support. To help them in the assessment, we provided a checklist aimed at assigning a fixed completeness score to each of the sub-tasks correctly implemented by participants when working on MT and DT. The checklist for the maintenance task was the following:

1. 15%: The home screen containing the buttons "Start Game" and "Show Best Scores" has been implemented.

2. 25%: The "Start Game" button correctly works, allowing the player to insert her nickname. Also, by clicking on "Go!" the game correctly starts.

3. 35%: When the game is over the player score is correctly stored in the `scores.xml` file.

4. 25%: The "Show Best Scores" button works, by showing the top 10 scores.

The percentage of completeness assigned for each sub-task was proportional to its difficulty and complexity. The evaluators were independent, and conducted a discussion in case of diverging

Table 4.4. Study II: Design.

Session	Group A	Group B	Group C	Group D
1	MT-Pr	MT-NoPr	DT-Pr	DT-NoPr
2	DT-NoPr	DT-Pr	MT-NoPr	MT-Pr

scores. This happened on four out of the 24 evaluated tasks and the divergence was quickly solved by evaluators performing an additional code inspection.

The main factor and independent variable of this study is the presence or absence of PROMPTER. Specifically, such a factor has two levels, *i.e.,* the availability of PROMPTER (Pr) or not (NoPr). Other factors that could influence the results are (i) the (possible) different difficulty of the two tasks MT and DT, (ii) the participants' (self-assessed) *Skills* and (iii) *Experience* in Java development, and (iv) the years of *Industrial Experience* (if any) they may have.

4.4.2 Study Design and Procedure

The study design—shown in Table 4.4—is a classical paired design for experiments with one factor and two treatments. The design is conceived in a way that:

- each participant worked both with and without PROMPTER's support,

- each participant had to perform different tasks (MT and DT) across the two sessions to avoid learning effect,

- different participants worked with and without PROMPTER in different ordering, as well as on the two different tasks MT and DT.

Overall, this means partitioning participants into four groups, receiving different treatments in the two laboratory sessions. When assigning participants to the four groups, we made sure that their level of experience was (roughly) uniformly distributed across groups.

We carried out a pre-laboratory briefing, in which participants were trained on the use of PROMPTER, and the laboratory procedure was illustrated in details. However, in doing so, we made sure not to reveal the study research questions. In addition, the training was performed on tasks not related to MT and DT to not bias the experiment.

Participants had to perform the study in two sessions of 90 minutes each (*i.e.,* participants had a maximum of 90 minutes to complete each of the required tasks) interleaved by a 60-minute break to avoid fatigue effects[8]. At the end of each session, each participant provided the code she implemented.

To simulate a real development context, participants were allowed to use whatever they want to complete the tasks including any material available on the Internet. After the study, we collected qualitative information by (i) using a post-study questionnaire and afterwards, by (ii) conducting focus-group interviews.

The post-study questionnaire was composed of: (i) three questions asking if participants used Internet, the suggestions by PROMPTER, and their own knowledge during implementation. To answer these three questions participants used a four points scale, choosing between *absolutely yes*, *more yes than no*, *more no than yes*, and *absolutely no*; and (ii) a question asking participants to evaluate the relevance of the suggestions generated by PROMPTER. In this case, we adopted a five-points Likert scale [Opp92] going from 1 (totally irrelevant) to 5 (very relevant).

---

[8]During the break participants did not have the chance to exchange information among them.

During the focus-group interview, two of the authors and all participants discussed together about PROMPTER, trying to point out its weaknesses and strengths. This interview lasted 45 minutes.

4.4.3 Analysis Method

In the following, we describe all the statistical procedures used to analyze data of this study and address $\mathbf{RQ}_2$. Analyses have been performed using the R statistical environment [R C14]. For all the used statistical tests, we consider a significance level $\alpha = 0.05$.

First, we provide an overview of the distribution of task completeness values for the two treatments PR and NOPR using box plots. In this study box plots are preferred over violin plots since they allow to better focus on the comparison between the completeness achieved by participants with the two treatments by not including visual details that violin plots provide. Then, we statistically compare results achieved with the two treatments. Given the chosen (paired) design, we test the null hypothesis:

H_0*: there is no statistically significant difference between*
the completeness achieved with and without PROMPTER*'s support.*

using the non-parametric Wilcoxon signed-rank test [She07]. This is a paired test, to be used when we need to compare related samples, as in our case where we need to compare the completeness achieved with and without PROMPTER. Since we do not know a priori in which direction the difference should be observed, we use a two-tailed test.

Besides testing the presence of a significant difference, we also assess the magnitude of the observed difference using the Cliff's delta (d) effect size [GK05] which is an effect size measure suitable for non-parametric data. The Cliff's d is defined as the probability that a randomly-selected member of one sample has a higher response than a randomly selected member of a second sample, minus the reverse probability. Cliff's d ranges in the interval $[-1, 1]$ and it is considered small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

To investigate whether PROMPTER helps differently for maintenance or development tasks, we pairwise compare results achieved for different tasks using the Mann-Whitney U test (equivalent to the Wilcoxon Rank Sum test) [She07]. In this case we use an unpaired test, because we cannot compare related samples, since each participant performed a development task with PROMPTER and a maintenance task without PROMPTER, or *vice versa*. Since here multiple tests have been performed, p-values have been adjusted using Holm's correction [Hol79]. This procedure sorts the p-values resulting from n tests in ascending order of values, multiplying the smallest by n, the next by $n-1$, and so on.

Finally, we used permutation test [Bak95] to check, from a statistical standpoint, the influence of the various co-factors and their interaction with the main factor treatment. The permutation test is a non-parametric alternative to the two-way Analysis of Variance (ANOVA); differently from ANOVA, it does not require data to be normally distributed. The general idea behind such a test is that the data distributions are built and compared by computing all possible values of the statistical test while rearranging the labels (representing the various factors being considered) of the data points. We used an implementation available in the *lmPerm R* package. We have set the number of iterations of the permutation test procedure to 500,000. Since the permutation test samples permutations of combination of factor levels, multiple runs of the test may produce different results. We made sure to choose a high number of iterations such that results did not vary over multiple executions of the procedure.

4.4.4 Quantitative Analysis of the Results

(a) All   (b) Maintenance task (MT)   (c) Development task (DT)
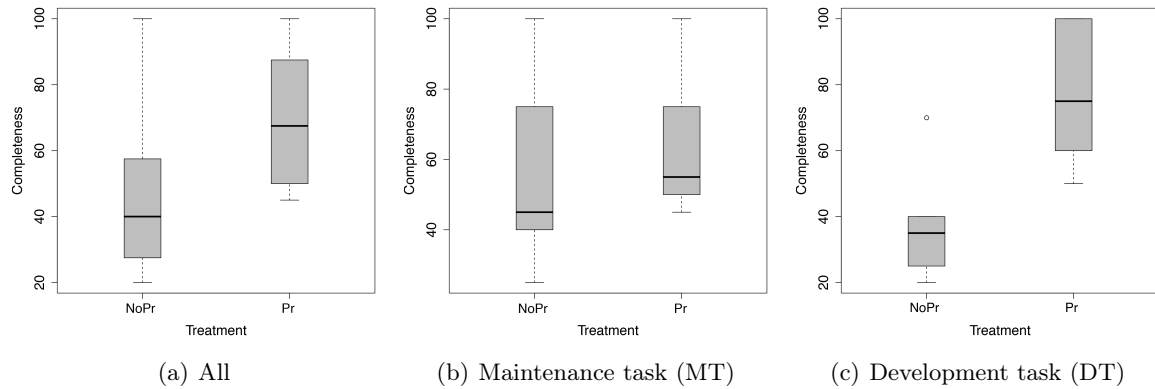
Figure 4.8. Box plots of Completeness achieved by Participants with (Pr) and without (NoPr) Prompter.

Figure 4.8(a) shows box plots of completeness achieved by participants when using (Pr) and when not using (NoPr) Prompter. As it can be noticed, participants using Prompter were able to achieve a level of completeness greater than those not using it. The Pr median is 68% (mean 70%) against the 40% median (mean 46%) of NoPr. In other words, Prompter allowed participants to achieve a median additional correctness of 28% (mean of 24%). The Wilcoxon paired test indicates the presence of a statistically significant difference, with a p-value lower than 0.01, hence rejecting H_0. The Cliff's d is 0.65, indicating a *large* effect size.

Figures 4.8(b) and 4.8(c) show box plots of completeness when focusing on results achieved for *MT* and *DT*, respectively, to better understand where Prompter results particularly helpful. Prompter helped participants in both MT and DT, increasing the median completeness achieved for MT of 10%, and for DT of 40%. The results of the Mann-Whitney unpaired two-tailed test indicates that for MT the difference is not significant (p-value=0.23) and the effect size is 0.38 (*medium*), while there is statistically significant difference for DT (p-value=0.03), with a *large* effect size (0.88). Prompter produced much more benefits for DT, where participants implemented from scratch and where they had to use several libraries, *e.g.*, to parse the HTML page, to convert it in PDF, to send an e-mail. In such a circumstance, Prompter provided an effective support by pointing to Stack Overflow discussions concerning the correct usage of such libraries.

Concerning the effect of all other co-factors, the permutation test results, reported in Table 4.5, indicate that:

- *Java Skills* and *Experience* have a significant effect on the participants' performance, although they do not interact with the main factor. In other words, people with higher Skills and experience perform better, independently of Prompter;

- There is no effect nor interaction of the *Industry Experience*;

- *Task* has no direct effect on the observed results. It marginally interacts with the main factor: As we have also explained above, and as one could have expected from Figures 4.8(b) and 4.8(c), Prompter resulted more helpful for DT than for MT.

Table 4.5. Effect of Co-Factors and their Interaction with the Main Factor: Results of Permutation Test.

Java Skills					
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	**0.01**
Java Skills	1	2566.87	2566.87	500,000	**0.02**
Treatment:Java Skills	1	1.88	1.88	58,583	0.94
Residuals	20	8487.50	424.38		

Java Experience					
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	**0.01**
Java Experience	1	2276.87	2276.87	500,000	**0.03**
Treatment:Java Experience	1	72.70	72.70	462,783	0.68
Residuals	20	8706.68	435.33		

Industry Experience					
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	**0.02**
Industry Experience	1	192.30	192.30	500,000	0.55
Treatment:Industry Experience	1	35.32	35.32	248,801	0.80
Residuals	20	10828.63	541.43		

Task					
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	**0.02**
Task	1	26.04	26.04	225,011	0.82
Treatment:Task	1	1426.04	1426.04	500,000	0.10
Residuals	20	9604.17	480.21		

4.4.5 Qualitative Analysis of the Results

Results from the post-questionnaire provided us with interesting observations. First, participants generally used Internet during the implementation of the required tasks. When being asked, six of them answered *absolutely yes*, five *more yes than no*, and one *more no than yes*. This is expected and consistent with the answers they provided to the pre-study questionnaire. Second, most participants felt to have used their knowledge in the tasks implementation, with four of them answering *absolutely yes*, six *more yes than no*, and two *more no than yes.*

As for the question related to the use of PROMPTER's recommendations, most of participants answered positively. Three of them answered *absolutely yes*, eight *more yes than no*, and two *more no than yes.* The latter participants explained that they received very few PROMPTER's recommendations, due to the fact that they spent much time on the Internet, trying to figure out how to implement the required tasks. This resulted in wasted time during which the code they were working on was untouched, leading PROMPTER to wait in vain for their moves to produce suggestions. Still, these two participants agreed that the few received recommendations were actually relevant to what they were implementing in the IDE.

The goodness of the PROMPTER's recommendations perceived by participants is evident when analyzing the answers to the question concerning the relevance of the Stack Overflow discussions pushed by PROMPTER in the IDE. Among the twelve participants, two of them classified the suggestions as *very relevant* (5), and the remaining ten as *relevant* (4).

We gained useful insights from the focused group interview. Participants agreed that PROMPTER is very useful when working on tasks in which the developer has poor experience, since the information brought in the IDE by PROMPTER helps the developer in enriching her knowledge about the task to be performed. For instance, one of the participants was experiencing problems with the repaint function provided in the `JFrame` by the `updateUI` method. PROMPTER pushed in his IDE a Stack Overflow discussion[9] exactly related to what he was trying to implement, solving his problem. Another participant, when starting to work on DT, observed the push notification of PROMPTER about a Stack Overflow discussion[10] providing guidelines on how to choose the HTML parser library to use. After reading the discussion, his choice was targeted on `jsoup`. Summarizing, participants identified the following PROMPTER strengths:

- the accuracy of the suggestions and the relevance of the suggested Stack Overflow discussions;

- the user interface: clean, clear, and not invasive;

- the ease of use, and minimal training required;

- the possibility to tune the sensitivity of PROMPTER, increasing or reducing the rate of suggestions.

Besides identifying PROMPTER strengths, the focused group interview we conducted allowed us to also identify PROMPTER's limitations. Specifically, participants would like to see the following improvements in future PROMPTER releases:

- the possibility to exploit information coming not only from Stack Overflow, but also from forums and programming tutorials available online; the current ranking model implemented in PROMPTER considers features that are typical of Stack Overflow discussions (*e.g.,* question score, accepted answer score, *etc.*), and thus it cannot be generalized to other sources of information as it is. However, the PROMPTER architecture easily includes additional ranking models customized to exploit useful data from specific sources of information (*e.g.,* other Q&A websites).

- a way to force PROMPTER to look for specific types of discussions on Stack Overflow. For example, participants would like the possibility to specify some key terms that should always be considered by PROMPTER when searching for discussions on Stack Overflow;

- the possibility to have a search field. Indeed, most participants agreed on the fact that PROMPTER loses its usefulness if the developer has no idea on how to start coding. In such a situation, the developer is forced to leave the IDE and surf the Web. Participants suggested the addition of a search field in the PROMPTER user interface that allows one to explicitly formulate and execute a query without leaving the IDE. As explained in Section 4.2.1, as a result of participants' feedback, the current version of PROMPTER already implements a search field for manual queries.

> **Summary of RQ$_2$.** *Quantitative results indicated that overall* PROMPTER *allowed participants to achieve a significantly better completeness of the assigned tasks. The collected feedback suggested that participants perceived the tool as usable, the suggestions accurate and not invasive. Eleven out of the twelve participants involved in our study claimed that they would like to use* PROMPTER *in their daily development activities.*

---

[9] http://stackoverflow.com/questions/11640494/
[10] http://stackoverflow.com/questions/3152138/

4.5   Prompter: one year later

One of the main challenges when dealing with recommenders like Prompter is the variability
of the information available. Prompter is mainly based on a Q&A website and search engines.
Thus, given the continuous growth of the web and its contents, our goal is to investigate is-
sues that pertain to the persistence of such information and the subsequent replicability of the
evaluation of approaches and tools based on information available on online forum.

In Section 4.3 we presented a study aimed at validating the usefulness of the ranking model
exploited by Prompter (see Section 4.2.4). In such a study the Prompter's ranking model was
used to recommend Stack Overflow's discussions for 37 development tasks (*i.e.,* code snippets).
Then, the relevance of the top-ranked discussion to its related task was judged by the involved
participants. After one year, we replicated *Study I* using the information available online at the
time of writing. In particular, *Study I* as described in Section 4.3 has been carried out in July
2013, while its replication (described in this section) has been performed in July 2014.

The *goal* is to replicate the retrieval of top recommendations for the 37 tasks considered
in Section 4.3 using Prompter, with the *purpose* of investigating (i) the replicability of the
study we performed and, above all (ii) to what extent the Prompter's recommendations—
and consequently, its performance—may vary over time. The *quality focus* is the performance
variability over time of recommender systems relying on online resources, and in particular of
Prompter which relies on Stack Overflow and on search engines. The *context* consists of
the same 37 development tasks considered in Section 4.3, and a pool of 18 people (industrial
developers, academics, and students) evaluating the relevance of the Stack Overflow discussions
retrieved by Prompter in July 2013 and July 2014.

4.5.1   Research questions

The research questions this study aims to answer are:

- **RQ₃:** *To what extent are the Stack Overflow discussions identified by* Prompter *in July
  2013 still relevant in July 2014?*

- **RQ₄:** *How is the developers' assessment of the new recommendations compared to those
  identified one year before?*

The first research question (**RQ₃**) aims at posing the premises of this study, *i.e.,* to investigate
whether starting from the same 37 code snippets used one year ago, Prompter recommends a
different Stack Overflow discussion for some of them (which of course may represent a better or
a worse recommendation). The results of **RQ₃** show that among the 37 tasks which we re-run
Prompter on, some resulted in a different recommendation as compared to the recommendation
obtained one year before. Thus, **RQ₄** aims at comparing the assessments provided by participants
to both recommendations, *i.e.,* the one provided by Prompter in July 2013 and in July 2014.

4.5.2   Study design and analysis method

The first step to answer **RQ₃** is to re-ask Prompter to retrieve and rank Stack Overflow dis-
cussions for each of the 37 tasks. This results in a ranked list of Stack Overflow discussions for
each of the 37 tasks.

To verify the replicability of *Study I*, we check for each of the 37 tasks in which position of its
related ranked list has been retrieved the Stack Overflow discussion recommended by Prompter

Table 4.6. Replication Study Answers Summary.

Question	Answer	Total	Percentage
Job	Industrial Developers	5	28%
	PhD Students	3	17%
	Master Students	6	33%
	Bachelor Students	3	17%
	Faculty	1	6%
Q1 : Have you ever worked in industry?	< 3 years	5	28%
If yes, how long?	3-5 years	2	11%
	> 5 years	1	6%
	Never	10	56%
Q2 : How long have you been	< 3 years	5	28%
programming in Java	3-5 years	7	39%
	> 5 years	6	33%
	Never	0	0%
Q3 : What kind of traditional	Javadoc	14	78%
documentation do you usually use?	Official API Documentation	14	78%
	Books	5	28%
Q4 : What kind of additional	Stack Overflow	18	100%
resources do you usually use?	Forums	14	78%
	Mailing List	0	0%
	Others	1	6%

one year before as the most relevant (*i.e.,* first in the ranked list). For matter of fairness, the ranking model was tuned exactly as in *Study I*. Also, we exploited the same entropy information we computed one year before on the Stack Overflow dump of June 2013. Our choice was dictated by the fact that the number of Stack Overflow discussions present in that dump was already huge (5,016,480), including a total of 105,439 different terms.

In a perfect scenario, where *Study I* is fully replicable, the top Stack Overflow discussion retrieved by Prompter in July 2013 for each of the 37 tasks should be still ranked in first position in July 2014. In other words, the study should be time independent.

As for **RQ$_4$**, we need to compare human-based assessment of the new recommendations with the assessment provided to the old recommendations. Such a comparison is performed only for the tasks on which the top-ranked Stack Overflow discussion has changed after one year. This is because we cannot just ask the study participants to assess the new recommendations and compare such assessments with those obtained in the previous study. This is because the assessment of the old recommendations has been performed by different people. Thus, the results could be influenced by subjectiveness or personal levels of experience/skills. To limit this threat, we asked the participants of this study to evaluate both the old and new recommendations. Based on the results of **RQ$_3$**, this study is limited to those tasks (29 in total) for which the top-ranked Stack Overflow discussion provided by Prompter (*i.e.,* the recommended one) changed between *Study I* and this replication. Overall, study participants assessed 58 recommendations.

We adopted the same instrumentation and set-up described in Section 4.3 by reusing the same web application to collect participants evaluations of the Prompter's recommendations. Again participants were required to create an account and to fill in a pre-questionnaire aimed at gathering information on their background. Of the 30 invited people, 18 completed our questionnaire. Their answers are reported in Table 4.6.

Table 4.7. Top-rated Stack Overflow discussions re-ranked by Prompter one year later.

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Google	-	1	-	-	-	-	10	-	1	-	-	4	6	9	43	-	-	4	-
Bing	3	-	-	-	1	-	-	10	-	-	1	-	-	-	-	5	-	-	2
Prompter	1	1	-	-	9	-	15	3	1	-	11	5	1	3	28	3	-	3	2

Task	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
Google	25	12	-	-	-	7	-	3	-	-	53	-	-	-	-	1	-	-
Bing	-	-	1	-	-	-	-	-	9	-	-	1	-	-	-	-	-	1
Prompter	26	1	1	-	-	1	-	2	4	-	2	59	-	-	-	1	-	4

Five of the involved participants are industrial developers, while eight declared to have some years of industrial experience (Q1). Only five participants have less than three years of experience in Java programming (Q2) and most of them rely on Javadoc and API documentation as standard sources of documentation (Q3) and on Stack Overflow and forums as additional resources (Q4).

Following the same setup used in *Study I*, we make use of violin plots to summarize the results. Also, we compare the assessment distribution for each task between the old and the new recommended Stack Overflow discussion by using Wilcoxon paired tests (two-tailed). In addition to that, we report the Cliff's d (paired) effect size measures of such comparisons. Note that, by design, this study requires a paired analysis, because for each task each participant evaluates both the old and new recommendation generated by PROMPTER.

4.5.3 RQ$_3$: To what extent are the Stack Overflow discussions identified by Prompter in July 2013 still relevant in July 2014?

Table 4.7 reports in column **Prompter** the rank assigned by PROMPTER in July 2014 to the top-ranked discussion retrieved in July 2013 (from now on, *old top-discussion*) for each of the 37 tasks object of our study. In particular:

- if the value in column **Prompter** is "1", this means that the top-retrieved Stack Overflow discussion for the specific code snippet (task) did not change after one year;

- if the value in column **Prompter** is x with $x > 1$, this means that the top-retrieved Stack Overflow discussion for the specific code snippet changed after one year, and the *old top-discussion* is now ranked in a lower position (x);

- if the value in column **Prompter** is "-", this means that the *old top-discussion* is not retrieved at all by PROMPTER one year later (*i.e.,* it is not present in the ranked list of Stack Overflow discussions generated by PROMPTER).

Table 4.7 also reports the rank of the *old top-discussion* in the Google and Bing search engines in July 2014 (*i.e.,* one year after *Study I*) to better analyze the cases where PROMPTER

was not able at all to retrieve the *old top-discussion* ("-" in column **Prompter**). As explained in Section 4.2, Google and Bing are exploited by PROMPTER to retrieve the Stack Overflow discussions to rank. Thus, if Google and Bing are not able to retrieve the *old top-discussion* for a task one year later, as a consequence also PROMPTER will not be able to retrieve it[11].

The numbers shown in Table 4.7 highlights as only for eight of the 37 tasks PROMPTER retrieves the same top-ranked discussion as one year before (*i.e.,* tasks 1, 2, 9, 13, 21, 22, 25, 35). This means that after one year, PROMPTER recommends a different Stack Overflow discussion for 78% of the tasks, highlighting a low replicability of *Study I* just one year after.

Analyzing the 29 tasks where the recommendation provided by PROMPTER changed, it emerges that in 13 of them (45% of cases) PROMPTER was not able to retrieve the *old top-discussion* because the employed search engines did not retrieve it anymore (*i.e.,* tasks 3, 4, 6, 10, 17, 23, 24, 26, 29, 32, 33, 34, 36—see Table 4.7). This could be due to several reasons, such as (i) the presence after one year of more related Stack Overflow discussions for the specific task, (ii) the deletion of the *old top-discussion* from Stack Overflow, or (iii) changes in the ranking algorithm exploited by the search engines. Despite the underlying reason(s) for such a result, it is clear that the recommendations produced by tools relying on search engines like PROMPTER are strongly influenced by changes in the output of such engines, undermining the replicability of any type of evaluation.

Concerning the remaining 16 tasks where the PROMPTER recommendation changed, in six cases (*i.e.,* tasks 8,14,16,18,19,27) the *old top-discussion* is still ranked in the top-three positions, even if it is no more the top-discussion. While for other kinds of recommender this result might show some sort of stability in the recommender's behavior (*e.g.,* tools using Stack Overflow discussions to document the source code [VPDC14]), given the *push mechanism* implemented in PROMPTER (*i.e.,* PROMPTER just pushes in the IDE the top-ranked Stack Overflow's discussion) also these cases represent a total different behavior by our tool at one year of distance. The situation is even more marked on the remaining ten tasks where the *old top-discussion* is ranked, in July 2014, in a much lower position.

Summary of RQ$_3$. *The recommendations provided by* PROMPTER *starting from the same 37 tasks exploited in* Study I *changed in 78% of cases when replicating the study one year later. This clearly highlights that (i) the performance of recommenders relying on volatile information mined from the Web can strongly change over time and (ii) empirical studies performed to evaluate such tools are almost not replicable. The results of RQ$_3$ pave the way to our RQ$_4$, where we investigate if the 78% tasks for which* PROMPTER *recommends a different Stack Overflow discussion results in an improvement or in a worsening of* PROMPTER*'s performance.*

4.5.4 RQ$_4$: How is the developers' assessment of the new recommendations compared to those identified one year before?

Figure 4.9 reports violin plots related to the assessment provided by the study participants to the old (red) and new (blue) recommendations. For the *old top-ranked* discussions (red violin plots), 18 out of 29 (62%) of the proposed discussions received a median score greater or equal than 4, that is, people agreed (31%) or strongly agreed (31%) on the statement *"The code and the Stack Overflow discussion are related"*. Of the remaining 11 discussions, 17% received a rating

---

[11]We consider a Stack Overflow discussion as not retrieved by Google and Bing if it does not appear in the first 100 retrieved Stack Overflow discussions.

between 2 and 4, meaning that people were generally undecided, while 21% received a rating lower or equal than 2.

For the *new* recommendations proposed by PROMPTER (blue violin plots), the results obtained are slightly worse than the previous one, but they seem to follow the same trend. Indeed, 15 out of 29 (52%) of the proposed discussions received a median score greater or equal than 4 (of those, 34% received a median of 5, while 17% a median of 4). For the remaining discussions, they equally (24%) received a median score either between 2 and 4 (participants undecided on the quality of the pushed Stack Overflow discussion), and lower or equal than 2.

Figure 4.9. Violin Plots of Scores Assigned by Participants to the Old (red) and New (blue) Top-ranked Stack Overflow discussion.
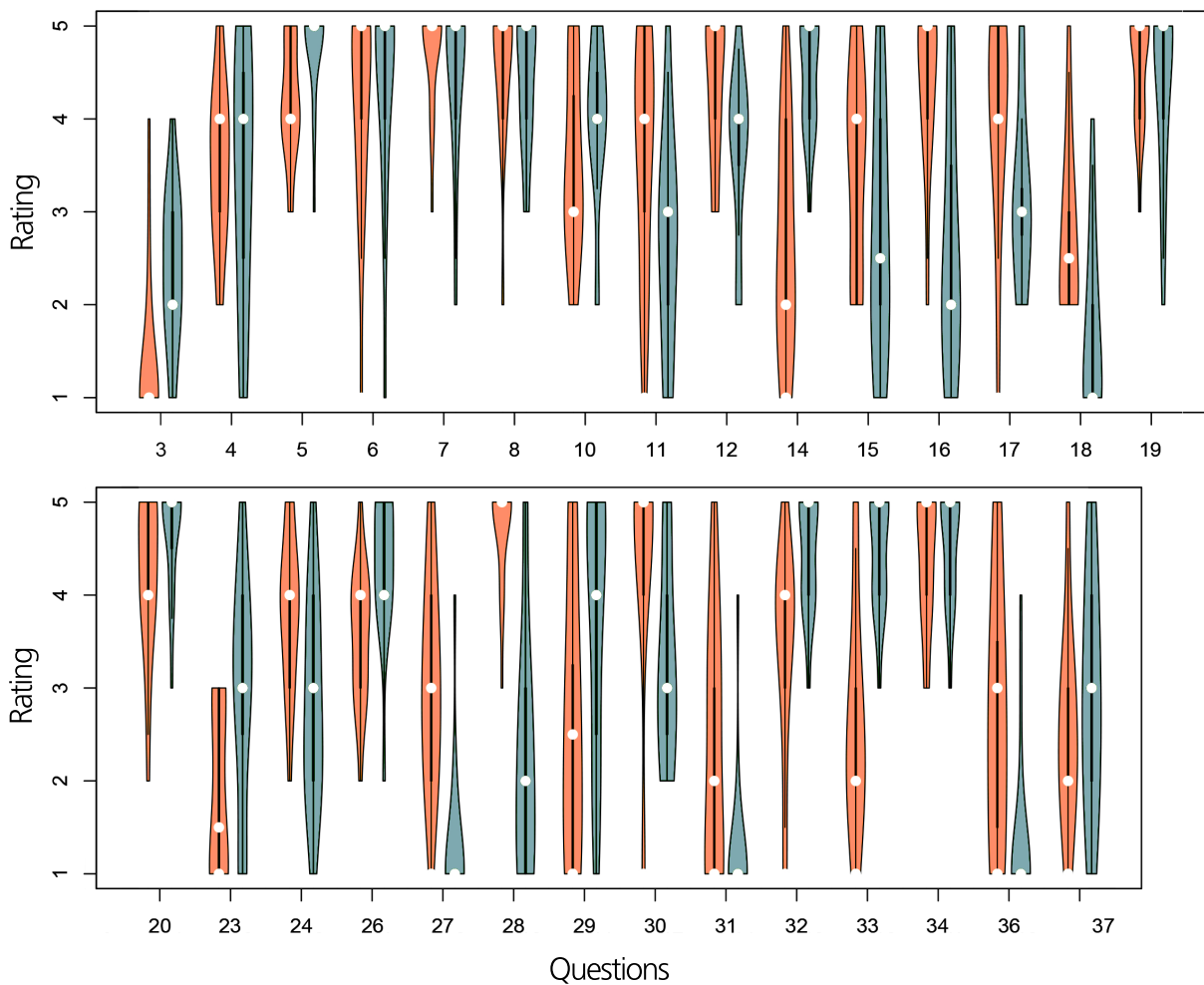
Table 4.8 reports the results of the Wilcoxon tests (p-values) and Cliff's d effect size measures when comparing the ratings assigned by participants to the old and the new top-ranked discussions for each of the 29 object tasks.

Based on the results of the Wilcoxon test, we divided the tasks in three categories: (i) *Improved*, seven tasks for which the new top-ranked Stack Overflow discussion achieves a statistically significant better user evaluation; (ii) *Neutral*, eleven tasks where none of the two top-ranked discussions is significantly better than the other as assessed by participants; and (iii) *Worse*, eleven tasks where the old recommendation provided by PROMPTER in *Study I* achieved a better

Table 4.8. Mann-Whitney test (p-value) and Cliff's delta (d). The recommendation achieving the better user' evaluations is reported in the second column: new (new recommendation), old (old recommendation), tie (not statistically significant difference).

Status	ID	Best	p-value	d
Improved	3	new	0.0050	0.71
	5	new	0.0147	0.42
	14	new	0.0036	0.78
	23	new	0.0261	0.50
	26	new	0.0312	0.56
	32	new	0.0085	0.57
	33	new	0.0028	0.75
Neutral	4	tie	0.7485	0.00
	6	tie	0.9319	0.05
	7	tie	0.1736	0.14
	8	tie	0.3796	0.20
	10	tie	0.1581	0.24
	19	tie	0.3053	0.16
	20	tie	0.3429	0.21
	29	tie	0.0685	0.41
	31	tie	0.0719	0.48
	34	tie	0.5653	0.08
	37	tie	0.3586	0.19
Worse	11	old	0.0334	0.44
	12	old	0.0070	0.59
	15	old	0.0146	0.31
	16	old	0.0036	0.69
	17	old	0.0282	0.51
	18	old	0.0017	0.59
	24	old	0.0332	0.48
	27	old	0.0020	0.78
	28	old	0.0014	0.82
	30	old	0.0031	0.73
	36	old	0.0054	0.64

user evaluation compared to the new one.

In the following we discuss the results achieved organizing the discussion by considering the above "group of tasks" (*i.e.,* Improved, Neutral, and Worse) trying to understand what happened in one year, what affected the model and produced different recommendations given the same task (*i.e.,* code snippet).

Improved

This category includes all the tasks for which the new recommendation achieved a better participants' assessment as compared to the old one. As reported in Table 4.8, for six of the tasks belonging to this category (all but task 5) the new recommendations were highly preferred by participants with respect to the old recommendations (the effect size is greater than 0.474, *i.e.,* a large effect size).

Table 4.9. Model Dump for Task 19.

Metric	New	Old	Original New
API Methods Similarity	1.00	1.00	1.00
User Reputation	0.00	0.00	0.00
Tags Similarity	0.75	0.67	0.75
Question Score	0.61	1.00	0.17
Textual Similarity	0.30	0.22	0.30
Confidence	**57.52%**	**56.04%**	**54.47%**

We manually analyzed each of the new recommended Stack Overflow discussions in this set to understand why the recommended Stack Overflow discussion changed after one year. A very simple explanation can be given for tasks 32 and 33 where the new recommended Stack Overflow discussions have been added on Stack Overflow in November 2013 and August 2013 respectively, *i.e.,* after the dataset construction for *Study I* was already completed. Thus, the new top-ranked discussions for these two tasks simply did not exist when *Study I* was carried out in July 2013.

For the remaining five tasks of this set, the *old top-ranked* Stack Overflow discussions have not been modified since the construction of the data set for *Study I* and the *new top-ranked* discussion already existed at the time *Study I* was performed. However, in all these cases PROMPTER assigns (in July 2014) a higher confidence level to the *new top-ranked* discussion. This is because in July 2013, when *Study I* was performed, the *new top-ranked* discussions were not retrieved by the search engines exploited by PROMPTER (otherwise the *new top-ranked* discussions would have been pushed by PROMPTER also in July 2013). This is likely due to changes in the ranking algorithms of the exploited search engines.

Neutral

This category includes all tasks for which there is no statistically significant difference in the assessment provided by participants to the old and the new PROMPTER's recommendations. The manual analysis of the old and new retrieved discussions highlights as for six of the tasks in this group (*i.e.,* tasks 6, 7, 8, 20, 34, 37) the reason for the change in recommendation is the same discussed above for the "Improved" group, *i.e.,,* in July 2013 the search engines exploited by PROMPTER did not retrieve the *new top-ranked* discussion.

The change in recommendation for tasks 4, 10, and 31 have a similar reason: the *old top-ranked* discussion exhibits a higher confidence level than the *new* one as assessed by the PROMPTER's ranking model. However, for these three tasks the search engines used by PROMPTER do not retrieve anymore the *old top-ranked* discussion in July 2014. Despite this, as assessed by participants, there is no clear difference in the quality of the old and the new recommendations.

The last two tasks in this group are those numbered with 19 and 29. For both of them the *new top-ranked* discussion already existed and was also retrieved by the search engines at the time of *Study I* (*i.e.,* July 2013). However, the PROMPTER's ranking model assigned a higher confidence level to the *old top-ranked* with respect to the *new* one at that time. The situation changed at one year of distance due to modifications applied in this time period to the *new top-ranked* discussion that pushed up its confidence level. To better understand, Table 4.9 reports the model dump for task 19 for (i) the *new top-ranked* in July 2014 (column "New"), (ii) the *old top-ranked* in July 2013 as well as in July 2014, since its confidence level was unchanged during the year (column "Old"), and (iii) the *new top-ranked* in July 2013 (column "Original New").

In the time period going between *Study I* and its replication, both the *new* and the *old*

Table 4.10. Model Dump for Task 15.

Metric	New	Old	Original New
API Methods Similarity	1.00	0.00	1.00
User Reputation	0.00	0.00	0.00
Tags Similarity	0.67	0.67	0.00
Question Score	0.17	1.00	0.17
Textual Similarity	0.25	0.25	0.25
Confidence	51.18%	26.85%	39.19%

top-ranked discussion received up votes. Note that the up votes affect the parameter *Question Score* exploited by the PROMPTER's ranking model. The *old top-ranked* discussion received six up votes, while two new up votes were assigned to the *new top-ranked* between July 2013 and July 2014. Given the normalization used, the increment in up votes for the *old top-ranked* discussion did not affect its overall confidence level. Indeed, the value of the *Question Score* parameter for the *old top-ranked* discussion was already equal to 1.00 (*i.e.,* the maximum score) when *Study I* was carried out (see Table 4.9). On the other hand, the change in the *Question Score* for the new top-ranked discussion allows its confidence value to increase from 54.47% up to 57.52%. This resulted in the overtaking of the *new top-ranked* discussion over the *old* one in July 2014. However, as highlighted by the very similar confidence levels and as also confirmed by participants, the two discussions have a very similar relevance for their related task.

Worse

This category includes all tasks for which the *old top-ranked* discussion was better assessed by participants as compared to the *new* one. For all these tasks but number 15, the Cliff's delta obtained in the comparison is greater than 0.474, *i.e.,* a large effect size.

   Also in this case, we looked into the different tasks to understand what driven the change in the PROMPTER's recommendation. Task 27 is the only one where PROMPTER recommends in July 2014 a Stack Overflow's discussion that did not exist at the time of *Study I* (one year before). For task 31 a change in the tags of the *old top-ranked* discussion happened in the year going from July 2013 to July 2014 has caused a decrease of its confidence level (and in particular of the *Tags Similarity* parameter), thus resulting in the recommendation of the *new top-ranked*.

   The case of task 15 is particularly interesting. Indeed, between *Study I* and its replication both the *old* and the *new top-ranked* discussions have been modified. In particular, the *old* discussion received 8 up votes, causing an equivalent increment in the overall score, while the *new* discussion has been modified in the tags and in the body, thus altering the metrics *Tags Similarity* and *Textual similarity*.

   Table 4.10 reports a dump of the model values for task 15. As before, column "New" shows the model values for the *new top-ranked* discussion in July 2014; column "Original New" reports the model values for the *new top-ranked* discussion when *Study I* was performed; and column "Old" shows the model values for the *old top-ranked* discussion. Despite the changes applied to the *old top-ranked* discussion its model remained stable between *Study I* and its replication since its *Question Score* was already equals to 1.00 (the maximum) in July 2013. The values reported in Table 4.10 clearly show as already in July 2013 the confidence level for the *new top-ranked* discussion was higher than the confidence level for the *old* one (49.22% *vs* 26.85%). This means that the *new top-ranked* discussion was simply not retrieved by the exploited search engines at the time of *Study I*.

Finally, discussions related to tasks 11,12,16,17,18, 24, 28, 30, and 36, have not been modified since July 2013. Thus, also in this case the PROMPTER recommendations have been strongly influenced by the different results generated by the search engines in the two different time periods.

Summary of RQ$_4$. *While the recommendations provided by* PROMPTER *one year later from Study I changed in 78% of cases, its performance did not show strong deviations, with just 24% (against the old 21%) of the new recommended Stack Overflow's discussions classified by participants as not related to the task at hand. Manual analysis suggests that the changes in the search engines together with the volatility of the information exploited by* PROMPTER, *represent the main reasons for 78% of different recommendations after just one year.*

4.6   Threats to Validity

Construct Validity

Threats to *construct validity* are related to the relationship between theory and observation. In *Study I* and in its replication, such threats are mainly due to (i) the fact that we mimic the code being written by a user by providing with PROMPTER a partially-complete class, and (ii) by letting the users provide evaluations using a Likert scale. Concerning the former, we made sure such classes were not too detailed nor too empty, to represent realistic situations where PROMPTER could be used. Concerning the latter, this is a standardized evaluation scale used to collect participants' feedback.

Study II overcomes the limitations of *Study I* mentioned above. In *Study II* threats to construct validity are due to how we measured the task completeness. Certainly, we could have used a test suite to measure the completeness in a objective manner. Conversely, code inspection allows us to evaluate partial implementations. The use of checklists and multiple independent evaluators limited the bias and subjectiveness.

Internal Validity

Threats to *internal validity* are related to factors that could have influenced the results. For *Study I* one factor to be considered is the knowledge of the participants—not known a-priori— of the APIs being used in the particular task. The availability of multiple participants with different degree of experience mitigates this threat. Students taking part in our evaluation were not evaluated based on the task outcome, and we asked participants not to use other sources of information during the task, *e.g.,* to use them as a comparative source to the provided discussion. In *Study II*, to limit the effect of participants' skills and experience, we have pre-assessed them and used this information assigning them to the four groups. We also analyze to what extent the usefulness of PROMPTER depends on the particular task.

For the replication of *Study I*, confounding factors could have influenced results of both **RQ$_3$** (different recommendation rankings) and **RQ$_4$** (developers' assessment). Specifically, for what concern the ranking, we cannot exclude that the different position (or the total disappear) of a question from the search-engine rankings may depend on changes/optimization in the search engines themselves. Nevertheless, we believe this can be one of the factors that affect the volatility of recommenders' results, and that one cannot control.

Concerning **RQ$_4$**, different subjects gave a different evaluation for the same recommendation (already assessed in *Study I*). Participants could have judged the same recommendation as very

relevant in *Study I*, and not relevant at all in the replication, and vice versa. This can happen because of the large difference of experience participants have. To verify whether such a situation could have occurred, we statistically compared—using Mann-Whitney tests (two-tailed)—the ratings provided to the 29 recommendations by participants to *Study I* and by participants to *Study I replication*). Results indicate the presence of a significant difference only for tasks 11 (p-value=0.03, median old study=4, median new study=3), 27 (p-value=0.0001, median old study=5, median new study=3), and 31 (p-value=0.02, median old study=4, median new study=2).

Conclusion Validity

For *Study I* we report descriptive statistics and violin plots of the collected results, along with participants' feedback, while for its replication, whenever possible, we use appropriate statistical procedures, namely Wilcoxon paired tests and Cliff's d effect size measures. For *Study II*, we used distribution-free tests (Wilcoxon, Mann-Whitney, and permutation test) and effect size (Cliff's d) measures, suitable for limited data sets as in our study. Whenever multiple tests are used on the same data, we apply p-value adjustment using the Holm's procedure [Hol79].

External Validity

Threats to *external validity* concern the generalizability of our findings. In terms of participants, the study involved both professionals and students, with different degree of experience. We claim the study provides a good coverage of the potential categories of users, although further studies with more participants are desirable. In terms of objects, we selected 37 tasks being different in terms of nature and required technical knowledge. However, we cannot exclude that our results depend on the particular choice of the tasks. For *Study II*, although we selected, both students and industrial developers, it is worthwhile to replicate the study with a larger number of participants. Furthermore, PROMPTER was only evaluated with two tasks that are not representative enough for tasks that developers would perform. We believe that *Study I* achieves a better *external validity* whereas *Study II* a better *construct validity*.

Finally, concerning the *Study I replication*, it is possible that the different ranking and evaluation obtained for the recommendations pertinent to the 29 tasks depend on these particular cases. In other words, there might be tasks—*e.g.,* related to emerging technology—for which recommendations can be more "volatile", while other tasks—*e.g.,* related to the usage of consolidated programming practices—-such as Java SDK—can be relatively more stable. Therefore, further studies can be needed to confirm or contradict the results obtained in this study.

4.7   Conclusions

We have presented an approach to turn an IDE into the developer's programming prompter. The approach is based on (1) automatically capturing the code context in the IDE, (2) retrieving documents from Stack Overflow, (3) ranking the discussions according to a ranking model, and (4) suggesting them to the developer when (and only if) it has enough self-confidence. We implemented our approach in PROMPTER, a tool embodying the ideal behavior a recommender should have: a silent observer of the developer, that only intervenes when it deems itself to have a relevant enough suggestion, and that does not force the developer to invoke it but is always available in case the developer needs it. Through a quantitative study (*Study I*), performed

via an online survey, we showed how the PROMPTER ranking model resulted to be effective in identifying the right discussions given a code snippet to analyze.

In a second study (*Study II*) we evaluated PROMPTER during maintenance and development tasks. We showed how, from a quantitative point of view, PROMPTER revealed to significantly help developers in completing the experiment tasks and how, from a qualitative point of view, the developer appreciated its features and usability.

We also replicated *Study I* after one year from the original experiment. Surprisingly, the results showed that starting from the same code snippets PROMPTER's recommendations changed in 78% of cases due to the volatility of the information it mines from the web. Despite this, the new recommendations still showed to be related to the task at hand in most of cases. However, the results of the replication clearly highlighted that recommenders built on top of information mined from the web may experience strong changes in their behavior during time. As a consequence, the replication of empirical studies aimed at evaluating such tools and techniques could be unfeasible.

Reflections

This chapter presented an approach not totally depending on pure information retrieval. The ranking model described in Section 4.2.4 takes into account several aspects of the information. The textual similarity between a Stack Overflow discussion and the source code in the IDE is decorated by including other types of similarity concerning code (*e.g.,* method names, and type names), and non-source related community information (*e.g.,* user reputation).

Even though this model moves some steps towards a holistic interpretation of the information, its implementation it is still reductionist. Indeed, the elements composing the model are just weighted and combined in a linear function, forcing the overall ranking model to be a sum of similarities. Implicitly, this model devises a fix structure of an artifact that needs to be satisfied, *i.e.,* in a Stack Overflow discussion, the code snippets need to match the same types, the same methods, while the narrative parts must use the same words as in the code in the IDE.

In addition, the model is reductionist in the way it treats the information together. According to Table 4.2, some of the weights assigned to metrics in the model are equal to zero (*i.e.,* Code Similarity, API Types Similarity, Accepted Answer Score), thus excluding such metrics from the computation. The configuration reported in Table 4.2 is just one of the possible local maxima that can be obtained in the training phase. Other model configurations, not reported in the chapter, exhibit different weight distributions, sometimes eliminating some of the zero values. In other words, a configuration of the model might better work for a subset of the artifacts used in the training phase, while another possible configuration might better fit a different subset of the same set of artifacts.

The actual ranking model highlights the need of a heterogeneous overview of the information to assess the quality of an artifact. The current implementation treats the information in a reductionist way by excluding certain types of information that might play a prominent role for certain artifacts. In the next chapter we show how the heterogeneity of the information can further increase when it comes at evaluating the quality of the narrative of a Stack Overflow discussion.

5

Improving Low Quality Stack Overflow Post Detection

In the two previous chapters Q&A websites like Stack Overflow, played a prominent role as source of information for developers. However, the quality of the contents provided by Q&A websites varies, and ranges *"from high-quality questions and answers to low-quality, sometimes abusive content [, thus making] the tasks of filtering and ranking more complex than in other domains"* [ACD⁺08]. In Stack Overflow, the task of keeping up the quality of questions is left to the crowd: Poor quality posts are identified by a selected subset of users in the community (*i.e.,* moderators) who have the rights of closing and deleting questions.

As reported by Correa *et al.* [CS14], around 80% of the questions take at least 1 month or more to receive a delete vote, and approximately 14% receive 3 delete votes before being actually deleted. This latency in the deletion process is a symptom of the amount of effort required by moderators to guarantee a satisfiable level of quality in Stack Overflow.

In this chapter we propose an approach to automatize the filtering process. We have devised a quality predictor that helps moderators in identifying poor-quality questions at their creation time, thus reducing the review time. To do so, we have investigated the concept of quality for Stack Overflow questions and developed the classification approach.

Structure of the Chapter

In Section 5.1 we describe the Stack Overflow review queue process. In Section 5.2 we discuss how we construct the datasets we use for our analysis. In Section 5.3 we present the metrics that we use to construct our classifier. In Section 5.4 we then present our classifier and the results we obtain. In Section 5.5 we discuss our findings. In Section 5.6 we discuss the threats to validity, and we draw our conclusions in Section 5.7.

5.1 The Stack Overflow Review Queue Process

Low quality posts in Stack Overflow are identified through a review queue system managed by moderators (a restricted set of users with enough reputation to unlock specific privileges[1]). Stack Overflow has 7 review queues:[2]

1. **Late Answers**: Answers which were posted much later than the question.

2. **First Posts**: First posts for users.

---

[1] http://stackoverflow.com/help/privileges/
[2] http://meta.stackexchange.com/questions/161390/

3. **Low Quality Posts**: Posts automatically determined to be of low quality based on several system criteria that generates a post quality score, or voted as such by users.

4. **Close/Reopen Votes**: Questions with active close votes or close flags show up in the close queue, and questions with active reopen votes, as well as questions which have been edited after closing, appear in the reopen queue.

5. **Suggested Edits**: Users without enough reputation to edit have their edits placed in this queue.

6. **Community Eval**: On the 60th day of beta, and every 90 days after that, this queue is filled with a set of posts which may be rated as "Excellent", "Satisfactory", or "Needs Improvement".

When a post in a review queue receives 3 *delete votes* by moderators, it is deleted from Stack Overflow[3]. The post remains visible to users with a reputation score above 10,000 and its author. A post can be undeleted again by moderators if and only if it receives 3 undelete votes.

The queue of our direct interest is the *Low Quality Posts* Queue, since it contains posts that have been automatically determined as low quality, by using several system criteria that generates a post quality score, or that have been manually flagged by users. We focus on improving the efficiency of the *Low Quality Posts* review queue. In particular, we propose an approach to refine the queue to remove misclassified (*i.e.,* good quality) post while retaining the bad quality posts in the review queue.

5.2   Dataset Construction

The September 2013 data dump contains 5,648,975 questions. Understanding the concept of quality for a question is very subjective if left to the judgment of a single person. On the other hand, in Stack Overflow low-quality question or a high-quality questions are defined by the crowd itself. We decided to rely on this information to identify quality. There are some actions the crowd can take to discriminate between bad and good posts: Every user can 'up vote' or 'down vote' a question or an answer, and moderators can vote for closing or deleting a question. Moreover, we can also consider information concerning the interaction with the question, *i.e.,* the answers. Indeed, a question with an accepted answer represents a discussion, or a posed problem, that has obtained the information needed. When an originator user (*i.e.,* the user who posed the question) accepts a specific answer, she is closing the discussion by pointing out the solution. Another aspect to take into account is the evolution of the question. Authors can modify their questions to clarify some parts, augment the information provided, and improve the overall quality. Modifying a question could have indirect side effects on the quality evaluation provided by the crowd. Therefore, we exclude from the dataset every question whose original body has been edited.

We also discard all questions whose score is 0. We assume that 0-scored questions have not attracted enough interests from the community, making it difficult to evaluate and classify their quality with the information at disposal.

After applying all the above mentioned filtering techniques we end up with a dataset of 1,262,959 questions, which we subdivide into high quality and low quality categories according to the following definitions:

---

[3]http://stackoverflow.com/help/deleted-questions

- **High Quality:** Questions, neither closed nor deleted, with a score greater than zero and with an accepted answer; 1,110,260 questions fall into this category.

- **Low Quality:** Questions with a score below zero, closed or deleted in their final state; 152,691 questions fall into this category.

With only two categories a clear quality distinction between questions is not available: The variance of the quality among posts in Q&A websites is considerable [ACD+08], which in turn leads to very noisy data. For this reason, we want to further refine each quality class by identifying 'very good' and 'very bad' questions.

'Very bad' questions are low quality questions that have been closed or deleted in their final state, without considering the score they obtained. We do not consider as 'very bad' the ones that have been closed because they were duplicates of existing questions, since the closing was not due to quality-related issues (indeed, a duplicate can be a clone of a very good question). We obtain a set of 81,854 'very bad' questions.

To define a set of 'very good' questions one is naturally drawn to selecting those with very high scores. This raises the question about which score threshold one should pick. We picked as score threshold of 7, which generates a set of 'very good' questions of roughly the same size as the set of 'very bad' ones. This leads to a set of 76,592 'very good' questions.

Table 5.1 reports the distribution of the quality classes in our dataset.

Class	Question Type Description	Size
A	*Very good* (with accepted answer, not closed, not deleted, score > 7)	76,592
B	*Good* (with accepted answer, not closed, not deleted, score 1-6)	1,033,676
C	*Bad* (not closed, not deleted, score < 0)	70,837
D	*Very bad* (closed or deleted)	81,854
	Total	1,262,959

Table 5.1. Quality classes of the questions in our dataset.

For the purpose of our study, we created four different datasets that we need for training and testing. As we see in Table 5.1, the four classes are unbalanced. In particular the class *Good* considerably differs from the other three classes. To reduce the bias in the classification phase, we balanced the size of the classes in each dataset by randomly downsampling the largest class [HG09]. Table 5.2 presents the four datasets with their related sizes.

Dataset	T_1 (Training)	T_2 (Testing)	T_3 (Training)	T_4 (Testing)
Very Good	10,000	66,592	5,000	65,837
Good	0	0	5,000	65,837
Bad	0	0	5,000	65,837
Very Bad	10,000	66,592	5,000	65,837
Total	20,000	133,184	20,000	263,348

Table 5.2. Datasets created for our study.

We created pairs of datasets with training and testing purposes respectively (see Section 5.4). Each dataset in a pair is not interleaved with the other. The first pair, T_1 and T_2, excludes

intermediate quality class, thus referring to our first rough definition of quality. The second pair , T_3 and T_4, refers to the extended definition of quality for Stack Overflow questions, thus including all four classes.

5.3 Metrics Definition

We identified three sets of metrics that cover textual and non-textual features of Stack Overflow posts. All the reported metrics are calculated by considering the data available (*e.g.,* author's information) at post creation time. All metrics range between 0 and 1, being normalized according to their minimum and maximum value over all the dataset (Table 5.1).

Stack Overflow (M_{SO}) Metrics (Table 5.3)

The staff of Stack Overflow provided us with a set of simple textual metrics currently in use. With such metrics Stack Overflow identifies the poor quality questions to be manually reviewed. Most of the metrics are mainly character-based (*e.g., Title Length*, *Title With Capital Letter*, *Body Length*, and *Lowercase Percentage*); exceptions are *Emails Count*, *URLs Count* and *Tags Count* as they identify emails, urls, and the amount of tags respectively. Stack Overflow also checks for text speak (*e.g.,* 'wats', 'doesnt', 'afaik') and emoticons as additional symptoms of poor quality posts.

Metric	Description
Body Length	The length in characters of the question including HTML tagging.
Emails Count	The number of e-mail addresses found in the question.
Lowercase Percentage	The percentage of lowercase characters all over the question.
Spaces Count	The total number of spaces in the question.
Tags Count	The number of tags assigned to the question by the author.
Text Speak Count	The number of text speak (*e.g.,*'afaik', 'rotfl') in the question.
Title Body Similarity	Textual similarity between title and body.
Title Length	The length in characters of the title of the question.
Capital Title	1 if the title begins with a capital letter, 0 otherwise.
Uppercase Percentage	The percentage of uppercase characters all over the question.
URLs Count	The number of URLs found in the question.

Table 5.3. Stack Overflow (M_{SO}) Metrics.

Readability (M_R) Metrics (Table 5.4)

We complement the Stack Overflow metrics with metrics that capture other textual features regarding readability. Focusing on the structure itself, we include in our analysis features like *Words Count* and *Sentences Count*. Another aspect characterizing a Stack Overflow question is the presence of code. Using the HTML structure of the posts, we identify the text within tags `<code>` to calculate the percentage of lines of code (*LOC Percentage*) in the full question's body.

We introduce *Metric Entropy* and *Average Terms Entropy* as features to evaluate the terms used in the textual part of a question. *Metric Entropy* is the Shannon entropy [CT91] divided

Metric	Description
Average Term Entropy	The average entropy of terms in a question, according to the Stack Overflow entropy index we devised. Each term's entropy is calculated on the Stack Overflow dataset.
Automated Reading Index	$4.71 \cdot (\frac{characters}{words}) + 0.5 \cdot (\frac{words}{sentences}) - 21.43$
Coleman Liau Index	$0.588 \cdot L - 0.296 \cdot S - 15.8$ where L = average number of letters per 100 words, S = the average number of sentences per 100 words.
Flesch Kincaid Grade Level	$0.39 \cdot (\frac{total\ words}{total\ sentences}) + 11.8 \cdot (\frac{total\ syllables}{total\ words}) - 15.9$
Flesch Reading Ease Score	$206.835 - 1.015 \cdot (\frac{total\ words}{total\ sentences}) - 84.6 \cdot (\frac{total\ syllables}{total\ words})$
Gunning Fox Index	$0.4 \cdot [(\frac{words}{sentences}) + 100 \cdot (\frac{complex\ words}{words})]$
LOC Percentage	The percentage of lines of code declared between tags <code> all over the text of the question.
Metric Entropy	$(\frac{shannon\ entropy}{body\ length})$. It represents the randomness of the information in the question.
Sentences Count	Number of sentences contained in the question, excluding <code> tags.
SMOG Grade	$1.0430 \cdot \sqrt{polysyllables \cdot (\frac{30}{sentences})} + 3.1291$
Words Count	The number of words in the questions, excluding <code> tags.

Table 5.4. Readability (M_R) Metrics.

by the length of the text, and represents the randomness of the information contained in the message. *Average Terms Entropy* measures the entropy of each term used in the question's text, against all the posts in Stack Overflow. We calculate the entropy for each term in the Stack Overflow data dump of September 2013 and we calculate the average of the entropy of each term used in the question's text. As we did in Section 4.2.3, the entropy value describes the discriminating power of a word, therefore the lower the average of terms' entropy, the higher the use of uncommon terms.

To assess the question readability, we also compute six standardized readability indexes: *Automated Reading Index* [SSS67], *Flesch Kincaid Grade Level* [Fle48], *Coleman Liau Index* [CL75], *SMOG Grade* [McL69], *Gunning Fox Index* [Gun52], and *Flesch Reading Ease Score* [Fle48]. These represent the comprehension difficulty when reading a passage in English and are different approximations and representations of the U.S. grade level[4] needed to comprehend the text. We argue that a lower readability could be a symptom of a poor quality question. To calculate these indexes we first remove code snippets from the question's body. We use the Stanford NLP Parser[5] to extract sentences and words, and TeX hyphenation [Lia83] to obtain syllables.

Popularity (M_P) Metrics (Table 5.5)

We also devise non-textual features to model the author writing the question. Analogously to M_{SO} and M_R metric sets, we require a snapshot of the status of authors when they created the question. The official data dump only reports the latest users' reputation levels (*i.e.,* computed

---

[4] http://en.wikipedia.org/wiki/Grade_levels
[5] http://nlp.stanford.edu/software/index.shtml

Metric	Description
Accepted by Originator Votes	The number of accepted answer obtained by the user.
Approved Edit Suggestion	The number of accepted edit suggestions the user obtained.
Answer Badges Count	The number of badges obtained for answers (*e.g.,* Great Answer, Good Answer, Nice Answer).
Badges-Tags Coverage	The percentage of tags covered by the badges owned by the user.
Bounty Start Votes	The number of votes the user received for having started a bounty (*e.g.,* gift points for the answer she wants).
Bounty End Votes	The number of votes the user received for having ended a bounty.
Close Votes	The number of close votes the user received for questions asked.
Deletion Votes	The number of deletion votes received for the questions asked.
Down Votes	The number of down votes the user received.
Favorite Votes	The number of favorite votes the user received.
Moderator Review Votes	The number of review votes the user received for her questions.
Offensive Votes	The number of votes the user received for offensive contents.
Reopen Votes	The number of votes the user received for questions already closed.
Question Badges Count	The number of badges obtained for questions (*e.g.,* Favorite Question, Stellar Question, Good Question).
Spam Votes	The number of votes the user received for spam contents.
Total Badges	The total number of badges the user obtained. It also includes badges for questions and answers.
Undeletion Votes	The number of undeletion votes the user received for questions already deleted.
Up Votes	The number of up votes the user received by the community.

Table 5.5. Popularity (M_P) Metrics.

in September 2013), thus we estimate Stack Overflow users reputation[6] by considering votes and badges received. The data dump provides all the votes a user received and the date when they were given. Representing the snapshot of the author's reputation at question-creation time requires to filter out votes and badges received after the question creation date. We consider three metrics: *Badged Answer Count, Badges Question Count,* and *Badges-Tags Coverage.* The first two represent the badges received concerning question (*e.g.,* 'Favorite Question' and 'Stellar Question') and answers (*e.g.,* 'Great Answer' and 'Good Answer'), while the latter refers to the coverage of author's badges with respect to the tags assigned by the author to the new question.

---

[6]http://meta.stackoverflow.com/questions/7237/how-does-reputation-work

5.4   Data Analysis

In the previous sections, we discussed how to define the quality of a question in Stack Overflow and we identified the features of a question and its author likely correlate with quality. In this section, we investigate such correlations through two empirical studies:

1. In Section 5.4.1, we use machine learning, and in particular decision trees, to classify a question's quality using different combinations of metrics.

2. In Section 5.4.2, we adopt a simpler approach that uses genetic algorithms to train a linear function expressing a measure of a question's quality, and then we investigate how such a measure can be used to perform more precise predictions on a question's quality.

3. In Section 5.4.3, we take advantage of the linear quality function trained devised in Section 5.4.2 to perform refinement of the review queue by leveraging the distribution of the output of the quality function.

5.4.1   Classification with Decision Trees

The first experiment we conducted involves the use of a simple machine learning algorithm to classify the quality of a question as bad or good, as defined in Section 5.2.

Considering the objective of our research, we want not only to predict the quality of a new question, but also to understand which classes of metrics influence the quality of a question. For this reason, we chose *Decision Trees* [WF05], a machine learning algorithm whose output can be easily interpreted. We conducted the experiments by considering all the different combinations of metric sets, to understand which ones give better precision in terms of identifying the quality of a submitted question.

A decision tree is a tree in which each internal node (non-leaf) is labeled with a feature, and arcs from any internal node are labeled with exclusive predicates that summarize possible distinct values for the feature. Finally, each leaf of the tree is labeled with a class. In Figure 5.1 we show a portion of a decision tree trained on T_4 using popularity metrics (M_P) as an example.

As we can see, in our case, the features are question metrics, and the class represents the quality to be assigned to the posts exhibiting the conjunction of predicates represented by arcs connecting the root to the leaf.

We trained decision trees on the largest datasets T_2 and T_4, considering a minimal amount of 50 posts per leaf and 0.25 confidence value. We performed 10-fold cross validation.

Results

Table 5.6 shows the results of the first experiment on the two data sets (T_2 and T_4) that correspond to the different definitions of quality we identified (see Section 5.2).

In both cases, considering a single set of metrics, popularity metrics (M_P) give the best results in terms of precision to identify both good and bad posts. The set of metrics considered by Stack Overflow (M_{SO}) perform worse than readability metrics, and may also introduce noise in the classification: When combined with popularity metrics or readability metrics, it may actually decrease prediction precision. The combination of all the three sets of metrics does not significantly increase prediction performance compared to popularity metrics alone. This reveals that *the popularity of the author is more important than textual features to determine the quality of a new question.*
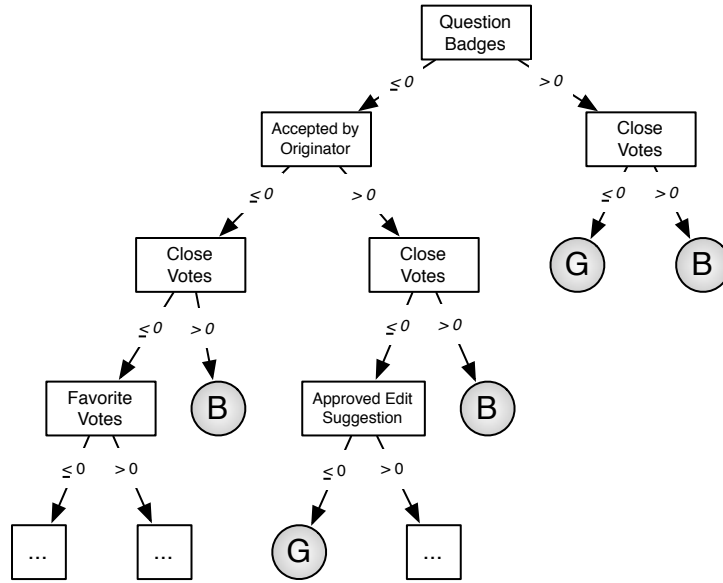
Figure 5.1. Portion of a Decision Tree trained on T_4 and M_P.

Dataset	Metrics	P_g	P_b	Average ROC
	M_{SO}	62.9%	62.1%	0.667
	M_R	66.8%	62.1%	0.676
	M_P	74.3%	73.2%	0.795
T_2	$M_{SO} \cup M_R$	66.3%	64.1%	0.702
	$M_{SO} \cup M_P$	74.0%	74.5%	0.808
	$M_R \cup M_P$	76.0%	75.1%	0.824
	$M_{SO} \cup M_R \cup M_P$	76.2%	75.2%	0.829
	M_{SO}	61.2%	61.7%	0.655
	M_R	62.9%	61.1%	0.653
	M_P	73.0%	69.7%	0.734
T_4	$M_{SO} \cup M_R$	63.3%	62.8%	0.676
	$M_{SO} \cup M_P$	72.7%	71.6%	0.780
	$M_R \cup M_P$	73.3%	71.9%	0.788
	$M_{SO} \cup M_R \cup M_P$	73.2%	72.0%	0.789

Table 5.6. Classification Results using Decision Trees.

We interpret this fact as follows. First, a question's quality, in terms of how the crowd will react to it, is inherently related more to the semantics and intention of a question than the textual way it is formulated. Second, the history of a users' interaction with the community, which determined their actual reputation, will determine the quality of questions that they will ask in the future or, at least, will provide some bias towards this author by the community (in particular if we consider that moderators are elected by the crowd). This insight makes the interaction between authors and the community a notable component for predictions about a question's quality.

Java Subset

The overall level of precision reached by decision trees is relatively unsatisfactory for an automated process aimed at discarding bad questions, and in particular in the ideal dataset T_4. Given these results, we considered that questions about programming languages could be less noisy and better classifiable in terms of quality. We constructed a subset of T_4 containing only questions about the most popular programming language, Java, and we ran again the decision tree learning algorithm. Table 5.7 shows the classification results for this subset. The results are only moderately better, leading us to the conclusion that it is not so much about what is being discussed, but by whom things are being discussed.

Dataset	Metrics	P_G	P_B	Average ROC
T_{Java}	M_{SO}	62.1%	61.7%	0.662
	M_R	63.8%	62.9%	0.672
	M_P	76.3%	77.0%	0.805
	$M_{SO} \cup M_R$	64.6%	64.3%	0.697
	$M_{SO} \cup M_P$	75.3%	75.5%	0.817
	$M_R \cup M_P$	76.9%	75.9%	0.818
	$M_{SO} \cup M_R \cup M_P$	76.3%	76.4%	0.823

Table 5.7. Classification Results using Decision Trees only on Java questions.

Leaf Inspection

The inspection of the learned decision trees gives other important and detailed insights on which metrics influence the most the quality of questions, and suggests a different way to approach the problem of quality prediction. Each leaf on the learned decision tree is linked to a particular decision on the classification of a question's quality, *i.e.,* either good or bad. When a decision tree is trained and tested against a given data set, the learning algorithm also outputs the amount of data associated with a specific leaf, and the number of misclassified elements. Even if the overall precision of the decision tree is low, some leaves may exhibit a precision value that is particularly high, thus disclosing metrics more related to a particular quality class.

Table 5.8 and Table 5.9 show examples of leaves on the decision trees that can correctly predict good or bad quality posts on subsets of data larger than **1%** of the original dataset, having a precision greater than **75%**. Nevertheless, the leaves that classify posts in such way are quite uncommon. Popularity metrics provide very good predictive performance even in this case, and we can mine the historical characteristics of users that influence future questions quality.

Consider, for example, the decision tree trained on dataset T_4 using popularity metrics. The precision P_B to predict bad posts is as low as **73%**. If we inspect the decision tree, we can see that on a subset of the data made of **14, 258** questions (corresponding to **5.4%** of T_4), the decision tree can correctly predict that **81.2%** of them is bad with only **770** misclassified questions. This subset corresponds to a specific leaf of the decision tree, predicting bad quality questions candidates if a user has received no badges for questions, and has received a certain number of question closure votes. We also learn that if a user has instead received question badges but no closure votes, then her question will likely be of good quality, with a precision of **83.9%**.

From the same decision tree, we also discover a very interesting leaf concerning bad quality posts. When a user obtained no question badges, neither close votes nor favorite votes, has no accepted answer in her history, possesses one badge at most, and asks a question of less than

Name	Decision Tree (DT) Leaf
L_1	$QuestionBadges = 0 \ \wedge \ CloseVotes > 0$
L_2	$QuestionBadges > 0 \ \wedge \ CloseVotes = 0$
L_3	$QuestionBadges = 0 \ \wedge CloseVotes = 0 \ \wedge \ AcceptedByOriginatorVotes > 0 \ \wedge$ $ApprovedEditSuggestion = 0$
L_4	$QuestionBadges = 0 \ \wedge CloseVotes = 0 \ \wedge \ AcceptedByOriginatorVotes > 0 \ \wedge$ $ApprovedEditSuggestion = 0 \ \wedge LOCPercentage > 0.017705$
L_5	$QuestionBadges = 0 \ \wedge \ AcceptedByOriginatorVotes = 0 \ \wedge \ ClosedVotes =$ $0 \wedge \ FavoriteVotes = 0 \ \wedge wordsCount \leq 64 \ \wedge \ totalBadges \leq 1$

Table 5.8. Selected Leaves on Learned Decision Trees

Dataset	DT Leaf	Metric Set	Size	Perc.	Class	Precision
T_2	L_1	M_P	13,033	9.9%	D	85.9%
	L_2	M_P	17,480	13.2%	A	85.2%
	L_3	M_P	28,091	21.1%	A	75.4%
	L_4	$M_P \cup M_R$	12,538	9.4%	A	88.7%
T_4	L_1	M_P	14,258	5.4%	C+D	81.2%
	L_2	M_P	26,426	10.0%	A+B	83.9%
	L_3	M_P	46,655	17.7%	A+B	76.7%
	L_5	$M_{SO} \cup M_R \cup M_P$	33,879	12.9%	C+D	78.2%

Table 5.9. Relevant Leaves on Learned Decision Trees.

64 words, it is likely to be a bad quality question with a precision of 78% on the 12.9% of the overall data in T_4. This finding remarks how the interaction of the user with the community matters and influences questions' quality estimation by the crowd. Indeed, some examples of users matching this leaf would be newcomers who have never interacted with the community, or people who provided neither notable questions (*i.e.,* no favorite votes, no question badges) nor accepted answers (*i.e.,* no accepted by originator), thus interacting not successfully with the community.

5.4.2 Linear Quality Function Classification

Given the limitations of the predictive performance of decision trees, and the fact that the analysis of leaves led to limited insights about what distinguishes good from bad questions, we decided to adopt a different approach for the classification of question quality, based on linear quality functions.

Intuitively, a quality function assigns a value to a post based on a given set of metrics. A quality function should assign a negative value to bad posts and a positive value to good posts. One of the benefits of such an approach to classification is that the predicted quality is not binary, but has a range and can therefore express intermediate levels of quality. To learn a quality function for a given metric set, we used genetic algorithms. A genetic algorithm [Gol89] is a search algorithm inspired by the process of natural selection; we exploit such a search approach to find a set of coefficients of a quality linear function given a metric set and a training dataset.

In a genetic algorithm, possible candidate solutions (*individuals*) are evolved towards better solutions that tend to maximize a given *fitness function*. A candidate solution (*i.e.,* a *gene*) is composed of a set of properties (*i.e.,* its *chromosomes*) that are mutated and altered during the

Metrics	Quantile	Left Tail	P_B	Right Tail	P_G
Trained on T_1, Tested on T_2					
M_{SO}	0.25	34,718	62.0%	34,106	58.3%
	0.10	14615	67.2%	14,466	58.2%
	0.05	7,341	69.5%	7,288	60.0%
	0.01	1,364	77.0%	1,740	57.4%
M_R	0.25	30,912	64.2%	39,528	61.9%
	0.10	11,906	64.2%	16,270	49.0%
	0.05	5,896	64.7%	8,091	39.1%
	0.01	1,237	66.9%	1,625	30.2%
M_P	0.25	46,016	68.4%	25,841	85%
	0.10	17,542	74.1%	11,474	88.3%
	0.05	8,495	78.2%	6,931	89.5%
	0.01	1,718	81.0%	2,251	90.1%
Trained on T_3, Tested on T_4					
M_{SO}	0.25	63,944	61.9%	66,888	59.2%
	0.10	25,114	66.6%	26,081	60.8%
	0.05	11,984	69.0%	12,781	60.4%
	0.01	2,291	73.8%	2,487	58.6%
M_R	0.25	61,630	62.8%	69,679	50.3%
	0.10	23229	63.5%	28,381	40.0%
	0.05	11,696	63.1%	14,421	34.9%
	0.01	2,338	61.7%	2,772	30.52%
M_P	0.25	63,152	64.7%	69,542	68.9%
	0.10	21,987	70.4%	24,350	70.9%
	0.05	10,480	71.3%	11,054	73.3%
	0.01	1,787	71.3%	1,661	90.8%

Table 5.10. Classification Results using Quality Functions.

process of *evolution*. Evolution starts from a set of randomly generated individuals, and proceeds by modifying a *generation* of individuals through subsequent iterations. At the beginning of each iterative process, the fitness of individuals of a generation is evaluated. Usually, the fittest individuals are selected from the population, and randomly mutated or recombined to form a new generation, *i.e.,* to produce a new set of individuals for the next iteration. The algorithm stops when a pre-defined value of fitness for the best individual is found, or when a maximum number of generations has been produced. Overall, a genetic algorithm requires a definition of individuals through their chromosomes and a fitness function. Since we want to search for a linear quality function, we implemented the evolutionary search as follows:

- The chromosome of an individual is a set of coefficients, one for each metric in the considered metric set, ranging in the $[-1, +1]$ interval.

- The fitness function is determined from the number of posts in the training set that are correctly classified, *i.e.,* the posts for which the quality function outputs a negative value for a bad post and vice-versa. In other words, the fitness function is the classification precision on a given training set.

We implemented the evolutionary search by using an open source framework called JGAP.[7] The fitness function evaluation is relatively costly, and depends on the size of the training set. Since each individual must be checked against the whole training set at each generation, it is impossible to search for quality functions using T_2 and T_4 as datasets, since they are too big. For these reasons, we used the smaller datasets T_1 and T_3 to train quality functions. We trained the genetic algorithm by using a population size of 64 individuals for 20 generations, and we constructed a quality function for each distinct set of metrics.

Results

Table 5.10 summarizes the classification results for quality functions. After training the quality functions on T_1 and T_3, we tested their predictive performance on T_2 and T_4, respectively. With quality functions, we can easily identify questions with very high or very low predicted quality. We consider the distribution of qualities as evaluated on the training set as reference, and we calculate 4 different quantile values, of decreasing size, corresponding to the left and right tail of the distribution. Then we project the quantile values on the testing set and we consider the projected left and right tails, on which we calculate corresponding precisions, respectively P_B and P_G on Table 5.10. Even in the case of quality functions, popularity metrics exhibit the highest precision on the testing sets. However, on the noisy dataset, and considering the smallest quantile size, the metrics on use at Stack Overflow could predict bad posts with a slightly higher precision compared to popularity metrics (73.8% vs 71.3%).

Learned Quality Function Inspection

The structure of learned quality function reveals important insights about the metrics to determine good or bad quality of posts. Table 5.11, Table 5.12, and Table 5.13 show, for each learned quality function on a given training set, the role of each metric. In particular:

- Each coefficient with a strong positive value, close to 1, contributes to increase a question's quality.

- Each coefficient with a strong negative value, close to -1, negatively contributes to a question's quality.

- Each coefficient with a value close to 0 essentially does not contribute to determine quality.

Although the generalizability of the results can be questioned, the following findings emerge:

- For both datasets, *the number of down votes received by a user is a strong component of quality.* Essentially, if users have received a significant amount of down votes, they will be more likely to formulate good quality questions to improve their reputation.

- Another strong component of high quality is having answers that have been accepted by the originator user. In other words, having produced very good answers in the past has impact on producing good questions in the future.

- On the contrary, *up votes received on the past do not influence quality.* This is counterintuitive; intuitively, it means that the fact that users performed well in terms of questions that the crowd appreciated does not have an impact on the quality of their future questions;

---

[7]http://jgap.sf.net

- A good *tagging of code elements* in a question determines high quality. We expected this, for a Q&A website like Stack Overflow.

- *Text speak determines bad quality.* Moreover, a low number of sentences in the question negatively influences quality.

The following characteristics influence one of the quality functions in the two data sets and become irrelevant on the other one:

- The number of urls in a post seem to be related to high quality ($w_G = 0.93$, Table 5.11) post in the noisy data set, but the contribution is only minimal ($w_N = 0.11$, Table 5.13) on the ideal dataset T_2.

A few metrics are related in completely opposite way to a post's quality if we consider the more noisy dataset T_4 instead of T_2 where the quality classes are clearly separated. In particular:

- Favorite votes received by a user seem to be a strong component to determine high quality posts ($w_G = 0.83$, Table 5.11), while instead it is a relatively strong component for bad quality ($w_B = -0.22$, Table 5.12) in the noisy dataset. This means that users that received favorite votes are somehow prone to produce high quality questions but also a great number of questions that are not to be deleted, but receive, on average, negative scores.

- Word count seems to characterize very good posts ($w_G = 0.83$, Table 5.11), but when intermediate quality questions are added in the data set, a high number of words determines bad quality, even if not strongly ($w_B = -0.44$, value not shown in Table 5.12).

- On the opposite side, body length relates to very bad quality ($w_B = -0.93$, Table 5.12) on T_2, but with strong quality ($w_G = 0.90$, Table 5.11) on the noisy dataset T_4.

Interaction between Metric Sets

Each quality function, associated with a given metric set, shows a relatively good predictive performance, which varies considering the dimension of the quantile to identify individuals with very low or very high quality. We manually inspected the smaller quantiles for each metrics set and we noted that each set contained questions with different features that would classify them as good or bad, as expected. In other words, each metric set captures different characteristics of a question quality and of the user who posted it, and it is reasonable to expect that we can achieve better precision by identifying questions who have very good or very bad values for quality functions of more than one set of metrics. A possible approach to investigate would be to train genetic algorithms with bigger chromosomes (corresponding to larger sets of metrics); however, this would be relatively expensive, and might introduce classification noise. While such a method might be worth investigating, in the scope of this chapter, we prefer to try a simpler, and hopefully more effective approach, to combine predictions of quality functions.

We considered a larger set of quantile sizes with respect to Table 5.10, and we studied the prediction precision of intersections of such quantiles, which correspond to posts which show very good or very bad quality as predicted by more than one quality function associated to a metric set. We obtained a large set of possible predictive models based on such intersections, which are summarized in Table 5.14.

Each combination of quantile sizes identifies a different set of questions, and with decreasing size of such set one can achieve better precision. It is an expected trade-off between an identified

Metrics	Top-3 $Features_G$	w_G
Trained on T_1, tested on T_2		
M_{SO}	Tags Count	0.75
	Title Length	0.72
	Spaces Count	0.39
M_R	LOC Percentage	0.92
	Coleman-Liau Index	0.88
	Words Count	0.83
M_P	Favorite Votes	0.83
	Down Votes	0.74
	Accepted By Originator Votes	0.50
Trained on T_3, tested on T_4		
M_{SO}	URLs Count	0.93
	Body Length	0.90
	Title Length	0.84
M_R	LOC Percentage	0.98
	Average Term Entropy	0.33
	Automated Reading Index	0.33
M_P	Accepted By Originator Votes	0.98
	Offensive Votes	0.97
	Down Votes	0.93

Table 5.11. Quality Functions Metric Weights for Good Quality Questions

Metrics	Top-3 $Features_B$	w_B
Trained on T_1, tested on T_2		
M_{SO}	Text Speak Count	-0.99
	Body Length	-0.93
	Lowercase Percentage	-0.82
M_R	Gunning Fox Index	-0.87
	Flesch Reading Ease Score	-0.54
	Sentences Count	-0.49
M_P	Approved Edit Suggestion	-0.91
	Moderator Review Votes	-0.90
	Spam Votes	-0.84
Trained on T_3, tested on T_4		
M_{SO}	Text Speak Count	-0.98
	Uppercase Percentage	-0.67
	Title-Body Similarity	-0.46
M_R	Coleman Liau Index	-0.74
	Sentences Count	-0.69
	Gunning Fox Index	-0.61
M_P	Favorite Votes	-0.22
	Approved Edit Suggestion	-0.15
	Moderator Review Votes	-0.06

Table 5.12. Quality Functions Metric Weights for Bad Quality Questions

Metrics	Neutral	w_N
Trained on T_1, tested on T_2		
	Uppercase Percentage	-0.17
M_{SO}	Title-Body Similarity	-0.15
	URLs Count	0.11
	Automated Reading Index	-0.19
M_R	Flesch Kincaid Grade Level	0.44
	SMOG Index	0.45
	Total Badges	-0.02
M_P	Bounty Start Votes	0.02
	Badges-Tags Coverage	0.03
Trained on T_3, tested on T_4		
	Title With Capital Letter	0.17
M_{SO}	Emails Count	0.32
	Tags Count	0.36
	Flesch Kincaid Grade Level	-0.09
M_R	Flesch Reading Ease Score	-0.07
	SMOG Index	0.14
	Total Badges	-0.02
M_P	Up Votes	-0.02
	Close Votes	0.00

Table 5.13. Quality Functions Metric Weights for Neutral Quality Questions.

set of questions and the precision to be obtained. In Table 5.14 we present models with top precision in three different range sizes, from around 1% to 20% of original testing set. We can achieve very high precisions on the intersections of tails. On separated dataset T_2, we can reach precisions as high as 97.4% to identify good questions, and as high as 89.2% to identify bad questions. On the noisy dataset T_4, precision reaches values around 80% for both good and bad questions on smaller portions of the testing set.

5.4.3 Tail-Based Classification

The classification approach proposed in Section 5.4.2 is based on a linear quality function (QF). We combine all the metrics (m_i) described in Section 5.3 by assigning a weight (w_i) in the $[-1, 1]$ interval. Metrics are normalized according to the minimum and maximum value calculated from the dump of September 2013, and range in the $[0, 1]$ interval. The QF is learned to assign negative values for bad quality posts and positive values for good quality posts.

$$QF = \sum_{i=1}^{n} w_i \cdot m_i \quad w_i \in [-1, 1] \quad m_i \in [0, 1] \tag{5.1}$$

We constructed a different QF for each metric set devised in Section 5.3, and we learned each one using genetic algorithms [Gol89] implemented with the open source framework JGAP. We trained the QF on the T_1 dataset, considering two levels of quality where A and B lie in the *Good* set, and C and D lie in the *Bad* set. We trained our genetic algorithm by using a population size of 64 individuals for 20 generations.

Class	Size Range	Quantile Intersection			Size	P
		M_{SO}	M_R	M_P		
Trained on T_1, Tested on T_2						
D	10-20%	0.5	0.5	0.5	22063	80.4%
		–	0.5	0.2	16808	81.0%
	5-10%	–	0.25	0.2	8532	81.5%
		0.25	0.5	0.5	12303	83.5%
	1-5%	0.1	0.5	0.5	5447	85.7%
		0.05	0.2	0.5	1341	89.2%
A	10-20%	–	0.5	0.25	15759	91.3%
		–	0.25	0.5	16729	88.9%
	5-10%	–	0.25	0.25	8293	95.8%
		–	0.25	0.2	6752	96.1%
	1-5%	–	0.25	0.1	3954	96.7%
		0.5	0.25	0.05	1544	97.4%
Trained on T_3, Tested on T_4						
C+D	10-20%	0.5	0.5	0.5	26987	76.4%
		0.5	0.25	0.5	43006	74.3%
	5-10%	0.2	0.25	0.5	14695	78.4%
		0.25	0.5	0.5	24256	76.9%
	1-5%	0.05	0.2	0.5	3965	81.4%
		0.1	0.2	0.5	7255	80.0%
A+B	10-20%	–	0.5	0.2	31694	74.4%
		–	0.5	0.25	40060	74.3%
	5-10%	0.25	0.5	0.25	14286	79.2%
		0.5	0.5	0.25	25712	77.6%
	1-5%	0.25	0.5	0.05	2916	83.2%
		0.5	0.25	0.05	4948	81.6%

Table 5.14. Quantile Intersection Models.

Figure 5.2 shows how we classify posts according to the distribution of a quality function. We consider the left and right tails based on specific quantiles of the quality function distribution. The left tail identifies posts with very bad quality (class D). The right tail identifies very high quality posts (class A). A given model may consider different (and more restrictive) quantiles to identify very bad or very good posts, and thus may be more or less precise to identify high/low quality posts.

Evaluating the Stack Overflow approach

In the public data dump of Stack Overflow many tables and data are missing, *e.g.*, the data concerning review queues and the information concerning the quality score assigned to a post by the system are not disclosed. We collaborated with Stack Overflow to work on this private information. To evaluate the base performance of their approach we introduce two definitions of precision:

Hard Precision (HP): the percentage of posts in the review queue belonging to the class D.

Soft Precision (SP): the percentage of posts in the review queue belonging to the class D and C.

Figure 5.2. Example of tails identification in the quality function distribution.
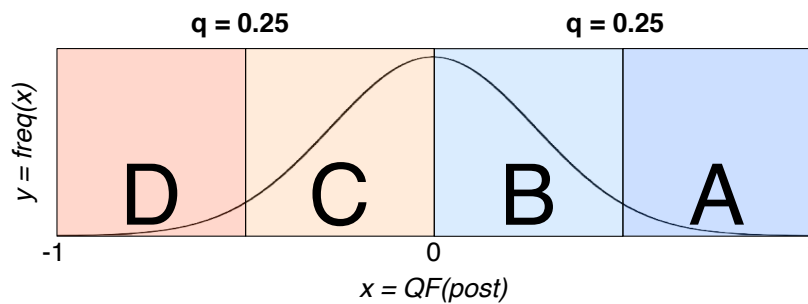
The rationale behind the hard precision is that the review queue should ideally contain low quality posts that need to be closed; the soft precision captures less problematic posts, which are low quality but do not need to be closed or deleted.

Table 5.15. Review Queue (RQ) Distribution for T_2.

Class	RQ Size	Percentage
A	228	6.68%
B	991	29.01%
C	764	22.37%
D	1,433	41.94%
Total	3,416	100.00%

Table 5.15 shows the number of posts classified as low quality by the Stack Overflow approach according to the dataset T_2 we constructed. The Stack Overflow approach has a hard precision of 41.9%, indicating that over 58% of posts in the review queue are of C-quality or better. If we consider the soft precision, the situation improves up to 64.31%, where 35.69% of the contents of the review queue are good posts. As we infer from Table 5.15, the Stack Overflow approach performs better when it comes to identifying high quality posts belonging to class A, while it fails when dealing with middle-high quality posts belonging to class B. With our approach we try to refine the review queue by removing high quality posts.

Table 5.16. Review Queue Models.

Name	Model	RQ Size	A	B	C	D
Mod_{P25}	$RQ \setminus A(M_P, 0.25)$	3,108	166	799	735	1,408
$Mod_{P25,SO30}$	$RQ \setminus (A(M_P, 0.25) \cup A(M_{SO}, 0.05))$	2,650	107	559	664	1,320
$Mod_{R25,P30}$	$RQ \setminus (A(M_R, 0.25) \cup A(M_P, 0.10))$	2,529	106	551	567	1,305
Mod_{P33}	$RQ \setminus A(M_P, 0.33)$	2,552	89	507	657	1,299
Mod_{R33}	$RQ \setminus A(M_R, 0.33)$	2,505	112	544	556	1,293
Mod_{R40}	$RQ \setminus A(M_R, 0.40)$	2,300	78	430	529	1,263
Mod_{P40}	$RQ \setminus A(M_P, 0.40)$	2,421	74	449	641	1,257
Mod_{P40b}	$RQ \cap D(M_P, 0.40)$	2,244	64	393	600	1,187
Mod_{R40b}	$RQ \cap (D(M_R, 0.40)$	1,912	33	251	468	1,160

Table 5.17. Review Queue Reduction with our approach.

Model	HP	SP	RQ Red.	A Red.	B Red.	C Red.	D Red.
Mod_{P25}	45.30%	68.95%	9.02%	27.19%	19.37%	3.80%	1.74%
$Mod_{P25,SO30}$	49.81%	74.87%	22.42%	53.07%	43.59%	13.09%	7.89%
$Mod_{R25,P30}$	51.60%	74.02%	25.97%	53.51%	44.40%	25.79%	8.93%
Mod_{P33}	50.90%	76.65%	25.29%	60.96%	48.84%	14.01%	9.35%
Mod_{R33}	51.62%	73.81%	26.67%	50.88%	45.11%	27.23%	9.77%
Mod_{R40}	54.91%	77.91%	32.67%	65.79%	56.61%	30.76%	11.86%
Mod_{P40}	51.92%	78.40%	29.13%	67.54%	54.69%	16.10%	12.28%
Mod_{P40b}	52.90%	79.63%	34.31%	71.93%	60.34%	21.47%	17.17%
Mod_{R40b}	60.67%	85.15%	44.03%	85.53%	74.67%	38.74%	19.05%

Refining the Review Queue

We refine the review queue using the tails of the distribution. Table 5.16 reports a summary of the different approaches we followed to improve and reduce the low quality review queue.

We see two possibilities to refine the review queue using a quality function: We can remove very good posts (A class) or intersect the review queue with very bad quality posts (D class). Since we have three different set of metrics, we can also combine intersections and subtractions of tails for different QFs originated with different metric sets. Let $\{A, D\}(M_x, q)$ be a tail for class A or D, for a QF learned with the set of metrics M_x, and q be a quantile to identify the tails. We tried different models to refine the review queue and we compared the hard and soft precision we obtained against the one shown in Table 5.15. Table 5.17 shows the result for the best performing models. We can see that there is a trade-off concerning the reduction of the review queue (RQ Reduction), the hard precision (HP), the soft precision (SP) and the percentage of D posts possibly removed. For example, by removing A-class posts originated with the popularity metrics (M_P) QF on the 0.25 percentile, we are able to remove 27.19% of posts belonging to the class A, and *19.3%* of posts belonging to class B, while losing only 3.8% of class C posts and 1.74% of class D posts. We obtain a hard precision of 45.3% and a soft precision of 68.95%, with and effective review queue reduction of 9%. If we increase the percentile with the same approach (*e.g.,* 0.4 percentile), we can remove 65.79% class A post with a loss of 12.28% class D posts in the worst case, increasing the review queue reduction to 29% and the hard precision to 52%. Similar results can be obtained with readability metrics (M_R) where we are able to obtain the highest review queue reduction (44%) while considering the 0.4 quantile and intersecting D-class posts, with a loss of 19% of class D posts and a reduction of 85.5% of class A posts.

5.5   Discussion

We defined more than 40 metrics that capture different aspects of a question at its initial stage. We classified these metrics in three distinct sets concerning (i) readability metrics, (ii) author's popularity, and (iii) simple textual features in use at Stack Overflow.

5.5.1 Decision Trees

Initially, we adopted decision trees, a machine learning approach whose output can be easily interpreted, and we considered two possible datasets: an *ideal* one, where we selected only posts with very low or very high quality, and a *noisy* one. Results of classification with decision trees exhibited poor predictive power: slightly better than a coin-flip. Inspection of decision tree leaves gave us preliminary insights on which metrics influence the quality of significant sets of questions. Overall, the author's popularity metrics better discriminated bad and good posts than the other two sets of metrics, reaching a precision above 70%. According to the public data dump of September 2013, about 6,000 questions are asked every day. A precision of 70% would lead to many misclassified questions to be reviewed that do not need to be closed or deleted from Stack Overflow.

5.5.2 Quality Functions

We considered a linear quality function-based model for each set of metrics and we trained the weights by means of a genetic algorithm. The linear model was built to classify poor quality questions with negative values, and high quality questions with positive values. The models obtained could not correctly classify bad and good questions in general, thus we measured the precision of the classification for questions residing in the tails, that is, the more positive and the more negative value ranges. To this end, we trained our linear model on a subset of 20,000 questions (with balanced bad and good posts) and we tested it against a balanced testing set. To verify the precision of the classification of questions lying in the tails, we adopted two approaches:

1) We took every metric set and we verified the precision in classifying the elements in the tails, by choosing different sizes for each of them (*i.e.,* 1%, 5%, 10%, and 25%). In the ideal data set, the model trained on popularity metrics was able to correctly classify from 85% questions out of 25,841 good posts lying on a bigger tail and up to 90.1% questions out of a tail containing 2,251 posts. On the other hand, the same model correctly classified from 68.4% bad questions out of a tail of 46,016 questions and up to 81.0% bad questions out of a tail of 1,718 questions. Results on the noisy dataset were worse but proportionally similar. While metrics in use at Stack Overflow performed badly, readability metrics seemed to slightly better classify bad questions with a precision of 73.8% on a set of 2,291 posts.

2) We inspected the coefficients of quality functions and obtained insights about which metrics influence good and bad qualities of posts. For example, we found that users who received down votes in the past are more prone to post high quality questions in the future, probably to raise their reputation.

3) By looking at the intersection of the posts classified by each metrics set, we noticed that each set seems to identify different types of bad and good posts. For this reason, we verified the precision of the intersection of these models by varying and mixing the tail sizes (*e.g.,* 50% popularity metrics, 10% readability metrics, 10% Stack Overflow metrics). With this approach, the best model was able to correctly classify 96.2% out of 2,464 good posts and 74% out of 2,230 bad posts. We also identified a model that improves the classification of bad posts up to 85% out of 1,194 questions.

An important insight we derived from the classification models above is that author's popularity metrics are the most effective feature in deciding if a post is of a good quality or not. If we consider that reviewers are users selected inside the community, then having prominence on

the popularity of an author matches the organizational behavior of the community itself. Last but not least, the prediction power given by author's popularity can be complemented by taking into account structural properties of the posts given by the other two sets of metrics.

5.5.3 Refining Low Quality Review Queue

Following and extending the discussion on the quality function, we can see that popularity metrics and readability metrics are the most useful metric sets to refine the low quality review queue. Readability metrics are effective alone and in combination with popularity metrics (Table 5.17), thus connecting readability to good quality posts, and both metric sets complement the simple textual features currently in use at Stack Overflow. Thus, we recommend to take into account popularity and readability metrics in the quality assessment of posts. For example, while an automatic refinement system could be long-term goal, a simple recommendation system could be more effective in the short term to help reviewers to discard posts that score high in popularity and readability metrics, or review first posts that score particularly low.

5.6   Threats to Validity

Construct Validity

Threats to construct validity are concerned with whether what one measures is what one intends to measure. In our case, there could be several reasons why the considered quality of the questions is incorrect. We rely on the judgement of the SO crowd to differentiate the quality of questions, which is a potentially error-prone process. In fact, the perceived quality might be different from person to person, and might be based on different definitions. This issue is alleviated by the fact that we also manually examined more than 100 questions not only to get insights on their features, but also to verify whether the choices made by the users were reasonable. Moreover, Stack Overflow relies on the same criteria.

Another issue regarding quality is the definition of the class A in our dataset (Table 5.1): 'very good' questions. We defined 'very good' questions as those with a Stack Overflow score higher than 7. We chose this threshold to obtain balanced datasets, and as a reasonable trade-off between choosing a too high value (which would have only included questions regarding trending topics) and a very low one (which would have included questions only inspected by few users).

Finally, the choice of balanced dataset could have impacted the results of the machine learning models. Nevertheless, this should not be a problem when enough training data is available.

Statistical Conclusion

Threats to statistical conclusions concern the fact that the data is enough to support claims. We considered statistically significant samples in our experiments; this was possible because we relied on the crowd assessment and not on manual inspections of questions.

External Validity

External validity threats are concerned with the generalizability of results. The approaches we tried may show different results when applied to a Q&A website other than Stack Overflow. To alleviate this issue, we chose to include questions related to any valid topic in the technical forum, thus including a very large population. An evaluation of our approach that involves other Q&A websites could measure the effect of this threat.

5.7 Conclusions

We devised, implemented and illustrated an approach to classify question quality, and in the same way understand what fundamentally influences and characterizes it. We devised three sets of metrics that capture both textual features of a question and the reputation of the user who asked it. Together with these metrics, we constructed two datasets from a Stack Overflow data dump which captured what the community identifies as a question's quality. We began by devising two types of quality for a question in Stack Overflow (*i.e.,* 'Good Quality', 'Bad Quality') and we then extended our definition to four level of quality (*i.e.,* 'Very Good', 'Good', 'Bad', 'Very Bad'), by imposing some empirical thresholds, based on data balancing.

We also conducted a twofold empirical study aimed at classifying Stack Overflow questions' quality and understand how the metrics we devised influence it. In the first part, we used a machine learning algorithm to infer decision trees. In the second part, we used genetic algorithms to learn linear quality functions that describe a question's quality. We encoded quality as a function classifying bad quality questions with negative values, and high quality questions with positive values, thus representing different shades of quality where the extremes represent very bad and very good levels. By analyzing the tails of such quality functions, and in particular intersections of them, we were able to reach prediction results that can be beneficial for an automatic quality classification approach, and confirm that popularity metrics are the best predictors, and identify which specific metrics strongly influence good or bad quality.

Reflections

This chapter shows how the estimation of the quality of a development artifact, like a Stack Overflow discussion, requires a heterogeneous analysis of the information that goes beyond the textual part of the artifact.

Even when focusing on the narrative of the artifacts, the simple analysis of the terms, or characters, as performed by the Stack Overflow team, poorly performs if compared to more elaborated analysis of the narrative, like readability indexes, or to complementary information like popularity metrics.

It is worth noting that according to the studies performed in this chapter, the best results are obtained by combining different type of information together. For example, the combination of readability and popularity metrics led to better performance, thus highlighting once again that the heterogeneity of the information can be exploited to improve current approaches.

In the next part we focus on extracting and modeling heterogeneous data from development artifacts, breaking the boundaries imposed by the narrative, and discovering heterogeneous elements like code elements immersed in it. We will see how this novel modeling phase steps out from the canonical, textual oriented, analyses of development artifacts, and how it can be leveraged to devise applications which preserve the multifaceted nature of the information.

Part III

Parsing and Modeling
Unstructured Data

Automated Multi-Language Parsing and Modeling of Software Engineering Artifacts

Extracting and analyzing non-textual elements in development artifacts is a non-trivial task. Indeed, code fragments unavoidably overlap with narrative, causing potential undesired matches. For example, many words in natural language can be valid identifiers for a programming language (*e.g.,* valid type names, identifiers).

Off-the-shelf parsers like the Eclipse JDT are not designed to extract code elements from the narrative, and even if the code is isolated (*e.g.,* in a `<code>` tag), such parsers generally fail to parse incomplete code fragments into a meaningful and correct abstract syntax tree (AST). Another challenge concerns the presence of multiple languages like JSON and XML, or structured elements like stack traces. Off-the-shelf parsers do not even consider these options. A viable way to overcome these limitations is to build an island grammar [Moo01, BCLM11], which is able to separate constructs of interest (*i.e.,* islands) from the uninteresting constructs (*i.e.,* water). We leverage island grammars to construct an analyzer capable of disentangling structured (*e.g.,* Java, JSON) and semi-structured contents (*e.g.,* Stack Traces) from the narrative.

This chapter describes an approach to automatically model the artifact contents through a Heterogeneous Abstract Syntax Tree (H-AST) that represents, in a unique structure, both textual fragments, interchange formats and languages of an artifact, thus enabling semantic modeling of its contents.

Structure of the Chapter

In Section 6.1 we present the foundations of our approach and its challenges. In Section 6.2 we investigate the two research questions, we describe a technique based on automated random testing to evaluate its effectiveness, and we apply it in the context of Stack Overflow discussions. In Section 6.3 we conclude the chapter.

6.1 Multilingual Island Grammar

Our approach targets both web resources (*e.g.,* Stack Overflow discussions) and non-web artifacts. The former are normally represented and stored as HTML documents or fragments. Often the `<code>` elements do not contain syntactically valid source code. Therefore, we treat the contents of `<code>` elements as document text, rendering the whole HTML and considering the resulting

text. Our approach can also be applied to non-web resources like plain text emails, since it works on the document's representation.

Given this context, we extend an approach based on island grammars [Moo01, BCLM11] to disentangle code constructs from the natural language narrative. In the literature, island grammars have been used to extract Java constructs of interest from mailing lists [BCLM11]. We extend it to enable the modeling of multi-language code fragments (*i.e.,* JSON, XML, and stack traces) supporting both complete (*e.g.,* class and method declarations) and partial constructs (*e.g.,* variable declarations, statements), even when they are interleaved with narrative. Furthermore, we support incomplete constructs like method declarations without return type or without body.

We proceed to describe the basic features of our approach, how we leverage it to extract and model Java fragments, and how we extended it to support multi-lingual contents.

6.1.1   Island Grammars with Parsing Expression Grammars (PEGs)

An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water) [Moo01]. An island grammar has the following structure:

$$Artifact \leftarrow (Island \,|\, Water)^+$$
$$Water \leftarrow Any^+$$

Conceptually, *Island* is a rule that matches constructs of interest, while *Water* matches all the remaining contents. In our context, we aim to extract and model code fragments as islands and consider the rest, *i.e.,* the water, as narrative.

For convenience, we implemented our grammar with Parsing Expression Grammars (PEGs). PEGs [For04] are a type of grammar formalism that is inherently not ambiguous. If a match in grammar rules happens, that match produces exactly one parse tree. PEGs achieve this goal by considering the order (priority) of the rules to be evaluated.

For example, consider a rule that describes an alternative between two possible strings, where one of the strings is contained in the other, *e.g.,* "int" and "integer". In a context free grammar (CFG), the rules *IntFirst* ← **int** | **integer** and *IntegerFirst* ← **integer** | **int** are equivalent, and the input string "integer" would match in both cases. This is not the case for PEGs, since order defines priority of rules, and the input string "integer" matches for the *IntegerFirst* rule but fails *IntFirst*. Priority is fundamental in the definition of islands, making it more challenging in defining the grammar, but essentially allowing linear parsing time [For04]. We implemented our grammar by using the PARBOILED[1] framework for the Scala programming language.

6.1.2   Island Grammar for Java 8

Our grammar supports all Java 8 constructs, including lambdas and type annotations. A proper definition of islands is the main challenge in our approach. Code fragments can be ambiguous with natural language, and thus rules must be methodically devised. Clearly, the higher the complexity of a grammar rule, the lesser the ambiguity with natural language that it can generate. Thus, constructs like type declarations (*e.g.,* classes) or non-empty blocks can be parsed and modeled as islands without restrictions. However, these constructs are too coarse grained.

For example, in Stack Overflow users tag code fragments in various ways. Generally, fragments are leveraged to focus the attention of the reader on the part of the code where the problem

---

[1] `http://parboiled.org/`

is supposed to be located. Therefore, instead of reporting whole classes, users tend to report simpler constructs, *e.g.,* methods, variable declarations, statements, or type names. Moreover, these constructs are often incomplete (*e.g.,* missing body for methods), making the disambiguation with natural language even harder. In the following we pick one example for each category of constructs that our approach handles, focusing on the precise methodology to parse it with PEGs to extract it from narrative, and the way we precisely model with an H-AST.

Fragments and Sub-constructs

Our first example focuses on method declarations. Consider the two following (simplified) rules for method and constructor declarations:

$$MethodDecl \leftarrow Modifier^* \: Type \: Identifier \: Args \: Body$$
$$ConstructorDecl \leftarrow Modifier^* \: Identifier \: Args \: Body$$

```
public Foo() { } // a constructor
private void aMethod() { } // a method
```

Listing 6.1. Example Declarations

The two cases are not problematic since they are complete and do not require grammar modifications, even when immersed in the narrative. If modifiers are present, there is no ambiguity on deciding that the first declaration is a constructor, and that the second declaration is a method (since it has a return type). If visibility modifiers are absent, things are more complicated.

```
Foo() { } // a constructor
void aMethod() { } // a method
```

Listing 6.2. Declarations without Modifiers

Listing 6.2 shows a case where the constructs are missing modifiers. From a grammatical point of view the constructs are valid, since modifiers are not mandatory. Due to the PEG parsing prioritization mechanism, the method declaration rule must take precedence over constructor declaration in the island definition; otherwise, the parser would prioritize `aMethod` as a constructor, ignoring the return type. Furthermore, when visibility modifiers (*i.e.,* public and private) are absent, ambiguity arises. Consider the same two declarations interleaved with narrative as in Listing 6.3.

```
Consider the constructor Foo() { } and the
method void aMethod() { }.
```

Listing 6.3. Declarations Immersed in Narrative

In this case, by following the standard Java grammar, there is no way to distinguish between a constructor and a method, since the word *constructor* is a valid Java identifier and thus it is

a syntactically valid return type.

$$MethodDecl \leftarrow Modifier^+ \, Type \, Identifier \, Args \, Body \mid$$
$$Type \, MethodIdentifier \, Args \, Body$$

$$ConstructorDecl \leftarrow Modifier^+ \, Identifier \, Args \, Body \mid$$
$$ClassIdentifier \, Args \, Body$$

To solve this issue, we must take into account the lexical structure of identifiers, since we cannot rely on pure syntactical aspects of the Java grammar. In fact, lexical constraints can be enforced to help disambiguation of such cases; in other words, ambiguity can be mitigated by enforcing naming conventions.

In Java, naming conventions discriminate constructors from methods. The former have a capital letter at the beginning of the name (since they share the same conventions as class names), while the latter start with a lowercase letter and implement the camel case convention whenever their name is composed of two or more words.

Incomplete Productions

The second example concerns incomplete productions, like incomplete declarations. Users tend to focus on the important aspects of code by hiding the parts they deem irrelevant for the discussion. For example, method bodies can be removed, and an ellipsis ("...") can be used instead of the actual body implementation, or to strip the parameters declaration (*e.g.,* `int aMethod(...)`). Figure 6.1 shows an example.

Figure 6.1. Example of Stack Overflow discussions with code tagged by users

The author is asking about an inheritance problem with the `JPanel` class, and reports just the partial signature of the classes `Entity`, and `TextEntity`, without the body.

The incomplete class declaration is a good example of the challenge to face when designing the island grammar. The incompleteness of the class should be managed to avoid ambiguity.

$$ClassDecl \leftarrow ... \mid Modifier^* \; \texttt{class} \; Identifier \; \texttt{extends} \; Type \mid$$
$$Modifier^+ \; \texttt{class} \; Identifier \mid \texttt{class} \; ClassIdentifier$$

The listing above shows some of the rules to parse incomplete class declarations. Incomplete class declarations can appear in even simpler forms like `class SomeTypeName`, without modifiers. In this case, the ambiguity with English is relatively high. If no restrictions are enforced on the identifier lexical structure, whatever word comes after the keyword `class` in a text would be considered a class name. Once again, naming conventions can help, and we can consider only identifiers respecting Java conventions for types.

Incompleteness must be taken into account on the modeling side. Listing 6.4 shows a simplified version of a class declaration in the H-AST.

```scala
case class ClassDeclarationNode(
 val modifiers: Seq[ModifierNode],
 val identifier: IdentifierNode,
 val typeParams: Option[TypeParamsNode],
 val superTypes: Option[TypeNode],
 val interfaces: Option[TypeListNode],
 val body: Option[ClassBodyNode])
```

Listing 6.4. Modeling a Class Declaration

Contrary to a normal Java AST, in our H-AST the class body is modeled as an optional construct.

In-Paragraph Fragments

The hardest fragments to disentangle from natural language concern in-paragraph code, where they take the role of parts of the discourse in the narrative. An example can be found in Figure 6.1, where the method invocations `getWidth()` and `getHeight()` are the subjects of a sentence. Moreover, Figure 6.1 also contains non-tagged class names like `TextEntity` and `JTextArea`, and it is also possible to find a fully qualified identifier (*e.g.,* Java.util.Date), and entire statements. Our approach considers these cases as well, and is also able to parse and model them.

As in the case of incomplete class declarations, the lexical structure of words, and in particular the compliance to naming conventions, can be used to retrieve and identify in-paragraph fragments. For example, naming conventions can be leveraged to identify type names like `TextEntity` and `JTextArea` in Figure 6.1. We devised additional rules to deal with specific in-paragraph fragments.

Qualified Identifiers

According to the Java grammar, a qualified identifier is a sequence of identifier separated by a dot. Unfortunately, this structure is highly ambiguous with the english grammar, in particular when a dot separates two periods. Consider the sentence "Hi Mike. How are you?", the rule would match the qualified identifier `Mike.How`, since spaces are normally ignored by Java parsers. To

mitigate these false positives, one simple solution is not to allow spaces between identifiers and the dots, which are very rarely left when mentioning qualified identifiers.

Method and Class Names

Class names are characterized by starting with an uppercase letter and being named using camel case notation. Extracting class names with just the first letter as uppercase would introduce noise, identifying as a potential class name every word at the beginning of a sentence. For this reason, we require class names to have at least a second word in camel case notation, *e.g.,* `TextEntity` [BWYS11, BLR10, BCLM11, RR13]. Similarly, a in-paragraph method name is an identifier that respects the Java naming convention, or exhibits features of C-like naming convention. In other words, we extract likely method names that start with a lowercase letter, and contain at least a case change (*i.e.,* `aMethodName`), or an underscore (*i.e.,* `a_method_name`) [BLR10, BWYS11, RR13].

Method Invocations

Method invocations have a peculiar structure, but they still maintain a not negligible level of ambiguity with English. The presence of elements like type arguments (*e.g.,* `write<String>("a")`) clearly distinguishes them from natural language. In some other cases, a strict qualified identifier followed by a method name respecting naming conventions for methods is enough (*e.g.,* `object.aMethod()`). However, our approach also considers the case where the name is composed of a single word, with no camel case (*e.g.,* `size`). In this other case, we resort to the arguments of the invocation to understand if the fragment is a likely method invocation.

We adopt the following heuristic: If more than one argument is provided, then the fragment is considered an invocation. If there is only one argument, we discriminate by its type. Every argument type but qualified identifiers are considered safe. If the type is a qualified identifier, then it needs to provide at least two identifiers separated by a dot. Allowing a lone identifier would introduce noise. For example, in the sentence "the color of the apples (red) is not yellow" the grammar would match `apples (red)` as a method invocation.

Listing 6.5. Example of island with lakes

```
@Entity @Table(name = "shops")
public class Shop {
  ...
  @ManyToOne(fetch=FetchType.EAGER)
  @JoinColumn(name = "shop_type_id")
  private TypeShop typeShop;
  ...
}
```

Island with Lakes

According to Moonen [Moo01], *island with lakes* are other types of constructs that can be enabled by parsing island grammars. An island with lakes is a valid construct that may contain water (lakes), in our case natural language narrative or other punctuation marks. Listing 6.5 shows an example where "`...`" would be a lake.

The sample is taken from a Stack Overflow question[2] and it represents a typical way for users to strip away unneeded code. Our grammar supports these types of constructs to avoid missing relevant information, like the class reported in Listing 6.5. For the sake of simplicity in engineering the grammar, we limited the support to blocks (*e.g.,* method body, class body) containing water. In doing so, we can support either code stripping and minor errors in statements (*e.g.,* missing semicolon). A complete class or method construct like Listing 6.5 is parsed as complete construct even if it contains minor errors inside its body. Allowing lakes complicates the modeling side of the H-AST. Essentially, we allow textual fragments to be interchangeable with statements in blocks (*e.g.,* in method bodies) and with member declarations in classes.

I am migrating from xml based spring configuration to "class" based configuration using the corresponding @Configuration annotation.

I came across the following problem: I want to create a new bean, which has a reference to another (service) bean. Therefore I autowired this class to set this reference during bean creation. My configuration class looks as follows:

```java
@Configuration
@ComponentScan(basePackages = {"com.akme"})
public class ApplicationContext {

    @Resource
    private StorageManagerBean storageManagerBean;

    @Bean(name = "/storageManager")
    public HessianServiceExporter storageManager() {
        HessianServiceExporter hessianServiceExporter = new HessianServiceExporter();
        hessianServiceExporter.setServiceInterface(StorageManager.class);
        hessianServiceExporter.setService(storageManagerBean);
        return hessianServiceExporter;
    }
}
```

But this doesn't work, because the causes a BeanNotOfRequiredTypeException exception during startup.

```
Bean named 'storageManagerBean' must be of type [com.akme.StorageManagerBean], but was actua
```

The StorageManagerBean is annotated with an @Service annotation. And the xml based configuration worked as expected:

```xml
<bean name="/storageManager" class="org.springframework.remoting.caucho.HessianServiceExport
    <property name="service" ref="storageManagerBean"/>
    <property name="serviceInterface" value="com.akme.StorageManager"/>
</bean>
```

So, I've no idea what I am doing wrong and why spring tries to auto wired a proxy class.

I appreciate your help.

Figure 6.2. A Stack Overflow discussion with multilingual contents

---

[2] http://stackoverflow.com/questions/26564952

6.1.3 Multilingual Support

Figure 6.2 exemplifies how different languages can be mixed together in the same discussion. Good examples are Android and SWING applications, where Java code is often accompanied with XML configuration files, and both are fundamental to understand the dynamics of either a GUI or an app. Similarly, JSON is often associated to Java classes via serialization and deserialization (*i.e.,* POJO objects), or when implementing RESTful web services as in JAX-RS[3]. Cross language references like the aforementioned ones are a *de facto* standard for certain platforms and domains.

One additional construct that must be taken into account when dredging the type of information contained in a discussion are stack traces. Traces provide meaningful information to backtrack bugs and to identify where the fault comes from. The relationship between stack traces and Java constructs can be exploited also at the discussion level: The information reported by traces is often used by developers to provide additional information to explain the error they are facing.

All these constructs are sufficient to model the information provided by many Stack Overflow discussions about Java. Modeling this type of information can help in backtracking and connecting traces to specific parts of the code snippets extracted from the discussion, and enable approaches that can exploit these heterogeneous relationships. Thus, we target four different sets of constructs: Java, XML, JSON and stack traces.

Lakes in Other Languages

The extension to multiple languages of island with lakes cannot be applied to XML and JSON in the same way that we applied it to Java blocks. When dealing with JSON, given the simplicity of the constructs, we do not allow lakes. The effect is that JSON fragments are split into multiple JSON sub-fragments when some narrative is present inside an element. The lake problem is treated differently when dealing with XML and stack traces. The XML grammar allows free-form text between tags, and whatever does not respect the grammar of the tags, is considered as pure text by definition. On the other hand, allowing lakes in a stack trace would lead to an unmanageable level of ambiguity where pieces of two stack traces would be considered as one by the grammar.

Priorities Between Languages

Another challenge lies in the priority given to the different constructs from different languages. Languages do not only coexist in the same text, but they can also have a containment relation. Listing 6.6 shows a simple *String* variable declaration where the initialization string contains a valid XML declaration.

Listing 6.6. Example of ambiguous code for heterogeneous island grammar

```
String xmlString = "<employee>" +
 "<name>John</name>" +
 "<surname>Doe</surname>" + "</employee>";
```

Depending on the priority given to each construct, the fragment can be parsed as a variable declaration, or as an XML element with water at the beginning (`String xmlString = "`), and at

---

[3]`https://jax-rs-spec.Java.net/`

the end (`";`). The other quotes and plus symbols are considered, respecting the XML grammar, as text contained in the top element (*i.e.,*`<employee>`).

Depending on the granularity of the grammar rules, this problem may manifest itself with other constructs. For example, in the Java grammar, we capture strict type names and strict method invocations respecting the Java naming conventions. If a stack trace is encountered first, and the Java grammar has the highest priority, the trace will be parsed as a set of types and method invocations with water all around. On the other hand, since stack traces do not allow lakes in the grammar, and have a singular construction, they do not overlap with any other construct.

According to these examples we devised these priorities for the constructs in the following order: JSON, stack traces, Java, and XML. The outcome of the approach devised by Bacchelli *et al.* [BCLM11] isolates and extracts constructs in textual form. The pure extraction of constructs is not enough to model the contents. Instead, our approach aims at transforming the contents in a traversable representation close to an abstract syntax tree (AST) provided by a compiler. Having an AST-like structure helps in defining the structure of the contents. As previously discussed, our parser mixes different grammars. For this reason, we use the acronym H-AST to identify an AST-like node with heterogeneous nature.

Language in Language Support

The contents of string literals in Listing 6.6 is not just text, but as we already pointed out, it is perfectly valid XML. As it often happens in code fragments, string literals and comments may contain interesting structured fragments, like qualified identifiers or full commented statements. Thus, we recursively invoke the island parser on the contents of string literals and comments. We enriched their H-AST with a special *contents* field that represents the H-AST of its contents as parsed and modeled by our approach. We also support language-in-language analysis in XML text nodes and attribute values, as well as in JSON strings, that often may contain qualified identifiers, *e.g.,* when such interchange formats are used for configuration.

6.1.4   Putting Everything Together

Table 6.1 shows some metrics for the rules composing the island grammar and the H-AST implementation.

Language	Productions	H-AST Classes
Java	321	116
XML	26	10
JSON	15	8
Stack Traces	21	6
Java Strict	60	-
Java Incomplete	20	-
Island Parser	124	3
Total	587	143

Table 6.1. Grammar Information.

Java is the most complex in terms of productions and classes in the H-AST, if compared to the other supported languages. In Table 6.1, there are two main parts. In the top half, the table shows metrics concerning each supported language. In the bottom half, the table shows some

metrics involving rules to catch and model incomplete and strict Java constructs, and to mix all the languages together. Incomplete and strict Java constructs do not add specific H-AST nodes, since Java H-AST nodes are reused in the respective productions. Instead, the island parser itself obviously adds two nodes: `TextFragmentNode` and `StructuredFragmentNode`, that represent water and code fragment islands, respectively.

The size of both the single language grammar and their combination in the island grammar is not negligible and requires an in depth testing phase to discover and unveil engineering errors and issues arising from ambiguity with natural language.

6.2   Evaluating the Island Grammar and Model Construction

Our approach consists of an island grammar for code-related artifacts that provides full-fledged modeling with a H-AST. In this section, we focus on the evaluation of the grammar, *i.e.,* its ability to correctly reconstruct a H-AST, and the ability to disentangle code fragments in practical settings.

A possible method to validate our approach may leverage the creation of a reference dataset. Ready-made datasets for this goal do not exists. Thus, testing island grammars generally resorts to manually tagging the contents of code-related artifacts and verify that the parser can produce the same tagging from the untagged text. Even though it would be possible, with a non negligible endeavor, to annotate even hundreds of artifacts, the result would still be a non-representative dataset, an unrealistic approximation of the real performance of our approach.

As a workaround, one could think of using the human tagging provided by a Stack Overflow discussion, where each post provides a HTML tagging created by its owner. The parser could be tested by verifying that the same tagging can be achieved from the rendered text. Unfortunately, human tagging in Stack Overflow is not reliable, and once again, this solution would boil down to manually checking a selected set of discussions whose tagging is correct. Another significant aspect to consider concerns the model created by the parser. If the manual tagging is performed with a boolean *"code/text"* flag, there is no way to verify the correctness of the H-AST.

We boil down the evaluation of our approach in three parts. First, we present an approach to automatically test the grammars in isolation by means of guided random testing (Section 6.2.1). Second, we compare its applicability to implement part of the approach by Rigby and Robillard [RR13] for which a reference dataset is available (Section 6.2.2). Finally, we consider the ability to disentangle structured fragments in the practical setting of a subset of Stack Overflow discussions (Section 6.2.3) where our grammar can identify fragments in isolation.

6.2.1   Testing Language Grammars In Isolation

In this section we tackle the first research question RQ1:

> *How can we effectively evaluate our approach in the context of its ability to reconstruct a correct model in the form of an H-AST?*

As in our case, when off-the-shelf parsers cannot be used, the grammar for each supported language must be written from scratch. This unavoidable phase can easily lead to errors, in particular when a H-AST is also generated. On one hand, retrieving a benchmark for a language like Java would not be a problem. However, the only solution to verify both the robustness of the grammar and the H-AST would be to compare it with an oracle. For example, we could use the Eclipse JDT as reference, check that they correctly parse the same samples in

the benchmark, and compare the JDT AST with our H-AST. However, the nodes of these two structures are likely to be implemented in a significantly different manner, for example in the case of incomplete constructs, and an interpreter to match the two should be constructed. Instead of relying on an oracle, we leverage the approach of QUICKCHECK [CH00], based on *random testing*, as implemented by the tool SCALACHECK[4], which we briefly describe here in our context.

As with any testing approach or tool, the user needs to provide an automatically checkable criterion or property. Listing 6.7 shows the reference correctness property, expressed in the internal domain specific language of SCALACHECK.

```
forAll { hast: HAST =>                    // 1
  val rep = hast.toCode                   // 2
  val parsedHast = parser.run(rep)        // 3
  parsedHast == hast                      // 4
}                                         // 5
```

Listing 6.7. Correctness Property

The property must be read as: (1) for every possible H-AST (`hast`), (2) considering a representation as code (`rep`), (3) the resulting H-AST obtained with the parser (`parsedHast`) (4) must be equal to the original H-AST. To test the validity of the property, SCALACHECK checks that it holds for a large number of cases, that are provided by generators: A *generator* is a mechanism to randomly generate an arbitrary structure.

Generator Construction

Given each rule of a grammar, a generator can be devised to create H-AST nodes respecting that rule's constraints. As grammar rules, generators can be combined together to fulfill the requirements of more complex rules. Listing 6.8 shows a simplified H-AST generator for Java class declarations.

```
def genClassDeclNode(size: Int) = for {
    modifiers <- genModifiers
    identifier <- genIdentifierNode
    superType <- genOptReferenceTypeNode
    members <- genMembers(size)
  } yield ClassDeclarationNode(...)
```

Listing 6.8. Example Generator for Class Declarations

The generator recursively requires a generator for modifiers (*e.g.,* `public`), for an identifier that represents the class name, an optional super class, and the class members (*e.g.,* methods and fields). This is also a *sized* generator: The size defines the maximum depth of a generated construct, that is, the maximum depth of the H-AST. A sized generator enables SCALACHECK to first explore smaller H-ASTs. This choice can be justified with the *small scope hypothesis* [Jac12]: Essentially, the hypothesis states that most bugs have small counterexamples, *i.e.,* if the checked assertion is invalid, it is very likely that it can be found in a small sized example, namely a small H-AST.

---

[4]https://www.scalacheck.org

XML is another example to explain this rationale. In XML there are three types of constructs: composed tags (*i.e.,* `<tag>...</tag>`, single tags (*i.e.,* `<tag/>`), and text. The first ones are recursive, and they can contain other sequences of every type of constructs, while the others are leafs. As a result, the generator creates text and single tags when the size is equal to one. When the size is greater than one, recursion is allowed, and the generator generates composed tags.

All the generators are written with a dedicated SCALACHECK internal domain specific language to support random testing and property checks.

Verifying the Parser Output

To ensure the correctness of the language grammars in isolation, we went trough a test-and-fix cycle for each language involved. This iterative process allowed us to unveil hidden bugs both in the grammar (*i.e.,* missing and incorrect rules) and in the model (the H-AST construction). For example, we discovered that our parser was discarding Java annotations in the model even though they were correctly parsed by the grammar.

The random-test approach aims at exploring the generable constructs space as much as possible. For this reason, the ideal approach is to generate a large amount (*e.g.,* hundreds of thousands) of tests for each language. Table 6.2 shows some basic metrics involving the generated data sets for the grammars of Java, XML, JSON, and the Stack Traces.

Language	# of Tests	Size of Tests
Java	1M	8.1GB
XML	1M	4.0GB
JSON	1M	4.3GB
Stack Traces	1M	2.3GB

Table 6.2. Random Testing Results for Grammars

Reaching such a large amount of test cases provides enough confidence of the correctness of our grammars, and of the H-AST reconstruction.

6.2.2 Comparison with State of the Art

We proceed the evaluation of our approach by tackling RQ2:

> *To what extent can our approach correctly disentangle structured fragments from narrative in a practical setting like Stack Overflow discussions, and how does it compare to competing approaches?*

The common technique to analyze and extract interesting structured fragments from software artifacts containing narrative is the use of ad-hoc regular expressions.From a conceptual point of view, using full-fledged parsing enables exact detection and real parsing of recursive structures like blocks and expressions, which is not possible with plain regular expressions. However, to frame our contribution and its potential, we compare our approach to an existing competing approach. To this end, we compare our approach with the work of Rigby and Robillard [RR13], for which a reference, manually generated dataset exists. A preliminary part of the approach implements an island grammar-like approach by using regular expressions, aimed at extracting the following Java code elements: package names, method names, types, annotations, and fields (and thus not recursive constructs like blocks, complex statements, expressions). For the extractor in the reference dataset, authors report a precision of 0.92 and a recall of 0.90.

Since our approach retrieves and models any possible valid Java construct and even incomplete ones, to establish a comparison we need to implement a visitor (for which our API gives full support) to collect the interesting elements from the H-AST. A first run on the reference dataset reported a relatively low precision and recall, respectively 0.79 and 0.86.

Such a different outcome required an inspection of our results, and highlighted a set of issues and significantly different assumptions. It may seem trivial, but it is not immediate what it means to extract elements like types in a code fragment. In fact, types are *recursive*: A generic type, like `Map<String,Date>` contains itself parametric types, *i.e.,* `String` and `Date`. Should these be reported in the dataset? We assumed that at first, but the dataset was inconsistent: Sometimes parametric types were reported, other times they were not. We found similar issues with array types and internal classes. Moreover, we found that the dataset contained classes marked as packages, and that fields did not contain *declared* class fields, but only constant and enum type's fields as likely extracted from qualified identifiers (*e.g.,* `Day.SUNDAY`). Finally, class literals (*e.g.,* `Object.class`) were incorrectly reported as fields.

After taking into account all these issues, the precision of our approach increased to 0.94 and recall increased to 0.93. When inspecting the remaining false positives and negatives, we found that essentially they are in-paragraph cases where a purely lexical/syntactic approach (like ours and like regular expressions as well) cannot be used to discriminate if elements are structured fragments or not. These cases contain, for example, types composed by a single word and mixed with the narrative, that are ambiguous with any other natural language noun starting with uppercase at the beginning of a sentence.

This comparison does not completely evaluate our approach. In fact, we can collect other potentially interesting code fragment to support more complicated analyses. For example, one could collect variables, formal parameters, any complex statement, to implement type resolution or any other complex code analysis technique.

6.2.3 Practical Island Grammar Testing

Stack Overflow is a source of ready-made natural language contents and tagged source code that is a potential candidate as a resource to better evaluate our approach. The data provided by Stack Overflow would allow to create more realistic cases by harnessing the combination of human-generated narrative and code. However, as explained in Section 6.1, it is not possible to completely trust the fragments tagged with the `<code>` tag (see Figure 6.2). In the following we tackle this issue, and we present a methodology to mitigate this problem, test our approach in a practical setting, and furtherly analyze RQ2.

```
<p>But <code>getWidth()</code> and
<code>getHeight()</code> returns 0.
Is it a problem with the inheritance
or the constructor?</p>
```

Listing 6.9. Fragment with Tagged Code

Fragment Extraction

The contents of Stack Overflow are tagged with a subset of HTML5. Tagging of code elements can be performed by using the `<code>`, either at the top level of the post (out-paragraph) and inside any other HTML tag (in-paragraph). By analyzing the DOM of a post, we separate code-tagged elements (*i.e.,*`<code>`) from the rest of the contents. In the end, each Stack Overflow post

is fragmented in tagged textual and code parts. Each top-level node in the body of the post is treated separately.

Consider the fragment in Listing 6.9. The contents of the paragraph are collected until a `<code>` is encountered, and marked as textual. Then, we extract the contents of `<code>`, and we mark it as code, and we keep repeating this process until the end of the input. In this case, the fragmentation would generate the sequence *"But "*, *"getWidth()"*, *"and "*, *"getHeight()"*, and *" returns 0. Is it a problem with the inheritance or the constructor?"*. This process is applied to every post of a discussion tagged as `<java>` in Stack Overflow. In total, we extracted 18,993,221 sub-fragments.

Tagging Agreement

After isolating fragments from Stack Overflow discussions, we need to check that fragments likely respect the tagging assigned by the users. Even though there is a degree of uncertainty about the behavior of the island grammar in ambiguous cases, in Section 6.2.1 we extensively tested the grammar of each language implemented, guaranteeing a reasonable level of confidence for complete constructs. Furthermore, the huge amount of fragments to be analyzed mitigates and distributes the possible ambiguity errors, still providing a realistic approximation.

The island parser can be used to estimate the code coverage of the fragments, that is, the percentage of character parsed as valid code elements out the total number of characters of the fragments. The calculation for the tagging agreement is relatively simple: If a fragment is marked as text, and it has 0% code coverage, the tagging agreement would be 100%, or 0% if the code coverage is 100%. The dual holds for fragments marked as code: agreement is 100% with 100% code coverage, and 0% with 0% code coverage.

Disentangling Stack Overflow Posts

The island grammar can be tested by analyzing to what extent it is capable of disentangling natural language from code elements in a subset of Stack Overflow posts. Having extracted and analyzed the fragments of every Stack Overflow discussion concerning Java, it is possible to select the posts whose tagging reaches full (*i.e.,* 100%) agreement either for text and code elements. Assuming enough confidence on the correctness of our approach, this analysis eliminates unavoidable ambiguity cases where the contents are wrongly tagged (*i.e.,* non tagged code elements within narrative), by selecting the ones that fully agree with their tagging when parsed in isolation.

The main idea is to select these posts, their fragments (that can be parsed in isolation), and verify that the island parser can reconstruct the human tagging when all fragments are merged together. The process to follow is conceptually similar to the one used in Section 6.2.1:

1. for each fragment, we parse it and create the H-AST;

2. we combine all the H-ASTs in one sequence;

3. we merge all the raw text of fragments to create one unique document;

4. we parse the document with the island parser to obtain another sequence of H-ASTs;

5. we verify that the two sequences are identical.

In the Stack Overflow dump of March 2016[5] there are 863,500 posts whose tagging has perfect agreement. We run the island parser by following the aforementioned process. Table 6.3 shows the results of the disentangling process.

Disentanglment	Posts	Percentage
Success	823,866	95.41%
Failures	39,634	4.59%
Total Posts	863,500	

Table 6.3. Disentangling Stack Overflow Results

The island parser is capable of correctly disentangling about 95.41% of the posts. When we inspected the results, we found that the failures were due to a complex rule matching and grouping sequences of isolated statements. We found that some constraints on the first statement of these sequences, that are used to avoid capturing some natural language constructs that resemble variable declarations, were indeed too strict. While we are able to capture single statements in isolation, the reference structure exhibits a mismatch, causing the failure. After fixing this issue, we were able to disentangle all the considered Stack Overflow posts, increasing our confidence on the ability of our approach to correctly disentangle unambiguous structured fragments from narrative.

Partial Agreement Analysis

Another interesting analysis can be done if we consider all the possible top-level paragraphs of posts and their contents. We can extend the agreement analysis to reveal some insights about both the correctness of how people tag code, and the limitations of our approach itself.

Consider the three types of tagging performed by humans:

(1) top-level paragraphs completely tagged as code (*i.e.,* enclosed by the `<code>` tag);

(2) paragraphs tagged as pure natural language, with no in-paragraph code tagging (*i.e.,* enclosed in any tag but `<code>`, like `<p>`);

(3) in-paragraph tagging, where paragraphs tagged as narrative exhibit some sub-fragments tagged as code (*i.e.,* as in Listing 6.9).

Table 6.4 shows the agreement analysis for paragraphs completely tagged by people as code (2,928,766 paragraphs).

	Agreement		
Type	**None**	**Partial**	**Full**
Code	8.15%	34.70%	57.15%

Table 6.4. Agreement for Paragraphs Tagged as Code

Our approach obtains full coverage on around 57% of fragments, meaning that for 57% of paragraphs tagged as code, our island parser reconstruct a full-fledged H-AST (please note that this includes the case of islands with lakes). Partial agreement (*i.e.,* the parser finds some narrative mixed with code) is found in 35% of the paragraphs, and no agreement (*i.e.,* the parser

---

[5] https://archive.org/details/stackexchange

finds only narrative) in 8% of the cases. By manually inspecting these cases, we mostly found examples of other programming languages (*e.g.,* SQL, CSS). Partial agreement, instead, is mostly due to cases where people tag as code console logs or error output other than stack traces that contain incomplete code elements mixed with narrative.

Table 6.5 shows the coverage results for the paragraphs completely tagged as natural language, with no in-code paragraphs (a total of 7,704,072 paragraphs).

	Agreement		
Type	**None**	**Partial**	**Full**
Textual	1.38%	18.74%	79.88%

Table 6.5. Agreement for Paragraphs Tagged as Natural Language

As expected, a large amount (79.88%, *i.e.,* 6M paragraphs) of the content tagged as natural language is coherent, that is, it contains only narrative without code. However, a significant part of the remaining paragraphs (18.74%, *i.e.,* 1.4M paragraphs) is reported to contain some valid constructs, that are very likely to be code elements for the languages we support. Assuming the correctness of the island parser, this is evidence that users tend to forget to tag, or avoid to tag on purpose code elements by using alternative markup tags to emphasize the code within a discussion (*e.g.,* by using `<strong>` or `<blockquote>` HTML tags). Only a minimal part of paragraphs (*i.e.,* around 100K) are reported to be completely code by our parser. By manual inspection, we found that the top two untagged constructs found by our approach are reference types and qualified identifiers.

Finally, Table 6.6 reports the agreement values for paragraphs tagged as narrative that contain elements tagged as code (*i.e.,* 1,665,340 paragraphs). We report the agreement aggregated by sub-fragment type, *i.e.,* text or code.

	Agreement		
Type	**None**	**Partial**	**Full**
Code	54.53%	6.76%	38.71%
Textual	0.04%	3.87%	96.09%

Table 6.6. Agreement for Fragments with in-paragraph Code Fragments.

We adopt the same fragmentation process that we applied for whole posts to evaluate the ability of our approach to disentangle posts. According to the island parser output, only 38.71% of the in-paragraph tagged code in totally agreement with the human tagging, and more than a half of the sub-fragments (54.53%) in total disagreement. Again, we found examples of other programming languages (*e.g.,* SQL, CSS), and actual limitations of our approach, like with types with no real camel case like `String`, or isolated primitive types. These constructs cannot be easily identified with just a syntactic/lexical approach like ours, but require a technique that integrates domain knowledge and natural language processing.

Finally, only a minority of in-paragraph sentences that remain untagged are completely recognized as code (*i.e.,* 0.04% of fragments) or have some code elements (*i.e.,* 3.87% of fragments). As in paragraphs completely tagged as narrative, we found that the top two untagged constructs found by our approach are reference types and qualified identifiers.

6.3 Conclusion

We presented an automated approach to parse and model software engineering artifacts. We devised a multi-lingual island grammar to isolate and model constructs of interest from four different languages like Java (supporting version 8 to the fullest), XML, JSON, and Stack Traces through a full-fledged H-AST. We leveraged random testing to evaluate the robustness of the language grammars and their modeling capabilities in isolation. We compared our approach with an existing competing technique to extract simple Java constructs, and we evaluated the island parser and the model reconstruction in a concrete setting like Stack Overflow discussions.

Reflections

Going beyond the boundaries imposed by the textual representation, and modeling the contents in a H-AST gives a totally new perspective on how to manipulate data for development artifacts. Indeed, with a H-AST structure is preserved, and development artifacts can be navigated, or even transformed or modified in its contents. This additional level of abstraction opens up new possibilities to build novel tools and novel analysis. Some examples of these applications are presented in the next chapter.

7

Applications and Reusability

A fundamental aspect of any mining approach involving Stack Overflow posts concerns the intrinsic heterogeneity of the data: Posts are composed of both *unstructured fragments* representing the natural language part of the discussion, and *structured fragments* (*e.g.,* Java code, XML, JSON), both co-existing in the same artifact. The pure extraction of constructs of interest from natural language leaves a conceptual "hole" in the process. For example, an analysis can focus on identifying relationships between XML configuration files (*e.g.,* for the Android platform) and code samples, but after the extraction of structured fragments, the data comes still in the form of text. The multilingual grammar devised in the previous chapter goes beyond the plain textual representation of a development artifact, resulting in the H-AST model produced by the parser. This output model keeps track of the structure of the unstructured fragments within an artifact, filling the "hole" left by the absence of a model.

The next step is thus to model the extracted information and discover the semantic links among these elements to actually perform the specific analysis. To perform this step an additional abstraction layer on the information is needed.

In this chapter we present STORMED, a dataset modeling more than 800k Stack Overflow discussion concerning Java, where the contents are further modeled with a meta-information system. We discuss how this additional modeling of the information provided by STORMED can be leveraged and reused to build analysis and tools.

Structure of the Chapter

In Section 7.1 we describe STORMED and its meta-information model. In Section 7.2 we present an exploratory study to discover usages of `sun.misc.Unsafe` in Stack Overflow by using STORMED. In Section 7.3 we describe a tool to sanitize untagged code elements in Stack Overflow. In Section 7.4 we conclude the chapter.

7.1 StORMeD: Stack Overflow Ready Made Data

In this section we present the first application of our island parser. We fully exploit the H-AST as the basic block to build a meta-information model of the contents provided by Stack Overflow discussions. The meta-information model is embedded in a full-fledged structural model of Stack Overflow discussions, which also preserves the human tagging of the contents. We created STORMED, a dataset counting more than 800k discussions concerning the Java programming language, enabling reusability of the Stack Overflow data.

7.1.1 The Artifact Model

Figure 7.1 shows the artifact model for a Stack Overflow discussion. Three different colors are used to highlight the structure of the document, reaching the structural detail of the contents.
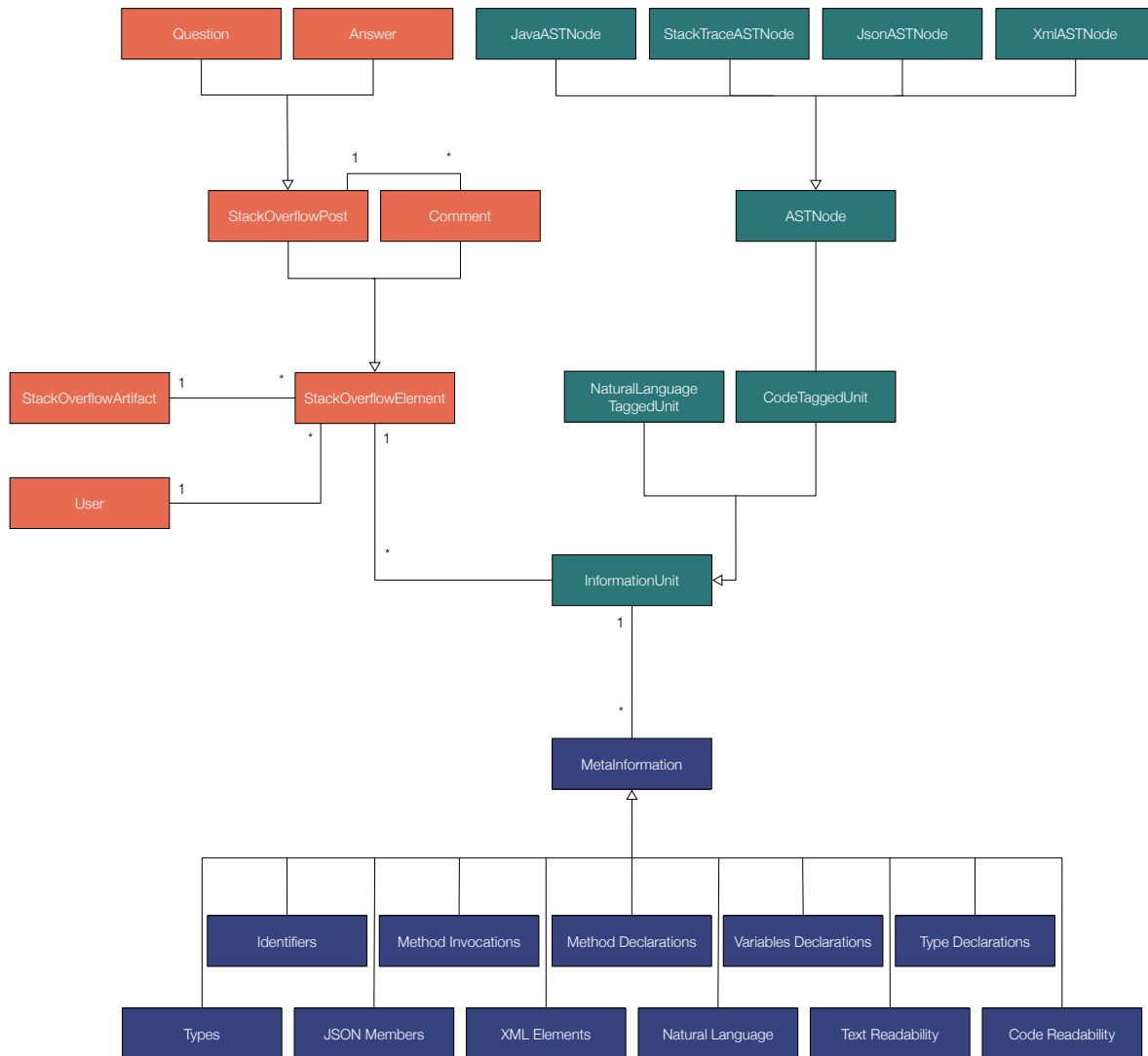
Figure 7.1. Object Model for a Stack Overflow discussion.

The object model depicted in Figure 7.1 follows a top-down decomposition of the artifact. The orange part represents the structure of the artifact itself: A Stack Overflow discussion is composed by a set of posts that can be question and answers, each post has an owner (*i.e.,* user) and a set of comments. The green part concerns the human tagging performed in the contents. In this case `CodeTaggedUnit` represents where users highlights non textual elements by using the `<code>` tag, while `NaturalLanguageTaggedUnit` represents all other HTML tagging in the body of a post. The blue part models the meta-information contained in the contents. Each type of meta-information describe a specific type of information concerning code and natural language.

`<p>` I am migrating from xml based spring configuration to "class" based configuration using the corresponding **@Configuration** annotation.

`<p>` I came across the following problem: I want to create a new bean, which has a reference to another (service) bean. Therefore I autowired this class to set this reference during bean creation. My configuration class looks as follows:

`<code>`
```
@Configuration
@ComponentScan(basePackages = {"com.akme"})
public class ApplicationContext {

    @Resource
    private StorageManagerBean storageManagerBean;

    @Bean(name = "/storageManager")
    public HessianServiceExporter storageManager() {
        HessianServiceExporter hessianServiceExporter =
                            new HessianServiceExporter();
        hessianServiceExporter.setServiceInterface(StorageManager.class);
        hessianServiceExporter.setService(storageManagerBean);
        return hessianServiceExporter;
    }
}
```

`<p>` But this doesn't work, because the causes a **BeanNotOfRequiredTypeException** exception during startup.

`<code>`
```
Bean named 'storageManagerBean' must be of type
[com.akme.StorageManagerBean], but was actually of type
[com.sun.proxy.$Proxy20]
```

`<p>` The **StorageManagerBean** is annotated with an **@Service** annotation. And the xml based configuration worked as expected:

`<code>`
```
<bean name="/storageManager"
  class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="storageManagerBean"/>
  <property name="serviceInterface" value="com.akme.StorageManager"/>
</bean>
```

Figure 7.2. Example of Stack Overflow question with HTML tagging.

7.1.2 Preserving the human tagging

A user on Stack Overflow can create a post by using a subset of the HTML language. In this subset, the user can indeed make use of tags like `<code>` to highlight code snippets in a discussion. In Section 6.2.3 we analyzed and estimated the agreement of the tagging performed by users. The analysis highlighted how contents tagged as `<code>` at the top level might not provide code at all, as well as the "textual" part (*i.e.,* not tagged as `<code>` at top level), might have untagged code elements.

Figure 7.2 shows the same conversation[1] discussed in Chapter 6 (see Figure 6.2) with the actual HTML tagging on the left side. By performing this tagging, the user is letting the reader now where the candidates structured fragments are. somehow differentiating the nature of two separated parts of the contents, an two different type of *information units*. To keep track of the

---

[1] http://stackoverflow.com/questions/23080114

human tagging, we generalize the tagged contents to two different types of information units:

Natural Language Tagged Unit: Whatever is not tagged as `<code>` at the top level, like textual decorations (*e.g.,*`<b>`, `<hr>`), lists (*e.g.,*`<ol>`, `<ul>`), and paragraph (*i.e.,*`<p>`).

Code Tagged Unit: Every contents tagged as `<code>` at the top level.

These information unit types expose a H-AST node providing the parsed contents. In doing so, the model takes care of mistagged contents, thus allowing potential analysis to take this aspect into account.

7.1.3   The meta-information Model

According to the object model depicted in Figure 7.1, every information unit carries a set of *meta-information*. The meta-information model enables the decoration of information units with, for example, the result of an analysis or the traversal of the H-AST. We provide the following pre-computed meta-information:

Types: the set of Java types mentioned in a unit, including qualified types (reference types) and primitive types (*e.g.,* int, double);

Type Declarators: all the H-AST nodes matching a Java type declaration, including classes, interfaces, and enumerators;

Variable Declarators: all the H-AST nodes matching a variable (or field) declarator;

Method Invocations: all the H-AST nodes matching a method invocation;

Method Declarators: all the H-AST nodes matching a method declarator;

Code Identifiers: all the H-AST nodes matching an identifier;

JSON: all the H-AST nodes matching a JSON member declaration, *i.e.,*, a pair composed by a member name and a declaration (*e.g.,* object, string, number, boolean);

XML: all the H-AST nodes matching an XML tags, both single tags with no children (*i.e.,*`<tag/>`, `<tag>`) and composed tags with children;

Natural Language: the term frequency (*tf*) vector that can be used to calculate, for example, textual similarities. The *tf* vector is generated using Apache Lucene[2]. We split text on case change, on digits and symbols, we lower the case, we remove stop words, and we apply the snowball stemmer[3] to the obtained terms;

Sentiment Analysis: it provides the overall sentiment analysis value of the information unit from a pure textual point of view. This type of information unit should not concern structured units, since sentiment analysis on source code (*e.g.,* Java, XML) would lead to non-sense results;

---

[2]`http://lucene.apache.org`
[3]`http://snowball.tartarus.org/`

Textual Readability Indexes: These represent the comprehension difficulty when reading a passage in English and are different approximations and representations of the U.S. grade level[4] needed to comprehend the text. We use the Stanford NLP Parser[5] to extract sentences and words, and TeX hyphenation [Lia83] to obtain syllables. The indexes include: *Automated Reading Index* [SSS67], *Coleman-Liau Index* [CL75], *Gunning Fox index* [Gun52], *SMOG Grade* [McL69], *Flesch Reading Ease Score* and *Flesch Kincaid Grade Level* [Fle48];

Code Readability Index: an index devised by Buse and Weimer [BW10] to evaluate the readability of java-like code samples that considers different metrics like identifier length, for loops, if blocks *etc.*

The aforementioned meta-information types are the set computed by default. Not all of the meta-information are suitable to every type of information unit. *Text Readability* and its code counterpart *Code Readability* are two examples. The former is designed to work with narrative, while the latter is designed to work with source code. If they were used on different input types they do not properly work. For example, source code would result unreadable according to text readability metrics.

Vice versa, meta-information concerning code elements (*e.g.,* method invocations, declarators) might help discovering structural and semantic links between textual units and code units, due to their general applicability. For example, in the discussion depicted in Figure 7.2, the *Types* meta-information would contain `StorageManagerBean` both for the first code unit and for the last text unit. With STORMED this information would be uncovered with a simple traversal of the meta-information model, without reprocessing the data.

The model can be easily generalized, allowing custom analyses to decorate the information unit with their result stored as ad-hoc meta-information type. For example, traditional source code metrics [LM10] could be calculated when applicable and modeled as meta-information of code units. The organization of the meta-information model, together with the ready-made nature of STORMED, favors the customization and reuse of Stack Overflow data to perform analysis tailored to specific needs. For example, Stack Overflow can be fully analyzed to discover information about undocumented libraries (*e.g.,* bugs, usages and patterns) that would require a full-blown analysis of the Stack Overflow dataset otherwise.

7.2   Usages of `sun.misc.Unsafe` in Stack Overflow

Unbeknownst to many application developers, the Java runtime includes a "backdoor" that allows expert library and framework developers to circumvent Javas safety guarantees. This backdoor is there by design, and is well known to experts, as it enables them to write high-performance "systems-level" code in Java. This backdoor is provided through an unofficial and undocumented API that allows the developer to access low-level, unsafe features of the Java Virtual Machine (JVM) and underlying hardware, features that are unavailable in safe Java bytecode. This API is provided through an undocumented class, `sun.misc.Unsafe`, in the Java reference implementation produced by Oracle.

Identifying Stack Overflow discussions concerning the usage of `sun.misc.Unsafe` cannot be performed by solely relying on the tagging system provided by Stack Overflow. The topic is rarely discussed and the only tag called `<unsafe>` is rather used to identify unsafe usages in code not only focused on the java programming language. If we consider the tag pair `<java>` plus

---

[4]`http://en.wikipedia.org/wiki/Grade_levels`
[5]`http://nlp.stanford.edu/software/index.shtml`

`<unsafe>`, the contents are not only focusing on `sun.misc.Unsafe`. An analysis of the contents is thus required to understand if a discussion tackles `sun.misc.Unsafe`.

Without relying on any tagging of the discussion, we need to discover specific constructs in the contents that suggest the usage of `sun.misc.Unsafe`. This information can be obtained by analyzing both the text and the code contained in a discussion. For example, a discussion could report a code sample using some features of the `sun.misc.Unsafe` class, or a user could mention the class in an answer to a question concerning some specific problem that the usage of the class can tackle. Identifying pieces of the discussion that matches the information concerning `sun.misc.Unsafe` requires an in depth analysis of the text. STORMED reveals to ideal tool to perform such type of analysis. As a proof of reusability, in this section we employ STORMED to analyze discussions on Stack Overflow, discover the ones concerning `sun.misc.Unsafe`, avoid false positives

7.2.1   Identifying discussions by type and method names

To identify Stack Overflow discussions concerning the `sun.misc.Unsafe` class, we start by analyzing all the discussions whose tags contains one among *java*, *scala*, and *android*, *jvm*. One possible solution to understand if a discussion concerns `sun.misc.Unsafe`, is to (i) discover usage of one of the methods exposed by the class or (ii) identify any mention of of the type *Unsafe*. We focus on the following AST nodes to check if a discussion matches one of the two criteria:

Method Invocations: each node matching a method invocation node is analyzed to understand if the invoked method name belongs to `sun.misc.Unsafe`. In case of match, the post is marked as containing a method name of *Unsafe*. We also perform check on the callee to understand if the method invocation is effectively performed on the *Unsafe* class. We check this information on the callee by applying the same rules used for qualified identifiers.

Strict Method Name Identifiers: we consider identifiers respecting the java naming convention for methods. Every identifier beginning with a lowercase letter and containing a case change (*i.e.,* fieldObject) is taken in consideration as method name. The method name must then match one of the methods declared in the class *Unsafe*.

Qualified Identifiers: qualified identifiers are nodes that are generally used in other constructs. For example, they are used in import declarations, method invocations (before the method name) and stack trace lines (between "at" and the line number). For this reason we check if the qualified identifier matches value like *Unsafe*, *unsafe*, *UNSAFE* or the fully qualified type `sun.misc.Unsafe`. In case of match, the post is marked as declaring the type *Unsafe*.

Strict Qualified Identifiers: as well as for the strict method name identifiers, we also check the strict qualified identifier appearing in the natural language. We look for all the occurrences of qualified identifiers composed by 3 identifier at least (*i.e.,*`sun.misc.Unsafe`). Whatever matches this construct is treated as a normal qualified identifier.

String Literals: we verify that the fully qualified type `sun.misc.Unsafe` is present in the literal. We also verify that literal matches the string "theUnsafe". This string is a special field name in the *Unsafe* class for the HotSpot VM to get the instance via reflection. Both these rules suggest an usage of the class via reflection and the presence of `sun.misc.Unsafe`.

Stack Traces: we keep track of full stack traces and lone stack trace lines to either identify method names or the type *Unsafe*.

7.2.2   Refining sun.misc.Unsafe.park usages

Whenever a thread is put in the idle state, a call to the *park*. If an exception occurs in the thread, it is likely to find *sun.misc.Unsafe.park* in the method invocations of the trace. In this case, the presence of the method park does not represent a relevant usage of *Unsafe* and makes the *park* method the most used in Stack Overflow. For this reason, we ignore occurrences of *park* inside stack traces.

7.2.3   Refining Parsing Results

The analysis performed on the AST allows us to identify if a post contains the type or a method name of the class *Unsafe*. We collected 20915 discussions matching at least one of the two criteria, out of which 560 discussion reports the type *Unsafe* and 20426 reports a method name of *Unsafe*. However, if the presence of the type *Unsafe* guarantees that the discussion is effectively about `sun.misc.Unsafe`, the lone presence of the method name could misclassify the discussion.

For example, methods like *getInt*, *getFloat*, and *getShort* can be found in other classes like *ByteBuffer*[6], while a method name like *defineClass* can be found in the java *ClassLoader*[7]. The absence of type in our parsing results does not guarantee that the discussion is not including `sun.misc.Unsafe`. Indeed, we do not check at parsing time if the lone term "unsafe" is mentioned among the natural language parts to avoid false positives. To overcome the safety limitation we imposed in the parser, we take all the discussion with a method name of the class *Unsafe*, and we perform a pure text search of the term "unsafe".

Out of 20426 discussions with an *Unsafe* method mentioned, only 49 discussions contain the term "unsafe" in the text. We proceed by manually inspecting and verifying each discussion, resorting to 18 discussion effectively reporting an usage of `sun.misc.Unsafe`. Thus, our final dataset contains a 560 discussions explicitly using the type *Unsafe*, and 18 discussions reporting the method name only and the term "unsafe", for a total of 578 discussions that effectively concern `sun.misc.Unsafe`.

7.2.4   Stack Overflow Discussions

To understand which topics are related to posts mentioning `sun.misc.Unsafe` and its methods, we started by analyzing the tags of the corresponding questions. Table 7.1 shows the overall occurrences.

Popularity of Repliers

To understand how difficult are the topics related to the specific features that might require the use of `sun.misc.Unsafe`, we collected all the repliers of the questions in our final dataset. The answers may or may not contain references to `sun.misc.Unsafe`, but they are representative of the possible topics involved in posts mentioning this undocumented class. The resulting set of users has, at the moment of the Stack Overflow dump, an average reputation of 18,000. This is just below the level of *trusted user*[8], which is the highest reputation rank for getting special privileges on Stack Overflow.

We refined our selection to include only the answers mentioning `sun.misc.Unsafe` or one of its methods. In this case, the average reputation of the corresponding repliers is around 21,770,

---

[6]`http://goo.gl/uH4oJZ`
[7]`http://goo.gl/iN3dhm`
[8]`http://stackoverflow.com/help/privileges`

Table 7.1. Most frequent tags

Tag	Occurrences	Tag	Occurrences
java	366	arrays	16
multithreading	32	memory-management	15
android	27	jni	14
jvm	26	bytebuffer	13
concurrency	24	c++	12
memory	20	reflection	11
unsafe	18	serialization	10
performance	18	atomic	10

which is even above the maximum privilege threshold. Assuming a correlation of Stack Overflow popularity with user expertise, one might conclude that this is evidence that the topics related to `sun.misc.Unsafe`, and even more the techniques to exploit and use `sun.misc.Unsafe`, require a significant development experience.

To get further evidence that the topics related to `sun.misc.Unsafe` attract popular and expert users, we computed the distribution of replier's reputation among the ranks defined in the Stack Overflow reputation league[9], as shown in Table 7.2.

Table 7.2. Distribution of Repliers Reputation.

Reputation Range	All Users	Repliers	
		all	with `sun.misc.Unsafe`
1–199	3,276,655	120 (0.0%)	31 (0.0%)
200–499	80,105	51 (0.1%)	10 (0.0%)
500–999	49,825	62 (0.1%)	18 (0.0%)
1,000–1,999	30,833	103 (0.3%)	31 (0.1%)
2,000–2,999	11,847	65 (0.5%)	17 (0.1%)
3,000–4,999	10,151	93 (0.9%)	35 (0.3%)
5,000–9,999	7,462	133 (1.8%)	34 (0.5%)
10,000–24,999	4,278	140 (3.3%)	42 (1.0%)
25,000–49,999	1,271	72 (5.7%)	19 (1.5%)
50,000–99,999	444	41 (9.2%)	17 (3.8%)
100,000+	234	39 (16.7%)	14 (6.0%)

From the distribution reported in Table 7.2, the topics discussed in posts where `sun.misc.Unsafe` is mentioned attracted 39 top-ranked users, corresponding to 16.7% of all top-ranked users, and 14 of them discussed and mentioned `sun.misc.Unsafe` or one of its methods (corresponding to 6% of top-ranked users).

---

[9]See `http://stackexchange.com/leagues/1/alltime/stackoverflow`

7.3 A Code Retagger for Stack Overflow

In Chapter 6 we showed that, due to the limitations imposed by Stack Overflow, `<code>` tags do not guarantee the presence of actual code in their contents. Similarly, we also showed how users tend to forget valid constructs untagged in the narrative (see Figure 6.1, and Figure 6.2).

In this section we take advantage of the power of the H-AST model to automatically re-tag the contents of a Stack Overflow discussion, and sanitize the lapses of the users. While parts of the discussion tagged as `<code>` are hard to sanitize, due to the limitations imposed by Stack Overflow (*i.e.,* there is no real replacement for `<code>`), it is possible to clean untagged code elements immersed in narrative. We developed an extension for the Chrome web browser that takes advantage of the island parsing service of STORMED.

Figure 7.3. The re-tagged version of the question depicted in Figure 6.1 rendered in the web browser.

Figure 7.3 shows the same discussion depicted in Figure 6.1 after the retagging process. Untagged elements like `TextEntity`, `JTextArea`, `getWidth()`, and `getHeight()` are identified and correctly retagged by the STORMED parsing service. The identified untagged code fragments are enclosed in additional `<code>` tags, while preserving the original contents of the discussion.

7.3.1 Architecture

Figure 7.4 shows the architecture of the Stack Overflow retagger. On the right side is depicted the plugin for installed on the Chrome web browser, while on the left side is depicted the STORMED parsing service. The Chrome plugin is composed of to components. The first one *Contents Extractor* is activated when the visited page matches the Stack Overflow domain. The extractor navigates the DOM of the web page, and select the HTML element enclosing both question, answers, and their related comments, yet excluding top and side bars containing irrelevant contents. The extracted contents is sent to the STORMED parsing service, where the *Multi-lingual Island Parser* parses the HTML, and creates an H-AST model of the contents. With the H-AST model,
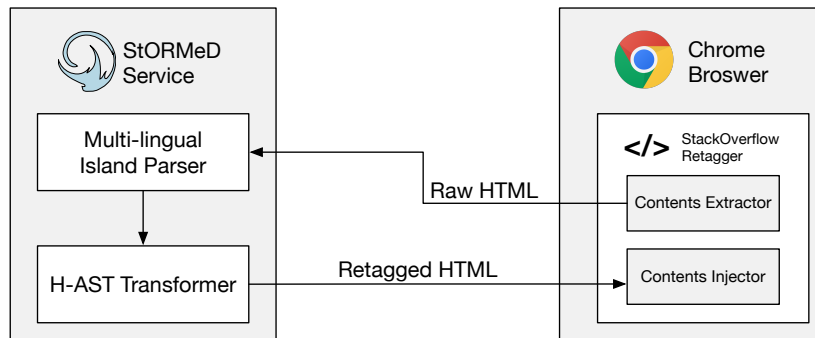
Figure 7.4. The Stack Overflow Retagger Architecture.

it is possible to visit each part of the HTML (*i.e.,* the DOM) and perform a second analysis to understand if the textual fragments of in the HTML actually contain valid code elements. The visit is performed by the *H-AST Transformer*, which excludes all the branches underneath a `<code>` tag, and analyzes all the free text encountered during the visit.

Listing 7.1. Example of HTML Tagging

```
<p>
    Some tagged
    <code>aMethod()</code>
    and untagged code anotherMethod()
</p>
```

Listing 7.1 shows an example of paragraph with untagged and tagged code. In this case, the *H-AST Transformer* exclude the `<code>` tag from the visit of the DOM, thus ignoring the text fragment *"aMethod()"*, and visits the the two remaining text fragments *"Some tagged"* and *"and untagged code anotherMethod()"*. To understand if the free text contains untagged code elements, the *H-AST Transformer* renders the HTML escapes (*e.g.,* `<`) to obtain plain text, and it runs the multi-lingual island parser to identify code elements. If the parser returns some element, the corresponding text enclosed in `<code>` tags.

Listing 7.2. Example of HTML Tagging

```
<p>
    Some tagged
    <code>aMethod()</code>
    and untagged code
    <code>anotherMethod()</code>
</p>
```

Listing 7.2 shows the final transformed HTML. The text fragments *"and untagged code anotherMethod()"* is transformed so that the part *anotherMethod()* is enclosed in `<code>` tags, while the rest of the H-AST remains untouched. Once the transformation of the H-AST is completed, the retagged HTML is sent to the *Contents Injector*, which substitutes the original HTML in the web browser, and add the STORMED logo on the discussion's title.

7.4 Conclusions

We presented StORMeD, a dataset and service that models Stack Overflow posts by building a H-AST for each discussion in a publicly available data dump. Our dataset enables the navigation of the contents of a discussion by differentiating among Java code, XML, JSON, stack traces, and natural language fragments. We described a ready made meta-information that describes and leverages the heterogeneity of Stack Overflow. The meta-information model describes several aspects of the information concerning code and text, like term frequency vectors, readability indexes, and code constructs either mentioned or standalone.

We conducted an exploratory study to discover usages of `sun.misc.Unsafe`, showing how the StORMeD dataset can be reused without having to analyze a dataset of considerable size like Stack Overflow from scratch. We discussed how the aggregation of meta data concerning community (*e.g.,* reputation) in StORMeD can be harnessed as well to discover that `sun.misc.Unsafe` catches the attention of the most well reputed and experienced users on Stack Overflow.

Last but not least, we also provided another proof of concept by building a Stack Overflow retagger that automatically sanitizes untagged code elements in the narrative. The approach followed in this second application could potentially be integrated as a tool in the Stack Overflow pipeline to automatically tag posts without requiring human intervention, or a helper tool to sanitize post at creation time.

Reflections

This chapter presented a set of applications and analysis that can be built on top of the multilingual island parser and the H-AST model described in Chapter 6. The usefulness of the H-AST model, and the resulting StORMeD dataset and service is described by the application and analysis themselves. All the chapter can be summarized with one single word: *modeling*.

In Chapter 6 we started a first phase of low level modeling, by devising the concept of H-AST, which allowed to preserve the structure of the contents. The StORMeD dataset described in Section 7.1 would not be possible without such low level modeling. The meta-information model of StORMeD is implicitly built on top of the H-AST, which in turn, provides an additional layer of modeling abstraction, focusing one the information. The analysis performed in Section 7.2, as well as the retagging tool described in Section 7.3, take both advantage of this two-sided modeling phase allowing to (1) analyze information without having to recompute data, and (2) reshape the contents of a development artifact like a Stack Overflow discussion on the fly.

In the next chapter we take advantage of the multi-lingual island parser and its H-AST model to model the information of artifacts whose primary nature is not textual. We start moving the first steps towards the definition of H-RSSE by cross-recommending items of heterogeneous nature like Stack Overflow discussions and YouTube videos in the same application.

8

Extracting Relevant Fragments from Software Development Video Tutorials

The approaches and applications described in the previous chapters focus on textual artifacts like Stack Overflow discussions. Even though most of the artifacts perused by developers are of textual nature (*i.e.,* bug reports, development emails), the knowledge needed by developers to understand a concept can be also acquired from other type of resources.

A prominent example are *video tutorials*, a new and emerging source of information that can be effective in providing a general and thorough introduction to a new technology, yet providing a learning perspective different and complementary to that offered by traditional, text-based sources of information [MSB15].

Despite these benefits, there is still limited support for helping developers to find the relevant information they require within a video. In many cases, video tutorials are lengthy, and lack an index to allow finding specific fragments of interest.

In this chapter we present CODETUBE, an approach to leverage the information found in video tutorials and other online resources. Given a textual query (*e.g.,* "implementing an Android listener") and the type(s) of video tutorial a developer is interested in (*e.g.,* "theoretical concepts", "code implementation", "working environment setup"), CODETUBE recommends video tutorial fragments relevant to the query and to the specific developer's needs, and complements the recommended video tutorial fragments with related Stack Overflow discussions.

Structure of the Chapter

Section 8.1 reports the design and results of a study we run with the aim of identifying categories of development video tutorial fragments and investigating how video tutorials are composed. Section 8.2 details CODETUBE, while Section 8.3, Section 8.4, and Section 8.5 describe and report the results of the three evaluations. Threats to validity are discussed in Section 8.6, while Section 8.7 concludes the chapter.

8.1   Investigating the Structure of Video Tutorials

Previous research on development video tutorials [MSB15] investigated the motivation and purpose of the *whole* tutorial, rather than looking deeper at its structure and content. Even if not explicitly stated, a video tutorial has an intrinsic structure embedded in the flow of actions performed by the tutor.

Table 8.1. Participants' Occupation.

Occupation	Total	%
Faculty	1	2%
PhD Student	3	7%
Master Student	4	10%
Undergraduate Student	31	76%
Professional Software Developer	2	5%
Total	41	100%

When it comes to devise an automated approach to analyze, fragment, classify, and index video tutorials, understanding the aforementioned structure of the original video is essential to provide, for example, advanced searching features.

The *goal* of this study is to understand which are the typical parts/sections composing a software development video tutorial (*e.g.,* setting of the IDE, code writing, *etc.*). The *context* consists of objects, *i.e.,* 150 video tutorials collected from YouTube, and participants, *i.e.,* 41 computer science students/professors and professional developers manually tagging the different parts of the tutorials (*e.g.,* "from 1:00 to 3:30 the tutorial shows how to set the IDE").

8.1.1 Context, Data Collection & Analysis

We collected from YouTube the video tutorials used in the context of our tagging study. The manual collection was needed to make sure of selecting real tutorials dealing with a diverse set of topics at different levels of abstraction (*e.g.,* theoretical *vs* practical tutorials). We selected (i) 50 generic Java tutorials, (ii) 50 tutorials dealing with JSPs and Servlets (*i.e.,* Java Web applications), and (iii) 50 Android-related tutorials (*i.e.,* Java mobile apps). We made sure to include both tutorials for beginners as well as for experienced developers and to select a mix of theoretical and practical tutorials. For example, the 50 Java-related tutorials included tutorials about Java basics (*e.g.,* exceptions handling), advanced topics (*e.g.,* multithreading), and theoretical notions (*e.g.,* how the garbage collector works). The selection of the tutorials was performed by one author and double-checked by a second author. All 150 tutorials focus on the Java programming language, because (i) as it will be detailed later, our approach exploits the multi-lingual island parser presented in Chapter 6 to identify code constructs shown in the video tutorial, and (ii) this eased the selection of participants for our study (*i.e.,* some basic knowledge of Java programming was required). Also, to limit the effort required from participants, we did not include video tutorials longer than 20 minutes.

We invited 55 computer science students and professors as well as five industrial software developers. Each participant received an email with a link to the web application where they could read instructions and then perform the tagging tasks. We asked each participant to watch video tutorials and to split them into categorized fragments: They had to identify disjoint parts of the video tutorials and tag each with a category explaining its main purpose (*e.g.,* "from 1:00 to 3:30 it explains how to set the working environment, from 3:31 to 5:00 it shows how to implement a JSP"). We asked participants to extract and tag fragments for at least 20 minutes of video tutorials. However, participants were free to tag more or less. Invitees had up to two months to perform the tasks. Data about the 41 participants who replied to the call is reported in Table 8.1, Table 8.2, and Table 8.3.

We implemented a web application to allow participants to tag video tutorial fragments in multiple rounds (*e.g.,* tagging one video tutorial today, and another one after one week).

Table 8.2. Participants' Experience in Java.

Experience in Java	Total	%
Less than 1 year	29	71%
1-3 years	7	17%
3-5 years	3	7%
5-10 years	2	5%
More than 10 years	0	0%
Total	41	100%

Table 8.3. Participants' Usage of Video Tutorials.

Usage of Video Tutorials	Total	%
Daily	29	71%
Few times a week	7	17%
Few times a month	3	7%
Rarely	2	5%
Never	0	0%
Total	41	100%

After registering, participants provided some background information which we report in Table 8.1, Table 8.2, and Table 8.3. The actual tagging of the video tutorial fragments starts through the user interface depicted in Figure 8.1. The web application shows to the participant one of the 150 tutorials embedded in a YouTube video player (1). The tagging application was designed to balance the number of participants tagging each video tutorial, *i.e.,* in a first iteration, the web application assigned each video tutorial in our dataset to at least one participant. Then, if possible, a second participant was assigned to each tutorial, and so on.

At the bottom of the page there are controls to allow participants to create and tag fragments—see Figure 8.1 (2). Participants are allowed to freely interact with the video player as they wish, yet they are forced to follow some constraints when devising the fragments. Firstly, all the fragments need to be contiguous or the application does not allow the user to store the tagging session. Secondly, the fragments have to cover the whole video. A progress bar (3) allows the users to keep track of the amount of video covered by the fragments already devised. To avoid corrupted data, the application raises an error if the tagging progress goes beyond 100%, or if any tag is missing, and does not allow to store the session if the video coverage is not complete. Last, each user is requested, but not forced, to tag at least 20 minutes of video tutorials. Another progress bar (4) shows the overall minutes tagged by the study participants. Once the bar gets to 20 minutes, the participant is notified with a pop-up label at the bottom of the application, but the application leaves the participant the choice of keeping tagging video fragments. Each participant tagged on average 29 minutes of video tutorials (min 7, median 27, max 75).

We collected 784 tagged video fragments (1,219 minutes) from 136 video tutorials. 14 of the 150 video tutorials we selected were not analyzed by any participant, while the remaining 136 were tagged by one participant each.

Two of the authors performed an open coding process on the 784 tagged fragments to group them into categories. They independently created classifications for the participants' tags. They met to discuss and refine (or merge) the identified categories, reaching an agreement when needed. After the first coding, the two authors disagreed on 53 fragments, for which one of them produced an *Unclassified* categorization, whereas the other was indeed able to produce a correct category.

Figure 8.1. User Interface of the Fragment Tagging Web Application.

Table 8.4. Categories Resulted from the Open Coding Process.

Category	#Tags	%
Code implementation (CI)	288	37%
Introduction to the tutorial topic (ITT)	144	18%
Execution of the implemented code (EIC)	124	15%
Theoretical concepts (TC)	87	11%
Closing of the tutorial (CT)	47	6%
Working environment setup(ES)	39	5%
Dealing with errors (DE)	19	3%
Unclassified	36	5%
Total	784	100%

Table 8.5. Transition frequencies between different parts of the video tutorials.

	ITT	TC	EIC	CI	ES	DE	CT	END
START	84.62%	3.50%	0.70%	6.99%	4.20%	0.00%	0.00%	0.00%
ITT	3.03%	21.97%	2.27%	48.48%	21.21%	2.27%	0.00%	0.76%
TC	0.00%	12.64%	6.90%	40.23%	3.45%	4.60%	5.75%	26.44%
EIC	2.70%	12.61%	5.41%	32.43%	1.80%	5.41%	7.21%	32.43%
CI	1.19%	7.94%	33.73%	33.33%	0.79%	4.37%	8.33%	10.32%
ES	2.38%	7.14%	9.52%	40.48%	2.38%	7.14%	4.76%	26.19%
DE	0.00%	17.86%	21.43%	21.43%	0.00%	3.57%	7.14%	28.57%
CT	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%

Overall the inter-rater agreement after the first coding phase—computed in terms of Cohen's Kappa [Coh60]—is equal to 0.86, which is considered a very strong agreement. The output of this process is a set of categories that can be used to describe the different parts composing software development video tutorials. To give an example, 144 fragments were marked with tags clearly referring to the *introduction of the tutorial topic* (*e.g.,* "introduction", "topic introduction", "tutorial introduction", *etc.*).

8.1.2   Analysis of the Results

Categories of video tutorial fragments

Table 8.4 reports the seven categories of video tutorial fragments derived from our open coding procedure. For 36 fragments we were not able to understand the meaning of the tags assigned by the participants—see the *Unclassified* row—(*e.g.,* details, observations). We excluded these tags from our study.

Most tagged fragments (37%) refer to *code implementation* activities. Examples of tags in this category include "program writing", "JComboBox implementation", and "implementing a JSP page". *Introduction to the tutorial topic* is the second most popular category, grouping 18% of the assigned tags. It concerns parts of the tutorial where the main tutorial topic is presented from a general point of view without providing implementation details. This category is followed by *execution of the implemented code* (15%), including 124 fragments in total. The latter category includes tags like "deployment and execution of the implemented web app" and "program execution and test logger".*Theoretical concepts* (*i.e.,* when some specific aspects of the
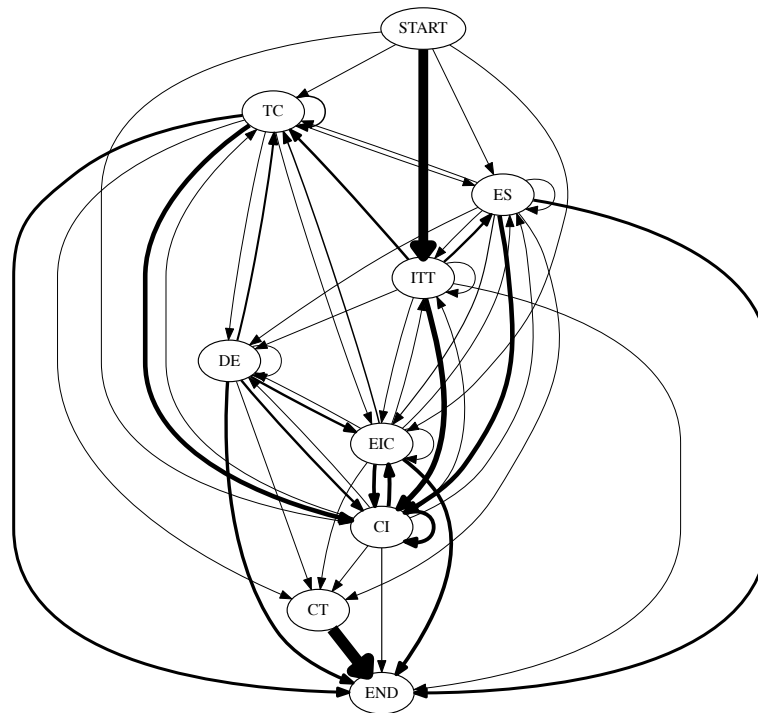
Figure 8.2. Transition graph between the different parts of the video tutorials.

topic are explained in detail, possibly interleaving slides/discussions with some code examples) are in 11% of the tagged fragments (*e.g.,* "explaining HTTP status codes") while the *working environment setup* category groups 39 (5%) tags (*e.g.,* "IDE settings"). Finally, 47 tags (6%) are related to the *closing of the tutorial* and 19 (3%) to explanation on how to *deal with errors* one could encounter while implementing the pieces of code subject of the video tutorial (*e.g.,* "what happens if we do not properly configure log4j").

These seven categories are the ones considered in CODETUBE when automatically classify the type of the extracted video tutorial fragments.

Structure of Video Tutorials

The availability of tagged and classified video fragments allows us to provide—without generalizing beyond the samples on Java development—an idea of how development video tutorials are structured.

Table 8.5 and Figure 8.2 report and depict the transition frequencies—estimated across all videos of our dataset—between different types of video tutorial fragments. While the table shows precise frequency values, the figure depicts the structure of video tutorial in the form of a Markov chain, where thicker transition edges indicate higher probabilities. When a video tutorial starts, in 85% of cases an introduction about the tutorial topics is provided. In the remaining 15% of cases, the tutorials directly starts with an implementation activity (7%), the setting of the working environment (4%), an explanation of theoretical concepts (3%), or the execution of the code that will be object of the tutorial (1%).
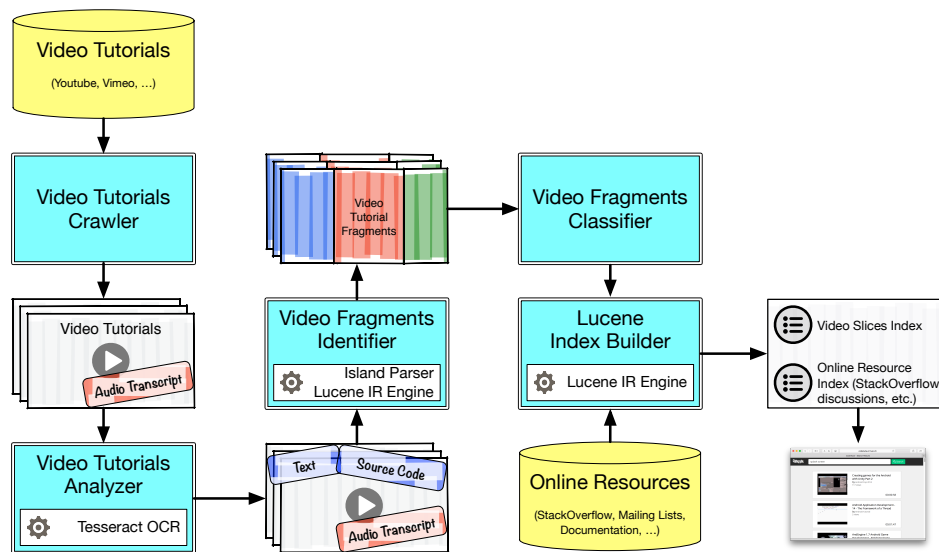
Figure 8.3. CodeTube: Analysis process.

After the topic introduction, 49% of the tutorials deal with code implementation activities, often representing the bulk of software development tutorials. Typical transitions are the ones START→ITT→ES→CI starting with a topic introduction (START→ITT=85%), continuing with the setting up of the working environment (ITT→ES=21%) and then starting an implementation activity (ES→CI=21%). Other tutorials (22%), focusing more on theoretical aspects, start explaining theoretical concepts right after the topic introduction.

In 37% of cases a code implementation fragment is followed by another one (CI→CI=37%), because the tutorial features independent implementation activities (*e.g.,* how to use method *A* and method *B* of a given API). Other frequent transitions happen from code implementation to code execution (38%), often (48%) followed by another code implementation activity (*i.e.,* a transition CI→EIC→CI).

Outgoing transitions from fragments dealing with theoretical concepts (TC node in Figure 8.2), are generally followed by implementation activities (TC→CI=55%) or by another theoretical fragment on a different concept (TC→TC= 17%). Instead, those outgoing from fragments dealing with common errors (DE) are almost equally distributed between: (i) theoretical concepts (DE→TC=25%), explaining why a specific error arises, (ii) the execution of the implemented code (DE→EIC=30%), showing how the error manifests at execution time, and (iii) code implementation activities (DE→CI=30%), showing how to fix the error.

The results discussed above highlight how the latent structure of a video tutorial can be quite complex. This recalls (and justifies) the need for an approach to automatically navigate among fragments to search/browse video tutorials to pinpoint the interesting parts.

8.2   CodeTube Overview

CODETUBE is a multi-source documentation miner to locate useful pieces of information for a given task at hand. The results are fragments of video tutorials relevant for a given textual query, augmented with additional information mined from other "classical", text-based online resources.

Figure 8.3 depicts the CODETUBE pipeline. It is composed of (i) an offline analysis phase aimed at collecting and indexing video tutorials and other resources, and (ii) an online service where developers can search these processed resources. The analysis of video tutorials is currently limited to English videos dealing with the Java programming language. In the following we detail each step of the CODETUBE pipeline.

8.2.1 Crawling and Analyzing Video Tutorials

The first step of the process is defining the topics of interest. The user provides (i) a set of queries Q describing the video tutorials she is interested in (*e.g.,* "Android development") and (ii) a set of related tags T to identify and index relevant Stack Overflow discussions (*e.g.,* "Android"). Each query in Q is run by the *Video Tutorials Crawler* using the YouTube Data API[1] to get the list of YouTube channels relevant to the given query $q_i \in Q$. For each channel the *Video Tutorials Crawler* retrieves the metadata (*e.g.,* video url, title, description) and the audio transcripts, which are either automatically generated or written by the author. Using GOOGLE2SRT[2] we extract the transcriptions for the videos. The crawling of video meta-information is performed on YouTube, but it can be extended to any video streaming service or video collection where the same type of meta-information and transcripts are available or can be extracted, *e.g.,* using a speech recognition API.

Once the videos have been crawled, their metadata is provided as input to the *Video Tutorial Analyzer*. It analyzes each video and extracts pieces of information to isolate video fragments related to a specific topic. The *Video Tutorial Analyzer* aims at characterizing each video frame with the text and the source code it contains. It uses multi-threading to concurrently analyze multiple batches of videos.

Frame Extraction

The analysis starts by downloading the video at the maximum available resolution. CODETUBE uses the multimedia framework FFMPEG[3] to extract one frame per second, saving each frame in a `png` image. Given the set of frames in the video, we compare subsequent pairs of frames (f_i, f_{i+1}) to measure their dissimilarity in terms of their pixel matrices. If they differ by less than 10% we only keep the first frame in the data analysis since the two frames show *almost* the same information. This scenario is quite common in video tutorials where the image on the screen is fixed for some seconds while the tutor speaks. This optimization considerably reduces the computational cost of our process without losing important information. After obtaining the reduced set of frames to analyze, CODETUBE performs the following *information extraction steps.*

English Terms Extraction

We use the OCR (Optical Character Recognition) tool TESSERACT-OCR[4] to extract the text from the frame. OCR tools are usually designed to deal with text on white background (*i.e.,* paper documents). In order to cope with this, many OCR tools convert colored images to black and white before processing them. When using an OCR tool on video frames, the high variability of the background, and the potential low quality of a frame can result in a high amount of

---

[1] https://developers.google.com/youtube/v3/
[2] http://google2srt.sourceforge.net/en/
[3] http://www.ffmpeg.org/
[4] https://github.com/tesseract-ocr

Figure 8.4. Example frames from which CodeTube is able to extract code fragments.

noise. Thus, after splitting composite words—based on camel case or other separators—we use a dictionary-based filtering, to ignore strings that are invalid English words[5].

Java Code Identification

In principle, the output of the OCR could be processed to extract the depicted Java constructs. However, such output often contains noise. Figure 8.4 shows three frames containing Java code. In frame 1 the code occupies the whole screen, and there is a clear background: the noise of the OCR output is limited. The noise increases in the Frames 2 and 3, due to the buttons, menu labels, the graphics on the t-shirt, *etc.* To limit the noise produced by the OCR we identify the sub-frame containing code using two heuristics, *shape detection* and *frame segmentation*.

Shape Detection. We use BOOFCV[6] to apply shape detection on a frame, identifying all quadri- laterals by using the difference in contrast in the corners. This is typically successful to detect code editors in the IDE as in Frame 2.

Frame Segmentation. The shape detection phase could fail in identifying sub-frames with code. In Frame 3 of Figure 8.4 BOOFCV fails because of missing quadrilaterals. In this case, we apply a segmentation heuristic by sampling small sub-images having height and width

---

[5]We use the OS X English dictionary.
[6]http://boofcv.org/

equal to 20% of the original frame size and we run the OCR on each sub-image. We mark all sub-images S_m containing at least one valid English word and/or Java keyword and we identify the part of the frame containing the source code as the quadrilateral delimited by the top-left sub-image (*i.e.,* the one having the minimum x and y coordinates) and the bottom-right sub-image (*i.e.,* the one having the maximum x and y coordinates) in S_m.

Identifying Java Code. After identifying a candidate sub-frame, we run the OCR to obtain the raw text that likely represents code. Then, we use the island parser described in Chapter 6 on the extracted text to cope with the noise, the imperfections of the OCR, and the incomplete code fragments. The island parser separates invalid code or natural language (water) from matching constructs (islands), and produces a heterogeneous Abstract Syntax Tree (H-AST). By traversing the H-AST we can exclude water nodes and keep complete constructs (*e.g.,* declarations, blocks, other statements) and incomplete fragments (*e.g.,* partial declarations, like methods without a body). If we are not able to match complete or incomplete Java constructs with any of the described heuristics, we assume that the frame does not any contain source code.

8.2.2   Identifying Video Fragments

The *Video Fragments Identifier* detects cohesive fragments in a video tutorial using the previously collected information. We refer to Figure 8.5 to illustrate the performed steps. CODETUBE starts by identifying video fragments characterized by the presence of a specific piece of code. The conjecture is that a frame containing a code snippet is coupled to the surrounding video frames showing (parts of) the same code.

Identifying the video frames containing a specific code snippet presents non-trivial challenges. First, a piece of code could be written incrementally during a video tutorial: if writing a Java class in a video tutorial lasts 3 minutes, all frames in the 3-minute interval will contain snippets of code related to that class and thus should be considered as part of the same video fragment. However, such code snippets are different (*i.e.,* they contain different programming constructs) due to the incremental writing. Second, the tutor could, to provide a line-by-line explanation, scroll the code snippet shown on video. Again, this causes frames showing the same code snippet to show different "portions" of it. Last, the tutor could interleave two frames showing the same snippet of code with slides or other material (*e.g.,* the Android emulator).

CODETUBE overcomes these challenges and identifies video fragments characterized by the presence of a specific piece of code by comparing subsequent pairs of frames containing code to verify if they refer to the same code snippet. The frames depicted in red in Figure 8.5 represent "code frames", that is, frames containing code fragments. Given two code frames CODETUBE verifies if they contain at least one common complete or incomplete Java construct. If so, the two frames are marked as containing the same code component. If not, we cannot exclude that the two frames do not refer to the same code; We have to take into account (i) possible imprecisions of the OCR when extracting the source code from the two frames, *i.e.,* it could happen that a Java construct is correctly extracted only in one of the two frames, and (ii) the possibility that a scrolling from one frame to another has hidden some constructs in one of the two frames.

If the island parser fails in matching a common construct in the two frames, we compute the Longest Common Substring (LCS) between the pixel matrices representing the code frames. Specifically, we represent matrices as strings, where each pixel is converted to a 8-bit grayscale representation. If the LCS between the two frames includes more than α of the pixels in the frames, CODETUBE considers the two frames as showing the same code snippet.

The process adopted to tune the threshold α is reported in Section 8.2.5.
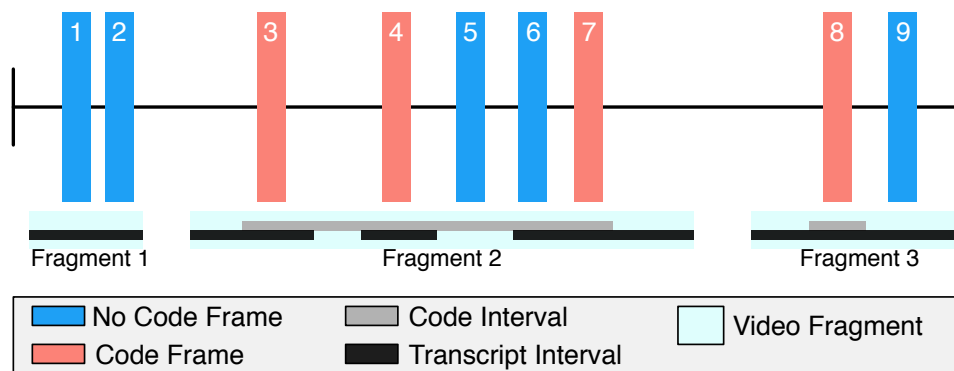
Figure 8.5. Identification of video fragments.

Note that the LCS is not affected by possible OCR imprecisions, and it does not suffer of problems related to the IDE scrolling, as shown in Figure 8.6 (in cyan the portion of the two frames identified as LCS). As a drawback, LCS is sensitive to zooming. Since the alignment of the proportions between two subsequent frames changes, LCS would fail in identifying a common part. Overall, given the advantages of the LCS over the Java constructs matching between the two frames via island parser, one may think that applying the LCS for each pair of code frames is the way to go. Unfortunately, the LCS is very expensive to compute due to the huge number of pixels composing a frame (a 1080p HD video has ∼2M pixels per frame), and estimating the LCS on each pair of code frames would require an unreasonable computation time.

For this reason, we adopt the LCS as a contingency strategy when the island parser is unable to identify common Java constructs in the two frames under analysis. To speed up the LCS computation we scale the frames to 25% of their size. In the example depicted in Figure 8.5, CODETUBE compares the code frame pairs (3,4), (4,7), and (7,8), identifying the first two pairs as containing the same code snippet. As highlighted by the grey line below the frames, it identifies the first two cohesive "code intervals", *i.e.*, the first going from frame 3 to frame 7 and the second containing frame 8 only. The "non-code frames" 5 and 6 (blue in Figure 8.5) are included in the first code interval, since they are surrounded by two code frames (4 and 7) containing the same snippet.

In a subsequent step CODETUBE analyzes the audio transcripts (black lines at the bottom of Figure 8.5) to refine the already identified code intervals (grey lines). CODETUBE identifies the audio transcripts starting and/or ending inside each code interval. The audio transcripts are provided in the SubRip[7] format when extracted from YouTube's videos. In the example reported in Figure 8.5, three audio transcripts are considered relevant when refining the code interval going from frame 3 to 7. CODETUBE uses the beginning of the first and the end of the last relevant audio transcript for a code interval to extend its duration and avoid that the code interval starts or ends with a broken sentence. The extended code interval represents an identified video fragment (Fragment 2—light cyan in Figure 8.5).

There might still be non-code frames in the video that have not been assigned to any video fragment (*e.g.,* frames 1 and 2 in Figure 8.5). These frames are grouped together on the basis of the audio transcript part they fall in. For example, the first two frames in Figure 8.5 are grouped in the same video fragment (Fragment 1), since they both fall in the same audio transcript part. As a final step, each subsequent pair of fragments is compared to remove very short video

---

[7] https://en.wikipedia.org/wiki/SubRip

```
public String read(File file) {
  fis = new FileInputStream(file);
  byte[] data =
      new byte[(int) file.length()];
  fis.read(data);
  fis.close();

  return new String(data, "UTF-8");
}
```

```
  byte[] data =
      new byte[(int) file.length()];
  fis.read(data);
  fis.close();

  return new String(data, "UTF-8");
}

public boolean isNull(Object obj) {
```

Figure 8.6. LCS between two frames showing the same code. The right frame is scrolled down by the tutor.

fragments and to merge semantically related fragments.

CODETUBE merges together two subsequent fragments if one of two conditions applies:

1. Their textual similarity (computed using the Vector Space Model (VSM) [BYRN99]) is greater than a threshold β. Each video fragment is represented by the text contained in its audio transcripts and in its frames (as extracted by the OCR). The text is pre-processed by removing English stop words, splitting by underscore and camel case, and stemming with the Snowball stemmer[8].

2. One of the two fragments is shorter than γ seconds. This is done to remove short video fragments that unlikely represent a complete and meaningful fragment of a video tutorial.

8.2.3   Features Computation for the Fragments Classification

Video Fragment Classifier is in charge of classifying them into one of the seven categories obtained as output of the study presented in Section 8.1 (see Table 8.4).

There are different aspects to take into account when devising an approach to automatically classify complex objects like video fragments. The information characterizing a video fragment is heterogeneous, and includes temporal aspects (*e.g.,* position in the video with respect to other fragments), structural features (*e.g.,* presence of shapes on the screen), semantic features (*e.g.,* textual topics), and information concerning code on screen. We present all the features involved in the construction of the feature vector that is used by the machine learning algorithm that enables the automatic classification of a given video fragment.

Temporal Features

It is natural to think that some types of video fragments have a tight relationships with their temporal position in the video, but also with their duration. Specifically, we consider the following three features:

Beginning Time. The relative beginning time of the video expressed as percentage of the whole video (*i.e.,* `begin_time/video_length`). Identifying the position of the fragments should help the classifier in identifying relationships between certain categories of video fragments

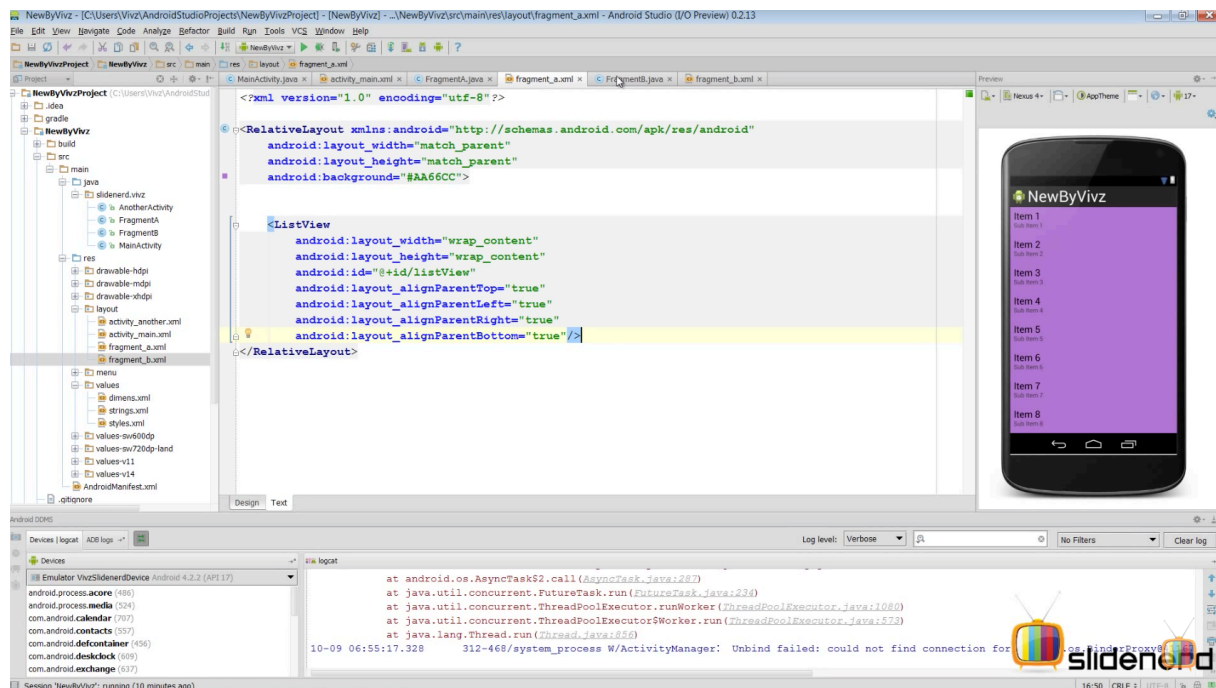---

[8]`http://snowball.tartarus.org`

Figure 8.7. A frame taken from a *code implementation* fragment.

and their temporal position within the video. For example, *introduction to the tutorial topic* and *closing of the tutorial* to the beginning and the end of the video, respectively.

Fragment Length. The length of the frame in seconds. Different fragment categories are likely to have different durations. For example, fragments related to the opening and closing of a tutorial are supposed to be short, while fragments showing *code implementation* activities are supposed to last longer since they constitute the core of a tutorial.

Fragment Coverage. The ration between the length of the video and the length of the fragment (*i.e.,* `fragment_length / video_length`) expressed as percentage. Although the coverage can be considered similar to the fragment length, it avoids possible issues related to video tutorials having a substantially different duration. Indeed, fragments extracted from long video tutorials are likely to be longer than those extracted from short video tutorials, despite their "type". This feature, being normalized on the video tutorial length, avoids this issue.

Structural Features

Another aspect to be considered is the structure of each frame composing the fragment. Specifically, different frames have different content of graphical elements. We focus on three different features:

Average Pixel Overlap. The overlap between all possible frame pairs in the fragment. The overlap is calculated pixel-wise. Only pixels in the same position and with the same color in two frames are considered overlapping.

Fragments categorized as *theoretical concepts* are likely to have a higher percentage of overlapping pixels between frames, since the tutor may show the same slide for several

seconds while discussing the concepts. Since we have discarded subsequent similar frames, we need to take this into consideration when computing such a feature.

For example, let us assume that our video fragment V is composed of frames F_1, F_2, and F_3 and that we have discarded F_2 as being too similar to F_1. Also, suppose that the pixel overlap between F_2 and F_3 is 60%. We consider 100% of pixel overlap between F_1 and F_2, thus obtaining 80% as average pixel overlap for the fragment.

Average Number of Quadrilaterals. The average number of quadrilaterals identified by shape detection analysis in the fragment's frames. Figure 8.7 shows an example of frame taken from a fragment tagged as *code implementation.* There are several quadrilaterals corresponding to code editor, console output, package explorer, and the UI designer. The number of quadrilaterals is likely to discriminate fragments in which the IDE is shown (*e.g.,* a *code implementation* fragment) from the others. Also, the number of quadrilaterals on the screen also helps in discriminating *execution of the implemented code* from *code implementation.* Indeed, we expect the execution of the implemented code to open new windows (*e.g.,* the Android emulator) on the screen. In this case we adjust our computation to take into account the removal of (quasi-)identical frames. In this case, if our video fragment is composed by the frames F_1 (4 quadrilaterals), F_2 (4 quadrilaterals), and F_3 (1 quadrilateral) and F_2 has been removed as too similar to F_1, we consider F_1 twice in the computation of the average: $(4+4+1)/3=3$. This approach is exploited in all features requiring the computation of the average between properties of the fragment's frames.

Average Largest Quadrilateral. The average size of the quadrilaterals shown in the fragment's frames, expressed as a percentage of the total screen area. The goal is to understand if the IDE is the foreground of the frame. In Figure 8.7 the code editor is the largest quadrilateral and occupies about 32% of the frame space. Having a high average coverage of the largest quadrilateral could imply having an IDE on the foreground and therefore helping the classifier in discriminating the categories involving development (*e.g., code implementation*, *execution of the implemented code*, and *dealing with errors*). As for the previous structural features, we considered frames that were removed, and we replicate the value of their predecessor in the calculation of the average.

Code Features

Using the information extracted with the island parser, we compute the following features:

Average Constructs. The average number of code constructs found in the fragment's frames. The classifier could use this feature to discriminate between fragment categories likely to show a lot of code (*e.g.,* code implementation), and categories with less or no code (*e.g.,* topic introduction, working environment setup). We considered frames that were removed, by replicating the value of their predecessor.

Average Specific Node Types. The average number of occurrences of some specific AST nodes in the fragment's frames. In particular, we count *identifiers*, *imports*, *class declarations*, *method declarations*, *blocks*, *statements*, *stack traces*, *XML tags*, and *JSON constructs.* The idea is to differentiate the type of constructs for the different categories. For example, categories like *theoretical concepts*, *topic introduction*, or *closing* are unlikely to have complex constructs like declarations and statements. Fragments *dealing with common errors* are likely to contain more constructs related to stack traces. Instead, *JSON constructs* or

XML tags could be shown in a frame when detailing some specific portions of the implementation (*e.g.,* JSON for illustrating access to remote services, or XML for Android app permissions or activity design). Also, *XML constructs* can be shown in the context of the *environment setup.* As for the previous structural features, we considered frames that were removed, and we replicate the value of their predecessor in the calculation of the average.

Semantic Features

The last set of features concerns the semantics of a fragment as captured by its textual content. Textual content relates to (i) the audio transcript, when available, and specifically the transcript subset related to the segment time interval, and (ii) the text contained in the frames.

Considering the occurrences of each term as a feature for the machine learning algorithm would inevitably hinder the performance of any classifier, *e.g.,* by introducing problems related to synonymy or polysemy. A viable solution is to reduce the dimensionality of the semantic features by substituting terms with a set of topics extracted from the fragments. In our approach we use Latent Dirichlet Allocation (LDA) [BNJ03], an unsupervised topic modeling technique that, as suggested by Blei *et al.* [BNJ03], can be used as feature reduction approach for terms. We use the Stanford Topic Modeling Toolbox[9] configured to identify seven topics from the fragments corpus. The number of topics has been selected according to the number of labels resulted from the open coding session and reported in Table 8.4. Although a near-optimal configuration of LDA could require a proper setting—*e.g.,* through search-based optimization techniques [PDO+13]—in this work we have set the number of topics equal to the number of expected categories, an approach already followed when LDA has been used to categorize text [GTGZ14].

8.2.4   Classifying Video Fragments

The starting point for building a classifier able to discriminate between the different types of video fragments is a *training set* built from a collection of video fragments, their temporal, structural, code, and semantic features, and their respective category (*e.g., code implementation*). While the computation of the video fragments' features is automated, the fragments' categories must be manually assigned (*e.g.,* by following a process similar to the one presented in Section 8.1).

Once a training set is available, a supervised learning algorithm is run on it. Our approach uses the *Weka* [WF11] implementation of the Random Forest machine learning algorithm [Bre01], which builds a collection of decision trees with the aim of solving classification-type problems, where the goal is to predict values of a categorical variable from one or more continuous and/or categorical predictor variables. The categorical dependent variable is represented by the video tutorial fragment category (*e.g., code implementation, execution of the implemented code, etc.*), and we use the features described in Section 8.2.3 as predictor variables.

We have chosen Random Forest after experimenting with different machine learner algorithms, and in particular SimpleCart, J48, Bayesian Network, Logistic Regression, and Bagging classifiers The built classification model can then be used to classify new video fragments. To do so, we extract from the test set fragments the same set of features considered in the training set. Based on these values, the Random Forest obtained in the training phase is used to automatically determine the video fragment category.

Since some of the features we considered might correlate, we perform an *information gain* feature selection process [Mit97] aimed at removing all features do not contributing to the information available for the prediction of the video fragment category.

---

[9]`http://nlp.stanford.edu/software/tmt/`

Table 8.6. Parameter tuning intervals.

Parameter	Min	Max	Δ
α	5%	50%	5%
β	10%	80%	5%
γ	1,000s	120,000s	10,000s

Also, when training the model we check the distribution of training set samples across the seven fragment categories. In case of imbalanced dataset, we apply a rebalancing technique as it will be detailed in Section 8.3.1.

8.2.5 Tuning of CodeTube Parameters

The performance of CODETUBE depends on three parameters that need to be properly tuned:

α – minimum percentage of LCS overlap between two frames to consider them as containing the same code fragment;

β – minimum textual similarity between two fragments to merge them in a single fragment;

γ – minimum video fragment length.

To identify the most suitable configuration, one of the authors—who did not participate in the approach definition—built a "video fragment oracle" by manually partitioning a set of 10 video tutorials into cohesive video fragments[10]. Then, we looked for the CODETUBE parameters configuration best approximating the manually defined oracle. A challenge in this context is how to define the "closeness" of the automatically- and manually-generated video fragments.

Estimating Video Fragments Similarity

A video can be seen as a set of partitions (video fragments) of frames, where each frame belongs to only one partition, *i.e.,* the generated video fragments are clusters of frames. To compare the closeness of the video fragments generated by CODETUBE and those manually defined in the oracle, we used the MoJo effectiveness Measure (MoJoFM) [WT04], a normalized variant of the MoJo distance, computed as:

$$MoJoFM(A,B) = 100 - \left(\frac{mno(A,B)}{max(mno(\forall E_A, B))} \times 100 \right) \qquad (8.1)$$

where $mno(A, B)$ is the minimum number of *Move* or *Join* operations needed to transform a partition A into a partition B, and $max(mno(\forall E_A, B))$ is the maximum possible distance of any partition A from the partition B. Thus, $MoJoFM$ returns 0 if A is the farthest partition away from B, and returns 100 if A is exactly equal to B.

While MoJoFM is suitable to compare different partitions (video fragments) of the same elements (frames), we must take into account that video fragments are characterized by a constraint of sequentiality (*i.e.,* they can only contain subsequent frames). This could lead the MoJoFM to return high values (similarity) even when applied to two totally different video partitions. For example, consider the video frames $F = \{1, 2, 3, 4, 5, 6\}$ and two sets of video fragments

---

[10]These 10 videos are not part of the 136 considered in the study presented in Section 8.1.

where the first set, $A = \{1, 2, 3, 4, 5, 6\}$, contains a unique partition (video fragment) with all the elements (frames), and the second set, $B = \{\{1, 2, 3\}, \{4, 5, 6\}\}$, contains 2 partitions of size 3. Since the MoJoFM is not a symmetric function, it would return $MoJoFM(A, B) = 25.0$ and $MoJoFM(B, A) = 80.0$, *i.e.,* two different values, despite the fact that the two partitions are the same. Keeping a one-way comparison between the oracle and the obtained video fragments undermines the tuning phase. To avoid this, yet keeping a margin of approximation, both sides of the MoJoFM should be taken into account. Two sets of fragments will tend to have the same value if they are close in their partitioning. For this reason, we calculate the similarity in both directions and compute their mean value:

$$closeness(A, B) = \frac{MoJoFM(A, B) + MoJoFM(B, A)}{2} \tag{8.2}$$

In doing so, spikes of high values for the *MoJoFM* between two sets of video fragments for one direction are lowered or preserved depending on the opposite.

Estimating the Most Suitable Parameter Configuration

For each parameter, we identified a set of possible values. Table 8.6 shows the intervals we adopted, and the step (Δ) used whenever a new combination is generated. In total, we experimented 1,800 different parameter combinations, adopting the one with the top ranked MoJoFM (α =5%, β =15%, γ =50s) for the full-fledged analysis phase.

Figure 8.8. CodeTube: User interface.

8.2.6 Integrating Other Sources of Information

CODETUBE can be enriched by mining other online resources, as our long-term goal is to offer a *holistic* point of view on the information at disposal, also because we argue that no single type of resource can offer exhaustive assistance. To illustrate this, we added as an additional online information source the Stack Overflow data dump. We mined and extracted discussions related

to the topics of the extracted video tutorials, pre-processed them to reduce the noise, and made them available to CODETUBE.

The last step in the data pre-processing of CODETUBE consists in indexing both the extracted video fragments and the Stack Overflow discussions, using LUCENE[11], where each video fragment is considered as a document. For Stack Overflow we separately index each question and answer for each discussion. The text pre-processing phase is identical to the one explained in Section 8.2.2. The text indexed for a video fragment is represented by the terms contained in its frames and audio transcripts. The text indexed for the Stack Overflow post is represented by the terms they contain.

8.2.7   The CodeTube User Interface

CODETUBE provides a service that allows user to search, watch, and navigate the different fragments of a video. The user can input a textual query and select via checkboxes the types of video tutorials she is interested in (one or more of the seven types implemented in CODETUBE). CODETUBE will provide a list of relevant video tutorial fragments (search results) from which the user can select the one she is interested in watching. When a video fragment is selected for watching from the search results, the GUI depicted in Figure 8.8 is shown: CODETUBE uses the YouTube player (1) provided by the YouTube API[12]. The video starts at the time devised by the selected fragment. CODETUBE provides an additional controller (2) to visualize the timestamps of the fragments identified by our approach, select a specific fragment, or move to the next/previous fragment. During the video playback, the selector underneath the video player keeps the pace of the video timing and shows the current fragment. When a new fragment is reached, or the user jumps to it, CODETUBE automatically extracts a query from the text contained in the fragment (*i.e.,* transcripts and OCR output of the frames it contains), queries both the index of Stack Overflow and of the video fragments, and updates the related discussions (3) and the suggested YouTube video fragments (4). A search bar (5) is always available to the user to run new queries.

8.3   Study I: Identify and Classify Video Fragments

The *goal* of this study is to evaluate CODETUBE with the *purpose* of determining its ability to (i) extracted meaningful video fragments, and (ii) correctly classify video tutorials. The two research questions (RQ) the study aims to answer are:

RQ$_1$: *To what extent are the automatically identified video tutorial fragments overlapped with respect to manually identified ones?* The first research question assesses the overlap between the video tutorial fragments automatically extracted by CODETUBE and those manually identified by people watching the same tutorial. A high overlap would indicate the ability of CODETUBE to split video tutorials as humans would do.

RQ$_2$: *How accurate is CODETUBE in classifying video tutorial fragments in the considered categories?* This research question (i) assesses the accuracy of our technique in classifying video tutorial fragments in the seven categories listed in Table 8.4, and (ii) investigates the importance of the different categories of features (*i.e.,* temporal, structural, code, and semantic) described in Section 8.2.

---

[11]`https://lucene.apache.org/`
[12]`https://developers.google.com/youtube/js_api_reference`

Table 8.7. Features selection results.

Feature Type	Name	Selected	
Temporal	Beginning Time	✓	1
	Fragment Length	✓	2
	Fragment Coverage	✓	4
Structural	Average Pixel Overlap	✓	3
	Average Number of Rectangles	✓	8
	Average Largest Rectangle	✓	7
Code	Average Constructs	✓	9
	#identifiers	✓	15
	#class declarations	✓	17
	#method declarations	✓	19
	#blocks	✓	13
	#statements	✓	10
	#imports	✓	20
	#stack traces	✗	–
	#JSON constructs	✗	–
	#XML tags	✗	–
Semantic	topic 1	✓	16
	topic 2	✓	5
	topic 3	✓	14
	topic 4	✓	18
	topic 5	✓	12
	topic 6	✓	11
	topic 7	✓	6

8.3.1 Study design and procedure

To answer our research questions we use the dataset of manually identified and classified video tutorial fragments we obtained through the tagging study and coding activity described in Section 8.1. The dataset consists of 748 manually tagged fragments from 136 video tutorials related to three main topics: (i) Java programming, (ii) JSPs and Servlets, and (iii) Android development. The 748 fragments are distributed across the seven categories of video fragments, as shown in Table 8.4. We run CODETUBE on this dataset in order to automatically identify and categorize video fragments.

Then, to answer $\mathbf{RQ}_1$, we computed the MoJoFM [WT04] between the fragments automatically extracted by CODETUBE and the ones manually identified by the study participants, by applying the same procedure adopted for the parameters' tuning (see Section 8.2.5). We show box plots of the MoJoFM achieved by CODETUBE over the 136 subject video tutorials.

To answer $\mathbf{RQ}_2$, we performed a 10-fold cross validation over the dataset, computing the overall average accuracy of the model when (i) only relying on temporal, structural, code, and semantic features in isolation, (ii) combining the four categories of features in pairs (six possible pairs) and in groups of three (four possible groups), and (iv) considering all of them. We performed a feature selection process before running the 10-fold validation. Table 8.7 shows selected and discarded features (and their rank as provided by the *information gain*) when considering the complete dataset. The temporal feature is top ranked as some fragments occur in specific time frames of the video. Some features that seem to be related (*e.g.,* fragment length
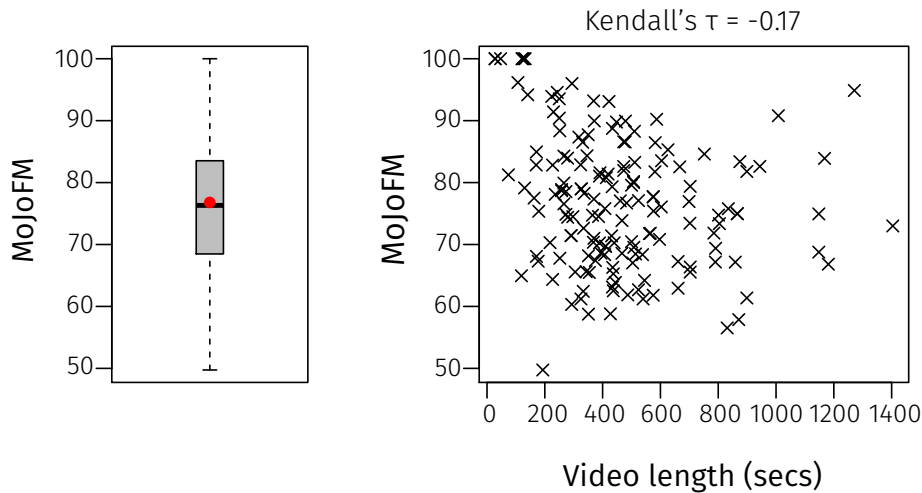
Figure 8.9. RQ_1: MoJoFM achieved on the 136 video tutorials and scatterplot between MoJoFM and video length.

and coverage) are both taken into account and ranked in the top position, hence indicating that they bring complementary information. Instead, data-specific features (*e.g.,* json constructs and xml tags) do not bring information for discriminating the considered categories. The same happens for stack traces, that could have been potentially useful for discerning error scenarios. One possible interpretation is that the OCR failed to successfully capture stack traces, *e.g.,* because the console output is not fully visible in the IDE.

Since our dataset is strongly unbalanced (see Table 8.4), we balanced the training set at each iteration (*i.e.,* for each of the ten folds) by exploiting the Synthetic Minority Oversampling TEchnique (SMOTE) [CBHK02]. SMOTE rebalances the training set by creating artificial instances obtained by joining nearest neighbors of the minority class instances. While we did balance the training set to build the classifier, the test set was never modified to avoid any bias.

We assess the overall performance of the model with its average accuracy. Also, we dig into the results by presenting (i) the obtained confusion matrix, (ii) the model accuracy for each of the seven considered fragment categories, and (iii) the Area Under the ROC curve (AUROC) [Bra97] obtained for each category as well as for the overall model. An AUROC of 0.5 indicates a model having the same prediction accuracy in identifying true positives as a random classifier. A prefect model (*i.e.,* zero false positives and zero false negatives) has instead AUROC=1.0. Thus, the closer the AUROC to 1.0, the higher the model performance.

8.3.2   Study results

This section discusses the results achieved in our study with the aim of answering our two research questions.

RQ_1: To what extent are the automatically identified video tutorial fragments overlapped with respect to manually identified ones?

Figure 8.9 shows (i) the box plots of the MoJoFM obtained when comparing the video fragments automatically extracted by CODETUBE with those manually defined by watching the 136 video

tutorials subject of this study, and (ii) the scatterplot of the MoJoFM and video length, useful to investigate possible relationships between the accuracy of CODETUBE in identifying video fragments and the video length.

On average, CODETUBE achieves 77% of MoJoFM (median=76%), suggesting a high similarity between manually- and automatically-identified video fragments. In seven cases, the video partition proposed by CODETUBE is exactly the same manually defined by participants and, for twenty videos, the MoJoFM is higher than 90%. Clearly, there are cases in which CODE-TUBE fails in identifying meaningful fragments, as it happens for the five video tutorials in which it achieves a MoJoFM lower than 60%. We looked into these cases to understand the reasons behind such low CODETUBE performance. We identified three main situations in which CODETUBE clearly show limitations in identifying meaningful video fragments:

1. *Very low quality of the video.* An example is represented by the video tutorial having id `-VRUX-iSPWc`[13]. The low quality of the video makes it difficult to extract meaningful text with the OCR, thus substantially limiting the information available to the CODE-TUBE *Video Fragments Analyzer.*

2. *Zooming on the screen.* In some videos the tutor zooms in and out the screencast, thus making very challenging the identification of video fragments. Indeed, zooming at different levels in different frames clearly limits the effectiveness of some of the heuristics adopted by the *Video Fragments Analyzer* (*e.g.,* the LCS). An example of such a scenario is represented by the video tutorial `xuX96Lik3Co`[14].

3. *Continuous shifting of the portion of the screen captured in the screencast.* In some tutorials the screencast does not capture the whole screen, bug just the portion of screen surrounding the mouse pointer. This results in a continuous shifting of the part of the screen shown in the tutorial, something that causes problems similar to the ones already discussed for the screen's zoom. The video tutorial `-L8FAKadrhg`[15] is an example of such a scenario.

The scatterplot in the right part of Figure 8.9 does not show any strong relationship between the MoJoFM and the length (in seconds) of the video tutorials. Also the Kendall's τ coefficient (-0.17) confirms that while CODETUBE is slightly more precise in the identification of video fragments on shorter videos, there is a very weak negative correlation between MoJoFM and video length.

RQ$_2$: How accurate is CodeTube in classifying video tutorial fragments in the considered categories?

Table 8.8 reports the accuracy (*i.e.,* percentage of correctly classified instances) and AUROC obtained by our approach when relying on different sets of features.

When exploiting the temporal, structural, code, and semantic features in isolation, the best performance are provided by the temporal features, with 56% of accuracy and AUROC=0.82. While this result might look surprising, temporal features can be very effective in identifying at least two of the categories considered in our study: the *introduction of the tutorial topic* and the *closing of the tutorial.*

Indeed, 85% of the tutorials start with an introduction to the tutorial topic and 35% ends with a closing part generally featuring a summary of the tutorial and/or information about future

---

[13] https://www.youtube.com/watch?v=-VRUX-iSPWc
[14] https://www.youtube.com/watch?v=xuX96Lik3Co
[15] https://www.youtube.com/watch?v=-L8FAKadrhg

Table 8.8. RQ$_2$: Performance achieved when using different combinations of features.

Considered set of features	Accuracy	AUROC
Temporal	56%	0.82
Structural	30%	0.65
Code	41%	0.68
Semantic	40%	0.70
Temporal+Structural	61%	0.85
Temporal+Code	67%	0.88
Temporal+Semantic	66%	0.84
Structural+Code	46%	0.74
Structural+Semantic	48%	0.75
Code+Semantic	45%	0.75
Temporal+Structural+Code	68%	0.88
Temporal+Structural+Semantic	68%	0.88
Temporal+Semantic+Code	70%	0.90
Structural+Semantic+Code	52%	0.79
Temporal+Structural+Semantic+Code	**72%**	**0.92**

tutorials that will be published. Thus, it is quite simple for the model to learn how to spot out these types of fragments by exploiting temporal features—*e.g.,*if *beginning time*[16] < 0.1 `then` fragment type `is` *introduction of the tutorial topic.* Temporal features also help in identifying fragments dealing with *code implementation* activities. For this fragment type, our approach learns, by exploiting temporal features, that if a fragment starts during the first half of the tutorial (*i.e., beginning time* < 0.58) and it lasts for over 40% of the overall tutorial length (*i.e., fragment coverage* > 0.42) it likely represents a *code implementation* fragment. This makes sense since code implementation often represents the bulk of software development video tutorials.

The other sets of features (*i.e.,* structural, code, and semantic) obtain substantially lower performance than temporal features when used in isolation. However, for some specific categories of fragments, they perform as well as or even better than the temporal features.

For example, semantic features help in characterizing fragments describing how to *deal with common errors.* This is possible thanks to a specific LDA topic (*topic 2*) described by key words such as *exception, try,* and *throw.* This topic is exploited by our approach in the identification of video fragments explaining how to deal with common errors. However, as it will be further discussed later, this is not enough for our technique to provide a high accuracy in the identification of this type of fragments. This is mainly due to the fact that *topic 2* does also play a major role in tutorial fragments dealing with implementation activities, thus leading to a high number of misclassifications.

Code features are the best ones in identifying *code implementation* fragments: Our approach learns that a high number of code snippets shown on the screen for the whole fragment duration likely indicates its focus on implementation tasks. Finally, structural features provide the lowest accuracy and, when used in isolation, exhibit very low accuracy in the identification of all categories of video fragments.

When combining the four sets of features in pairs, performance are boosted up to 67% of accuracy and 0.88 of AUROC (obtained when combining temporal and code features), indicating

---

[16]Beginning time expresses the relative beginning of the video fragment as percentage of the video tutorial length.

Table 8.9. RQ$_2$: Confusion Matrix and AUROC per each Category when Using all Features.

	CT	DE	ES	EIC	CI	TC	ITT	AUROC
Closing of the Tutorial (**CT**)	**37**	0	0	5	3	0	0	0.98
Dealing with Errors (**DE**)	0	**7**	0	0	8	3	1	0.88
Environment Setup (**ES**)	1	0	**19**	0	7	4	7	0.88
Execution Implemented Code (**EIC**)	11	2	3	**83**	17	6	1	0.91
Code Implementation (**CI**)	7	8	18	26	**200**	20	6	0.89
Theoretical Concepts (**TC**)	1	4	0	6	14	**55**	7	0.90
Introduction tutorial topic (**ITT**)	0	1	3	0	2	6	**131**	0.98

quite good performance of the built model. Again, it is clear the major role played by the temporal features. Indeed, the three models exploiting them have AUROC≥0.84 as compared to the 0.74 and 0.75 obtained in the two models do not exploiting temporal features. The accuracy further increases up to 70% (AUROC=0.90) when using three groups of features at a time (temporal, semantic, and code features), reaching its maximum value (72%) when all four sets of features are exploited. In this case, the built model exhibits a quite high AUROC of 0.92.

Table 8.9 reports the confusion matrix obtained by this comprehensive model. Also, it shows the AUROC for each of the seven categories of fragments. As expected, our approach is very effective in identifying fragments related to the *introduction to the tutorial topic* (accuracy=82%, AUROC=0.98) and to the *closing of the tutorial* (accuracy=92%, AUROC=0.98). As previously discussed, this very high AUROC is possible thanks to the temporal features able to carefully discriminate video fragments of these two types.

Classification performance are also very good for fragments dealing with the *execution of the implemented code* (accuracy=67%, AUROC=0.91), the explanation of *theoretical concepts* (accuracy=63%, AUROC=0.90), and *code implementation* activities (accuracy=70%, AUROC=0.89). Concerning the former (*e.g.,* the execution of an implemented app in the Android emulator), we expected structural features to be highly discriminating. This is because our conjecture was that more often than not the execution of the code would have opened a new window, thus resulting in more quadrilaterals shown on the screen with respect, for instance, to the ones present during implementation activities. However, these features resulted to be useless for the identification of these fragments. This is due to the fact that often the implemented code is executed directly inside the IDE's console (always shown on the screen) without opening a new window. This makes difficult to discern this situation from a code implementation with no execuyetion activity. Currently, our approach identifies these fragments as "short implementation activities" (*i.e.,* they are characterized exactly as code implementation fragments, but they are much shorter). Probably, other features should be thought to increase the classification accuracy for this category of fragments.

An effective identification of *code implementation* fragments is possible with a combination of temporal and code features. For example, one of the rules used in a decision tree generated by our approach identifies implementation fragments are those lasting at least 42% of the overall video length, starting during the first half of the tutorial, and containing at least one code statement and one block statement.

Finally, while still being acceptable, performance decrease when categorizing fragments related to the *environment setup* (accuracy=50%, AUROC=0.88) and to *common errors* one could encounter during implementation activities (accuracy=37%, AUROC=0.88). In these cases, we expected semantic features to help in the classification of these fragments. As previously mentioned, semantic features only partially help in the identification of fragments related to imple-

mentation errors. A deeper investigation revealed that the limited contribution of the semantic features is due to the high imprecision of the OCR in extracting terms from the video frames. As previously explained, OCR tools are usually designed to deal with text on white background (*i.e.,* paper documents). When using an OCR tool on video frames, the high variability of the background can result in a high amount of noise. Probably, the accuracy of our approach could strongly benefit from the implementation of more robust OCR tools designed to deal with such a noise.

8.4   Study II: Intrinsic evaluation with users

In our previous study we assessed the accuracy of CodeTube in (i) identifying meaningful video fragments, and (ii) correctly classify video fragments in the seven considered categories. In this study we dig deeper in the quality of the extracted video fragments, looking at their cohesiveness, self-containment, and relevance to a query as perceived by developers. Also, we assess the relevance and complementarity to the video fragments to the Stack Overflow discussions recommended by CodeTube.

The three research questions (RQ) the study aims to answer are:

RQ$_3$: *To what extent are the extracted video tutorial fragments cohesive and self-contained?* This research question aims at assessing the capability of CodeTube to extract fragments that, as explained in Section 8.2 are on the one hand cohesive—*i.e.,* related to a very specific (sub)topic of the tutorial—and on the other hand self-contained, *i.e.,* they can be understood without watching the rest of the video.

RQ$_4$: *To what extent are the Stack Overflow discussions identified by CodeTube relevant and complementary to the linked video fragments?* The purpose of this question is to assess the capability of CodeTube to link video tutorial fragments to relevant Stack Overflow discussions. Our aim is to determine whether the textual content of the video tutorial fragment can be used to retrieve such discussions. Also, to determine the usefulness of a multi-source recommender like CodeTube, we are interested to understand whether the Stack Overflow discussions provide complementary information with respect to the video tutorial.

RQ$_5$: *To what extent is CodeTube able to return results relevant to a textual query?* This question assesses the CodeTube's retrieval capabilities over the indexed video fragments, mainly for the purpose of determining whether the indexed textual corpus allow to find relevant video fragments, and whether such fragments are at least as relevant as those returned by YouTube with the same query.

The *context* of the study consists of *participants* and *objects*. The *participants* have been identified using convenience sampling among personal contacts of the authors, and by sending invitations over mailing lists for open-source developers. In total, 40 participants completed the survey. The *objects* of the study are a set of 4,747[17] video tutorials about Android development indexed in CodeTube. From these video tutorials, CodeTube extracted a total of 38,783 fragments. Note that in this study we chose to focus on video tutorials dealing with a specific technology (*i.e.,* Android) and we only involved participants having experience with such a technology. This was a constraint to ensure a good assessment of the video tutorial fragments and Stack Overflow discussions identified by CodeTube.

---

[17]The number of videos and related fragments refers to a previous publication [PBM$^+$16a]

8.4.1 Study design and procedure

The study has been conducted using an online survey questionnaire, through which we asked questions to the potential respondents to assess the results of CODETUBE. The survey questionnaire is composed of two sections, preceded by preliminary assessment of the primarily activity (industrial/open source developer, student, academic), programming experience, and specific experience about Android development of respondents. This preliminary section also included questions having an exploratory nature and aimed at understanding (i) how often and in which circumstances respondents use video tutorials and Q&A Websites, (ii) whether they found useful information there, and (iii) how they react to video tutorials being too long (*e.g.,* scroll it, watch it anyway, or give up). We also asked participants what the main points of strength and weakness of video tutorials are, compared to standard documentation and Q&A Websites.

The first section shows to respondents three video fragments extracted by CODETUBE, as well as the original video tutorial from YouTube. Then, it asks (**RQ$_3$**) whether the fragment is cohesive and self-contained. For each video fragment, we also show the top-three relevant Stack Overflow posts, and ask (**RQ$_4$**) to what extent they are relevant and complementary to the video tutorial fragments. Our aim is to determine whether the textual content of the video tutorial fragment can be used to retrieve relevant discussions. For each respondent, the this section is repeated for two video tutorials randomly chosen from a sample of 20 video tutorials randomly selected from the 4,747.

The second section aims to assess the relevance of the top-three returned video fragments to a given query (**RQ$_5$**). As a baseline for comparison, we evaluate the relevance of the top-three videos returned by YouTube using the same query. The query shown to each respondent is sampled from a set of 10 queries formulated by graduate students at Florida State University, having a long experience in Android development. The queries are related to typical Android problems, *e.g.,* sending logs to servers, initiate activities in background, animate transitions, access accelerometer data, stopping background services, or modifying the UI layout.

The queries are generic, and YouTube is likely able to return as relevant results as CODE-TUBE. Only specific queries, referring to code elements—not contained in YouTube metadata—would show the advanced of the indexing capabilities of CODETUBE. Instead, we are interested in showing that, for the typical queries a developer formulates, CODETUBE returns at least as relevant as YouTube, but consisting in shorter, cohesive and self-contained fragments.

Finally, after the second section, we asked the respondents to evaluate, through an open comment, the main points of strength and weakness of CODETUBE.

All the assessment-related questions follow a three-level Likert scale [Opp92], *e.g.,* "very cohesive", "somewhat cohesive", and "not cohesive". We limit the number of video fragments, Q&A discussions and queries for each respondent to avoid the questionnaire being too long. Before sending the questionnaire to perspective respondents, we ran a pilot study to assess its estimated duration, which resulted to be between 25 and 40 minutes.

The questionnaire was then uploaded on the QUALTRICS[18] online survey platform, and a link to the questionnaire was sent via email to the invitees. We made it clear that anonymity of participants was presented and data were only published in aggregate form. The *Qualtrics* survey platform allowed us to achieve randomization and balancing, by automatically selecting video tutorials (with related Stack Overflow discussion) and queries to be evaluated by each respondent. After sending out the invitation, invitees had two weeks to respond.

---

[18] https://az1.qualtrics.com

8.4.2   Study results

Out of the 40 study participants, 6 declared to have no experience in Android development. Since the video tutorials considered in the study were not introductory but related to specific Android topics, we excluded their answers. Excluding these, we collected a total of 180 video tutorial fragment evaluations (with respect to their cohesiveness and self-containment), 540 Stack Overflow discussion evaluations, and 90 video tutorial fragment evaluations with respect to a query. Ideally, we could have collected more evaluations, but we have to consider that each of them requires respondents to watch a video tutorial fragment (and in the case of the queries also the whole video tutorial itself), hence we had to be realistic in the workload required by the targeted respondents. With such numbers and given our design, each fragment and SO discussion received a number of evaluations varying between 3 and 5, except for 3 videos and 2 queries, that, due to the exclusion of some participants motivated above, received less than 3 evaluations. These videos and queries were excluded from the analysis. With a set of videos smaller than our 20 we could have obtained more responses per fragment and SO discussions. We decided to favor the evaluation of a relatively larger set—and variety, hence more generalizability—of videos rather than having more responses and therefore more reliable evaluation for each video.

The population who completed our survey is composed of 70.6% of professional and open source developers, 17.6% of master students, and 11.8% of PhD students. The majority of developers in the population guarantees, on average, a higher level of experience: 32.3% of the population has more than 10 year experience, 17.5% has between 5 and 10 years, 38.3% between 3 and 5 years, 11.8% between 1 and 3 years. No one declared less than 1 year of programming experience. When asked about Android programming experience, the majority (38.3%) declared less than 1 year of experience, followed up by 23.5% of respondents with more than 3 years experience, 20.5% between 2 and 3 years, and 17.6% between 1 and 2 years of experience.

The participants use video tutorials either on a weekly (38.2%) or monthly (35.3%) basis. 3% declared to use video tutorials on a daily basis; nobody declared to never use them. Video tutorials are unlikely to help bug/error fixing (5%), but are the primary means to learn new concepts (43%).

When asked to provide open comments on the weaknesses and strengths of video tutorials, respondents pointed out different key aspects. The primary point of strength is the step-by-step nature of a video. One respondent wrote *"As opposed to Q&A Websites, video tutorials describe a complete process step-by-step. The visualized flow of actions is particularly useful in setting up working environments"*, another emphasized the *"possibility to see the complete interaction of the developer with the IDE"* and *"how a specific library is imported before it is used in the code. This does not hold when you simply copy and paste code from Websites"*. Another point of strength identified by respondents concerns the guidance given by a tutor. One respondent reported that *"there is a 'real' person talking with you, so it is easy to learn new concepts"*, while another respondent emphasized the fact that *"you can see what the tutor does"*.

The primary weakness identified by respondents concerns time. When a video tutorial is too long, respondents said they would either try to scroll it to seek the relevant information (47%), or give up to find alternative sources (53%). Nobody opted for the third option, *i.e.,* watching the whole video anyway. Respondents generally consider videos too long and slow and not suited *"if you need to quickly solve a problem"*, or *"if you need just a small piece of information"*. One of the respondents reported how *"due to time constraints during software development I cannot always watch the entire tutorial"*. The lack of searching and indexing functionalities of the contents of a video is also considered a weakness. One of the respondents claimed that *"browsing is not easy, unless the video has an index to navigate through the concepts/sections in the video"*, while
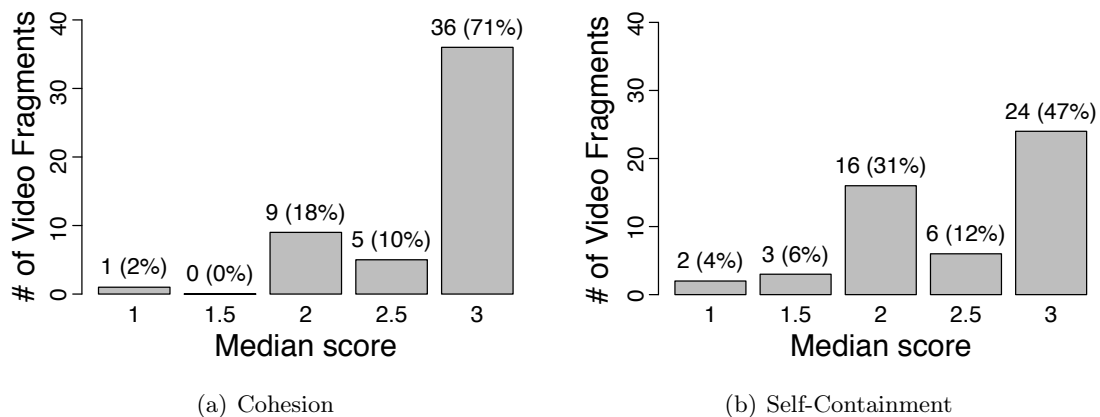
(a) Cohesion

(b) Self-Containment

Figure 8.10. RQ_3: Distribution of median cohesion and self-containment scores for the assessed video fragments.

another highlighted how *"searching for a particular piece of information in the whole video is much harder than doing the same in a text document"*.

RQ_3: To what extent are the extracted video tutorial fragments cohesive and self-contained?

Figure 8.10(a) shows the distribution of median perceived cohesiveness scores for the 51 fragments of the 17 videos that received at least three evaluations. The first quartile, median and third quartile of the distribution are 2, 3, and 3, respectively. A large majority (71%) of the evaluated fragments achieved a score of 3 (cohesive), and only one fragment was considered as not cohesive.

Figure 8.10(b) shows the distribution of the median self-containment score of the video fragments as provided by the evaluators. In this case, the first quartile, median, and third quartile are 2, 2.5 and 3, respectively. As one can notice from the figure, the proportion of video fragments that received a median score of 3 is lower than for cohesiveness (47%). This is not surprising because obtaining self-contained fragments—and hence understandable without watching the rest of the video—is more challenging than achieving a high cohesiveness. Nevertheless, the achieved cohesiveness is overall more than reasonable as 59% of the fragments achieve a score greater than 2, and only 10% of them were considered as not self-contained (score less than 2).

RQ_4: To what extent are the Stack Overflow discussions identified by CodeTube relevant and complementary to the linked video fragments?

Figure 8.11(a) shows the distribution of the median perceived relevance of the Stack Overflow discussions associated to the video fragments of each video tutorial considered in the study. The distribution first quartile is 2, the median 2 and the third quartile 3. On the one hand, the perceived relevance is relatively low, with only 38% of the Stack Overflow discussions achieving a median relevance of 3.

On the other hand, if we look at Figure 8.11(b), we notice that the distribution is polarized towards the maximum value—first quartile, median and third quartile equal to 3—with 14 (82% of the total) of the videos where the Stack Overflow discussions were considered as complementary. Results indicate that, while respondents only considered the retrieved discussions fairly relevant
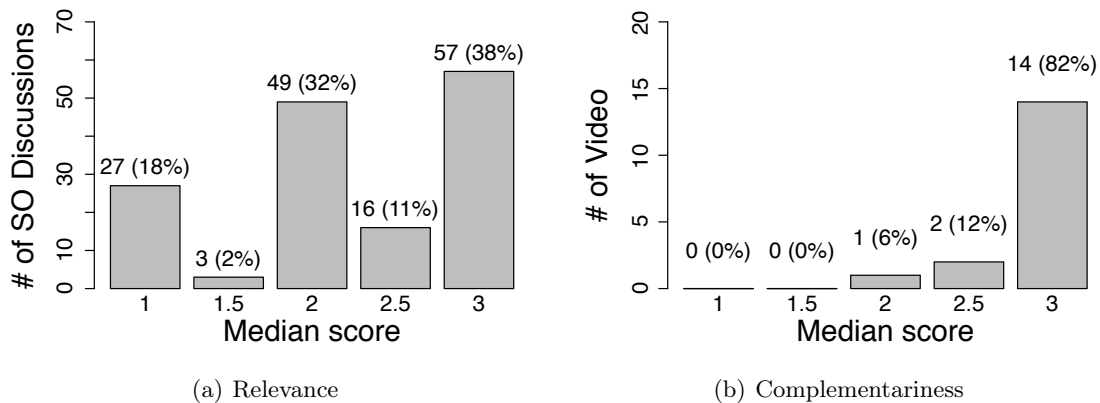
(a) Relevance   (b) Complementariness

Figure 8.11. RQ_4: Relevance of Stack Overflow discussions to video fragments, and complementariness to videos.

to the fragments from where the queries were generated, they almost totally agreed about the complementarity of the provided information. We believe that video tutorials have a different purpose than Stack Overflow discussions. The former have an introductory, step-by-step guide to a given problem, the latter discuss a specific problem/answering a specific questions.

RQ_5: To what extent is CodeTube able to return results relevant to a textual query?

In the last part of the survey, we asked participants to evaluate the top-three results that CODE-TUBE and YouTube retrieved for a set of 10 queries. Each participant evaluated the relevance of a result with respect to the query by following a three-level Likert scale[Opp92], *i.e.,* "very related", "somewhat related", and "not related". We use the Normalized Cumulative Discounted Gain (NDCG)[MRS08] to aggregate the results.

Similarly to what done for the other research questions, queries with less than 3 replies are ignored. The NDCG is thus calculated on a set of 8 queries out of the initial 10. We obtained $NDCG_{CT}(Q,3) = 0.67$ and $NDCG_{YT}(Q,3) = 0.63$ for CODETUBE and YouTube, respectively. Even if CODETUBE seems to perform slightly better than YouTube, a statistical analysis of the $NDCG_{YT}$ and $NDCG_{CT}$ distributions, performed using the Wilcoxon paired test, did not show the presence of a statistically significant difference (*p*-value=0.49). Even though the data collected is not enough to draft any statistically significant conclusion, there are some considerations to make. First, when extracting the top-three results from YouTube we removed all the retrieved videos that are not included in the CODETUBE dataset. This makes the comparison unfair for our approach. Second, YouTube recommends entire videos, while CODETUBE recommends specific fragments. Thus, our approach is potentially more focused even if both the fragment and the whole video recommended by YouTube are equally relevant.

CodeTube: Strengths and weaknesses

In the last part of the questionnaire we asked participants to freely comment about CODETUBE. The participants have in general a positive impression of CODETUBE. The UI has been appreciated by some of the participants. For example, one of the respondents reported *"CodeTube looks very useful, added to the bookmarks!"*, while another wrote *"excellent work, [...] the idea behind*

CodeTube is brilliant". The extraction of fragments from video tutorials has been appreciated and considered *"very useful for developers who are already knowledgable about the topic, they can save a lot of time"*.

The possibility of having complementary sources of information, *e.g.,* Stack Overflow has been appreciated by some participants. One of them reported that *"the concept is amazing, and has a lot of possibility of improvement, given the huge amount of different sources of data available"*, while other participants asked for additional features to improve this functionality. One participant asked for *"the possibility to search for SO discussions directly below the video"*, while another wondered that *"it would be nice if the tool can provide a summary/description that describes the context"*.

8.5   Study III: Extrinsic evaluation

A successful technological transfer is the main target objective for each prototype tool. Thus, the *goal* of this second study is to extrinsically investigate CODETUBE's industrial applicability. Specifically, the research question we aim to answer with this second evaluation is:

RQ$_6$: *Would* CODETUBE *be useful for practitioners?*

The *context* of the study is represented by three leading developers—all with more than five years of experience in app development—of three Italian software companies, namely Next, IdeaSoftware, and Genialapps.

8.5.1   Study design and procedure

We conducted semi-structured interviews to get quantitative and qualitative feedback on CODE-TUBE. Each interview lasted two hours. During the interview we let developers explore CODE-TUBE for about 90 minutes, searching for video tutorials on specific technology or to fix problems. Each interview was based on the think-aloud strategy.

We also explicitly asked the following questions: (1) Do you use video tutorials during development tasks? (2) Would the extraction of shorter fragments make you more productive? (3) Is the multi-source nature of CODETUBE useful? (4) Are you willing to use CODETUBE in your company? Participants answered each question using a 4-point Likert scale: absolutely no, no, yes, absolutely yes. The interviews were conducted by one of the authors, who annotated the answers as well as additional insights about the strengths and weaknesses of CODETUBE that emerged during the interviews.

8.5.2   Study results

This section discusses the outcomes of the semi-structured interviews we conducted.

Nicola Noviello, Project Manager @ Next

Nicola positively answered to our first three questions (*i.e.,* "absolutely yes"). Nicola declared to use video tutorials daily; "*they are particularly useful for senior and junior developers for both learning a new technology or finding the solution to a given problem. I see very often my developers on specialized YouTube channels searching for and watching video tutorials"*.

Nicola also appreciated the multi-source nature of CODETUBE; "*the video tutorial provides the general idea on the technology, while Stack Overflow discussions are particularly useful to manage alternative usage scenarios and specific issues*".

Regarding the extraction of fragments, Nicola commented that "*I usually discard video tutorials that are too long, because when I try to scroll/fast forward it to manually locate segments of interest, I am generally not able to find what I need. I strongly believe that the relevant segment is there but randomly scrolling a video tutorial is not worthwhile! I prefer to look for more focused video tutorials. Also, the possibility to filter video fragments on the basis of their category is a fantastic feature!*". Nicola then confirmed that the availability of shorter fragments would make him much more productive.

Nicola answered "yes" to the question related to the usefulness of CODETUBE; "*I did not answer absolutely yes because of the limited number of indexed tutorials. However, I strongly believe that the tool has an enormous potential*". Nicola declared that he will present the tool to a newcomer trainee to quantify to what extent the tool is useful for developers that have a little knowledge on the Android world; "*I usually suggest to trainees to look for and watch video tutorials but very often they are not able to find the right information. I would like to see whether CODETUBE is able to mitigate such a problem*".

Luciano Cutone, Project Manager @ IdeaSoftware

Luciano positively answered to our first three questions; "*I love video tutorials but several times they are too long and I do not have enough time to watch whole videos. Thus, I have to scroll the video hoping to identify relevant segments. This takes time and makes video tutorials less effective. With CODETUBE life will be easier!*".

Luciano particularly appreciated also the possibility to filter video fragments on the basis of their category. He also suggested an interesting new features:*It could be nice to give the possibility to the user to change the classification of the video tutorials. In this way it is possible to correct possibly misclassification and improve the classification accuracy of the tool. Of course, a moderator is required to accept the proposed change.*"

When exploiting different sources of information, Luciano works differently from Nicola; "*I like the idea of having video tutorials together with Stack Overflow discussions. However, the main source of information for me is Stack Overflow, while video tutorials should be used to fix problems; if I need to apply a new technology, I would like to start from Stack Overflow since there I can find snippets of code that I can copy and paste into my application. Then, if something goes wrong, I try to find a video tutorial to fix the problem*".

Luciano also suggested a nice improvement; "*Besides the integration of video tutorials with discussions on forums, I suggest to add another source of information, namely sample projects. Specifically, on GitHub there are several sample projects that explain how to apply specific technologies. Having them together with video tutorials and Stack Overflow discussions would be fantastic.*" Another suggestion was the addition of a voting mechanism to provide information on the usefulness and the effectiveness of a specific (fragment of a) video tutorial.

Luciano answered "absolutely yes" to our last question (*i.e.*, the one related to the usefulness of CODETUBE); "*I just added CODETUBE to my bookmarks. This is the tool I wanted. I spent several hours of the day and of the night on YouTube and Stack Overflow to fix problems or learn new things. This is part of my job, unfortunately. With CODETUBE I am sure that I will find relevant information quickly. I can finally go back to sleep during the night!*". The day after the interview, we got a text message from Luciano: "*I have just used CODETUBE this morning. I was looking for something related to Android WebSocket. I found all I needed. Awesome!*".

Giuseppe Socci, Project Manager @ Genialapps

Giuseppe answered "absolutely yes" to our first question, stating that in his opinion "*Video tutorials are a crucial source of information for learning a new technology*". Instead, he answered "no" to our second research question related to the extraction of fragments; "*I am not 100% sure that extracting shorter fragments makes you more productive. It depends on the scenario where the video tutorial is used. To me, video tutorials should be used to learn a new technology. In this case I should watch the whole video. However, there could be cases where you just need to fix a problem or have some clarifications on a specific part of the technology. In this case watching fragments instead of whole videos could be worthwhile*". In this particularly scenario, Giuseppe found the possibility to filter video fragments on the basis of their category a killer functionality: "*the filtering based on the category of the video fragments can really help in improving productivity.*"

Giuseppe also suggested a way to make the tool more usable based on his way of interpreting video tutorials; "*the search of a video tutorial should be scenario-sensitive. Before searching, the user should specify why she is searching for a video tutorial. The first option could be 'I have a problem'. In this case, the search is based on fragments. The second option could be 'I want to learn'. Here, whole videos should be retrieved*".

As well as the other two developers, Giuseppe liked the integration of video tutorials with forum discussion (he answered "absolutely yes" to our third research question). Consistently with findings of Study I (Section 8.4.2), he highlighted the need for manually refining queries when retrieving Stack Overflow discussions: "*all the visualized Stack Overflow discussions are related to a specific video tutorial. However, Stack Overflow discussions should be useful to resolve a problem I encountered when applying the technology explained in the video tutorial. Thus, it might be useful to filter the retrieved discussion by a specific query (e.g., the type of error I got)*".

Finally, Giuseppe answered "yes" to our final question; "*I think that the tool is nice. You are trying to solve an important and challenging problem, that is merging accurately different sources of information in order to make them more productive*". Giuseppe also gave a suggestion on how to improve the visualization of the relevant fragments; "*After submitting a query, CODETUBE provides the list of relevant video fragments. However, it is quite difficult from the title of the video and the cover image to identify the most relevant one. I strongly suggest to show for each video the relevant textual part of the video content, similar to the part of the text in a Web page content visualized by Web search engines. The same approach could be used also to make the navigation of the fragments of a specific video easier*".

8.6   Threats to Validity

Construct Validity

Threats to *construct validity* are mainly related to the measurements performed in our studies. In Study I this is mainly due to subjectiveness in the construction of the labeled fragment dataset used in our study, since each video has been fragmented and tagged by one expert only. However, during the second phase (open coding) two authors, before starting the open coding activity, performed a sanity check of the obtained fragments and tags. Finally, subjectiveness in the open coding was mitigated by employing a multiple-coder strategy, for which there has also been a very strong inter-rater agreement even since the first coding phase.

In Study II, instead of using proxy measures, we preferred to let developers evaluate video fragments and their related Stack Overflow discussions. Subjectiveness of such an evaluation was mitigated by involving multiple evaluators for each video, although, as explained in Section 8.4.1,

we favored the number of videos over the number of responses per fragment. Although a four or five-level Likert scale [Opp92] could have provided a more accurate evaluation, we preferred a simpler three-level scale to facilitate the task to the respondents, which was already long, due to the need for watching the videos before answering the questions.

Internal Validity

Threats to *internal validity* concern factors internal to our studies that could have influenced our results. One possible problem is that the evaluation in Study II could have been influenced by the knowledge of respondents about the topic. We mitigated this threat by discarding responses of participants not having any knowledge about Android. In addition, the evaluation is mainly related to cohesiveness, self-containment and relevance of video fragments, and relevance and complementariness of Stack Overflow discussions, rather than to how they would be helpful for the respondents. Another possible bias is represented by the videos used in the survey, that have been randomly sampled by considering 7 minutes as maximum video duration, and three as maximum number of fragments for each video and query results. These limitations have been introduced to restrict the survey duration to a reasonable time.

Concerning the machine learning classifier, for the preprocessing phase, we have mitigated possible multicollinearity problems by using a feature selection approach. We have used SMOTE to deal with unbalanced data. Finally, while we have tried different machine learning techniques and chosen the one (Random Forest) producing the best results, it is possible that we did not consider techniques (or parameter settings for a technique) producing even better results than what we achieved. As explained in Section 8.2 we adopted a simple and possibly sub-optimal LDA calibration when extracting semantic features. This is in line with what done when using LDA in classification approaches [GTGZ14].

External Validity

Threats to *external validity* concern the generalizability of our findings. Our studies are limited to Java video tutorial only. We do not expect large differences in the structure of a video tutorial for a different programming language. It is possible that results might not generalize. Although our approach captures a wide range of information characterizing video tutorials from different perspectives, it is possible that additional information might be required when dealing with tutorials about pieces of technology not considered in our dataset. Finally, the validity of the third study is limited to the three very specific mobile app development contexts considered.

8.7   Conclusion

Software development video tutorials are on the rise. They are a modern medium to disseminate in-depth technical knowledge, and if we were to make an informed guess, they represent the next frontier in software documentation. However, the intrinsic nature of audio-visual content poses a number of challenges. First, it is difficult, if not impossible, to search videos based on their contents. This is a prime requisite to make the information contained in the video tutorials more accessible. Second, it is non-trivial to understand whether a video contains the information one is looking for, short of watching the whole video. Third, video tutorials are somewhat remote from the working context of developers, specifically their development environment.

We presented CodeTube, a novel approach to extract and classify relevant fragments from software development video tutorials. CodeTube mixes several existing approaches and tech-

nologies like OCR and island parsing to analyze the complex unstructured contents of the video tutorials. Our approach extracts video fragments by merging the code information located and extracted within video frames, together with the speech information provided by audio transcripts. Also, it automatically classify the "type" of video fragment (*e.g.,* theoretical, implementation) and complements the video fragments with relevant Stack Overflow discussions. We conducted three studies to evaluate CODETUBE, showing its ability to identify and correctly classify meaningful code fragments. Also, we investigated the perception of our approach in industry environments by interviewing three leading developers, receiving useful insights on the strengths and potential extensions of our current work. To our knowledge, CODETUBE is the first, and freely available[19] approach to perform video fragment analysis for software development.

Reflections

This chapter presented, to the best of our knowledge, the first classification and fragmentation approach for video fragments for software engineering. By devising and implementing CODE-TUBE, we showed how a H-AST model can be beneficial even for non-textual artifacts, once the contents are extracted.

In the next part we leverage the approaches and results of previous chapters to move a further step towards a holistic interpretation of the information provided by development artifacts. We will take advantage of the meta-information model to devise novel analyses on the contents, unveil latent semantic links among different parts of an artifact, as well as different artifacts. The analyses that will be developed on top of the meta-information model will provide the foundations of the first H-RSSE, an application capable of understanding and guiding the developer through all the information provided and retrieved when navigating heterogeneous resources.

---

[19]`http://codetube.inf.usi.ch`

Part IV

Holistic RSSEs

9

Summarizing Complex Development Artifacts by Mining Heterogeneous Data

When searching for information to complete their tasks, developers have to deal with documents whose size and complexity might be not negligible. The information needed by the developer may reside in a single part of the artifact, that developers have to seek within the contents. The irrelevant parts become noise hindering the perusal, and causing an information overflow on the developer.

A H-RSSE should summarize the information to prevent or reduce the information overflow on the developer. Several approaches tackle this problem and proposed automated summaries of development emails [RMM14a, BLM12], bug reports [LMC12, MCSD12], and source code [RMM+14b, HAMM10, MAS+13, GBR15].

However, all of these approaches treat every artifact as a purely textual artifact, or limit their summarization technique to a single type of artifact. Relying on the textual and reductive interpretation of the data of the contents hinders the analysis of the artifacts. The information is rather heterogeneous and includes code, text, and many other types of information whose semantic links cannot be uncovered and exploited from a textual point of view to build a summary.

In this chapter, we propose a novel technique to summarize Stack Overflow discussions by dealing with heterogeneous information. We revise LexRank [ER04], a summarization approach based on PageRank [BP98], by devising HoliRank, a customized LexRank using a *holistic* similarity function for heterogeneous entities like code samples and XML configuration files. The preliminary results we obtained suggest that a holistic approach on the heterogeneity of the information could lead to better summarization results.

Structure of the Chapter

In Section 9.1 we present an overview of PageRank and how it can be tailored to achieve LexRank. In Section 9.2 we present HoliRank, an extension of LexRank leveraging our meta-information model to establish a semantic between two elements by using a *holistic* similarity function. Section 9.3 presents a preliminary comparison of our approach with LexRank, and Section 9.4 concludes the chapter.

9.1 LexRank

Erkan *et al.* developed LexRank [ER04], an unsupervised summarization algorithm based on PageRank [BP98], a well known algorithm designed by Brin and Page. Understanding LexRank requires background knowledge of the PageRank algorithm. We briefly describe it from a high-level perspective to give a general idea.

9.1.1 PageRank

To compute the relevance of a page within a network of pages, the PageRank algorithm models the behavior of a *"random surfer"*: a user who randomly surfs the web and *"keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page"* [BP98]. The random surfer model is defined by the following equation:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{9.1}$$

where d is a dumping factor generally set to 0.85 [BP98], $M(p_i)$ is the set of pages that link to p_i, $L(p_j)$ is the number of outgoing links from p_i, and N is the total number of pages in the network and serves as normalization factor.

According to Equation 9.1, the *random surfer* can navigate to a neighbor page with probability d, or to jump to any other page with a probability of $1-d$. This user behavior is combined with the distribution of the rank of a page across the network by repeatedly equally distributing it to the neighbors of a page (*i.e.,* $\frac{PR(p_j)}{L(p_j)}$ in Equation 9.1).

The computation of PageRank can be performed in a iterative way, by starting from an initial guess of the rank of each page, and iteratively refining it until it reaches a steady state [BP98]. The output returned by the PageRank algorithm is a (normalized) probability distribution PR where each $PR(p_i)$ represents the probability that the *random surfer* visits a page i. The probability associated to a page by this algorithm represents the centrality of this page in the network.
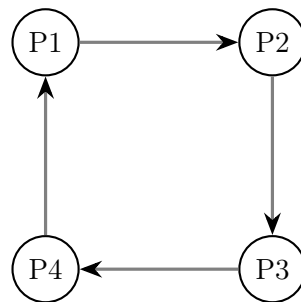

Figure 9.1. Example of equally Distributed Network

For example, Figure 9.1 shows an equally distributed network, where each page has a link to only another page in the network. The centrality of the pages in this network is equally distributed, and PageRank scores 0.25 to each page.

Figure 9.2 shows another example where the network has a strong hierarchical organization. It is possible to divide the nodes in three layers: The top layer ($L1$) is composed by $P1$ only, the second layer ($L2$) is composed by $P2$, $P3$, and $P4$, and the last layer ($L3$) is composed by $P5$,
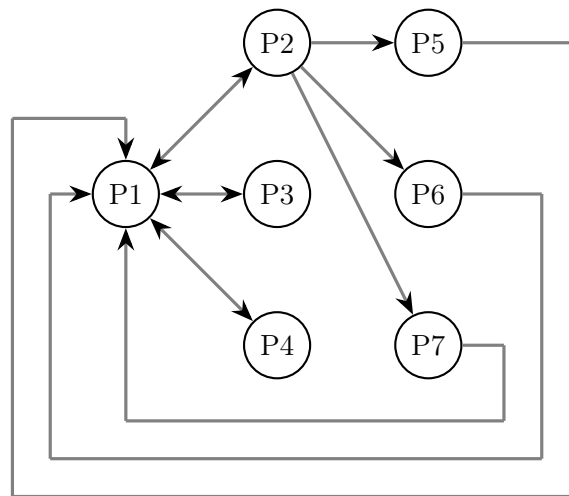
Figure 9.2. Example of hierarchical network.

$P6$, and $P7$. The flow of the links in the network is clearly visible: $L1$ links $L2$ which in turn backlinks to $L1$, $L2$ links $L3$ without backlinks, and $L3$ links to $L1$, thus providing a indirect backlink to $L1$ for $L2$.

This hierarchical structure is also reflected in the values of the PAGERANK since it assigns 0.43, 0.14 and 0.05 to all the nodes in to $L1$ ($P1$), $L2$, and $L3$, respectively.

9.1.2   From PageRank to LexRank

The main intuition of Erkan *et al.* [ER04] behind LEXRANK is to consider a set of documents as a network of sentences, use PAGERANK to evaluate the centrality of the sentences within the network, and select the top-k central sentences to build an extractive summary. The original application of LEXRANK concerns multi-document summarization, but it can be also used to build a summary of a single document.

Differently from web pages, sentences have no explicit link among each other, thus the construction of the input graph for PAGERANK requires a different approach. Instead of a graph built by analyzing the links of webpages, LEXRANK builds a similarity graph where the nodes represent the sentences, and the edges the similarity. An edge between two sentences in the graph exists if and only if the textual similarity surpasses a threshold (*e.g.,* 0.1, 0.2). The textual similarity employed in the LexRank algorithm is *tf-idf* [MRS08]. Since *tf-idf* is a symmetric function (*i.e.,* $f(a, b) = f(b, a)$), the created edges are not directed, thus forcing the input graph to be undirected as well.

Being the distribution estimated by PAGERANK on undirected and directed graph close but not identical [PF08], the computation of the original PAGERANK can be performed on the similarity graph, allowing to estimate the lexical PAGERANK, or LEXRANK [ER04].

9.2   HoliRank: Holistic PageRank

LEXRANK could have been used "as is" to generate a summary of a software artifact. As widely discussed in Chapter 6 a software artifact is not a pure textual entity, but it contains heterogeneous types of information. For example, if we consider a Stack Overflow discussion, and we

limit the heterogeneous elements to text and code, leveraging measures like pure textual similarity based on vector space model (*i.e.,* tf-idf), or topic-based similarity, might reveal themselves as unappropriated solutions.

Even if we consider simple term extraction or code labeling approaches as the minimal extractive summary processes [HAMM10], where the heterogeneity of the information can be reduced to code and comments, a pure vector space model does not perform well. According to De Lucia *et al.* [LPO+12], a naïve heuristic that extracts names from class definitions outperforms vector space model approaches (*i.e.,* tf-idf) in labeling source code. They suggest how *"ad-hoc heuristics can be used to better approximate the mental model used by developers when identifying class keywords"*.

We believe that textual similarity as the only dimension to evaluate the "distance" between two heterogeneous types of information is constraining and reductive. For example, textual similarity hinders the whole information provided by a code sample, and it is not practical to devise a concept of similarity between homogeneous code elements. Also, textual similarity cannot be applied to any non-textual elements, preventing us from including information coming from images or video, as well as information derived from third party elements like users or developers. Textual similarity should be considered as complementary part of the information within a concept of multidimensional information, where every type of information unit contained in an artifact contributes in each dimension in a different way.

In this section we draft the basis of HOLIRANK, customized version of LEXRANK, that uses a *holistic* function to evaluate the similarity between two nodes in the graph.

9.2.1 Meta-Information

Since we want to analyze the information from a *holistic* point of view, we need to go beyond the limitation imposed by the text. To this aim we reuse the STORMED dataset (see Chapter 7). Each discussion in STORMED is modeled as a set of information units that preserves the human tagging (*i.e.,* Code Tagged Unit, and Natural Language Tagged Unit). By reusing the information units, we can also take advantage of the meta-information model, and the H-AST model defined and discussed in Chapter 6, since each information unit carries one or more meta-information types concerning a specific aspect of its contents. We consider the following subset of meta-information to analyze the contents of each information unit encountered in a discussion:

Types: It represents the set of Java types mentioned in a information unit. We consider fully qualified types (reference types), simple names matching Java convention for classes (*i.e.,* begin with a capital letter), and primitive types (*e.g.,* int, double). This meta-information applies to all the information units (*e.g.,* types mentioned in natural language and extracted with the island parser).

Variable Names: All the AST nodes matching a variable name are extracted and stored in this meta-information node. This applies to code samples and textual information units.

Invocation Names : All the AST nodes matching a method invocation are extracted and the name of the invoked method is stored in the meta-information node. We discard arguments passed to the method. This meta-information applies to both code samples and textual information units.

Natural Language : We also complement the meta-information with pure textual information. For each type of information unit we can generate a *tf-idf* vector that will be used later on in the calculation of the similarity.

9.2.2 A Holistic Similarity Function

The final step in devising HOLIRANK is to devise a similarity function that takes two information units and returns a similarity value, which ranges between 0 and 1. Each type of information unit can carry an arbitrary number of meta-information. To explain how we construct the similarity of two information units from their meta-information, and thus how the similarity function works, we go through an example scenario. Consider two information units U_x and U_y of different type. Since we can compare only shared meta-information, let $T_{x,y}$ be the set of shared types of meta-information between the units, and let $M(U, t)$ be the meta-information of type t for the information unit U. We define the similarity vector V_{xy} as:

$$V_{x,y} = \langle v_0, \dots, v_{|T_{x,y}|} \rangle$$
$$\text{with } v_i = M(U_x, t_i) \sim M(U_y, t_i) \text{ and } t_i \in T_{x,y} \tag{9.2}$$

Each element v_i of the vector V represents the similarity value between two homogeneous meta-information units, and ranges in the interval $[0, 1]$. Once we have computed all the elements of V, we calculate the general similarity between two information units U_x and U_y as the norm of the vector:

$$f_{sim}(U_x, U_y) = \frac{||V_{x,y}||}{\sqrt{dim(V_{x,y})}} \tag{9.3}$$

Since we want f_{sim} to provide a value in the range $[0, 1]$, we divide the norm of the vector by the maximum possible value of the norm. Being 1 the maximum value of each v_i, the maximum value of the norm is the square root of the arity of $V_{x,y}$. We can use this similarity function to devise the edges of the graph and use the PAGERANK algorithm to compute the centrality of an information unit inside the network of information units of an artifacts.

9.2.3 Summary Generation

Once the HOLIRANK is computed, each information node receives a centrality value. We select the top n units, according to the percentage of the summary we want to show. In essence the user can decide how concise the summary will be. However, we keep one single constraint on the original structure of the Stack Overflow discussion: There must always be one information unit from the question and one information unit from one of the answers. The unit can either be extracted from the body or from one of the related comments.

The generated summary is interactive and allows the developer to incrementally disclose information on demand. Figure 9.3 shows the user interface of the summarizer. By using the slider in the top right corner, the developer chooses the percentage of the original discussion that she wants to see.

9.2.4 A Practical Example

Figure 9.3 depicts a Stack Overflow discussion[1] where the user is asking about the possibility of overriding the Spring XML configuration in Java. The user prepares a detailed report of the problem including the class and the XML configuration he is writing, and the solutions he already tried out. The question by itself is, due to the details, already verbose, contains

---

[1]`http://stackoverflow.com/questions/10534893`

Figure 9.3. The Stack Overflow Summarizer Interface with full discussion.

comments revealing the evolution of the question, where another user ask to add a code sample to the question. Similarly, in the answer, there are comments where the author thanks the replier.

All of these parts are indeed needed by the community to work out the problem concerning the question. However, the usefulness of such parts decades when a "external" user start reading the discussion. Begin de-contextualized, all the aforementioned parts might become superfluous, boiling down to additional noise hindering the comprehension.

By using the summarizer, the developer can automatically reduce the noise of the discussion and reach the core of the information. Figure 9.4 shows the same discussion with only 40% of

Figure 9.4. The Stack Overflow Summarizer Interface with 40% of the discussion.

Figure 9.5. The Stack Overflow Summarizer Interface with 10% of the discussion.

the contents selected. Our approach analyzes the relationships among the information units, by leveraging the meta-information model, and reduces the discussion by removing most of the comments, the code sample, the XML configuration from both the question and the answer.

Even though the question is slimmer, some parts are still noisy. For example, there are still parts concerning the edits (*e.g.,"EDIT: Updated with XML."*) in the question and in its comments, which are now useless. The developer can further ask the summarizer to reduce the question by sliding to the left.

Figure 9.5 shows the question to the very minimum (10%). In this case the summarizer select the two most prominent units from the question, and the answer respectively. The overall discussion is thus reduced to a Q&A where the user explicitly asks *"Is it possible to override imported resources using Spring annotation configuration"*, replicating the title of the discussion, and the answer is reduced to the only sentence *"You cannot override spring xml configuration using annotation"*. If a developer reads a summary like this, she would immediately understand the point without having to read everything.

Figure 9.6. Example of Stack Overflow discussion proposed to users.

9.3 Preliminary Evaluation

We present a preliminary evaluation of our approach, starting from its setup. We involved nine people (6 Master students and 3 PhD students) to annotate information units on Stack Overflow discussions through a web application developed by us, illustrated in Figure 9.6.

The application presents the discussions with a random order to each user, and shows the contents of a discussion by separating each single information unit from the other. Each subject annotated the units of 9 different discussions. Every user gave a rating to each information unit by providing the number of stars on a Likert scale between one and five. We asked people to give a rating according to the prominence of the unit in the discussion.

9.3.1 Evaluation Approach

According to the current state of the art, there is no standardized way of evaluating artifacts summaries [BLM12]. Moreover, nobody tackled the evaluation of summaries containing heterogeneous information units. We devised our own approach to evaluate our summaries. We tried to simulate the generation of a "golden standard" summary. Given a Stack Overflow discussion, we calculate the average rating received for each information unit, and we sort the units in descending way. Then, we fix a percentage value that represents the subset of information unit we want to show in the summary. This subset represents the golden summary for a given percentage.

Table 9.1. Precision on human annotated discussions.

Our Approach

Size	D1	D2	D3	D4	D5	D6	D7	D8	D9
5%	0%	**50%**	0%	**100%**	100%	0%	50%	0%	33%
10%	33%	**25%**	50%	**67%**	50%	**67%**	**50%**	0%	**33%**
15%	60%	43%	33%	**60%**	75%	**40%**	33%	20%	**44%**
25%	56%	55%	30%	33%	43%	25%	**50%**	**38%**	**53%**
35%	62%	75%	50%	50%	30%	45%	**57%**	**42%**	52%
50%	58%	57%	57%	56%	43%	65%	**70%**	47%	67%

Original LexRank Algorithm

Size	D1	D2	D3	D4	D5	D6	D7	D8	D9
5%	0%	0%	0%	0%	100%	0%	50%	0%	33%
10%	33%	0%	50%	33%	50%	33%	25%	0%	17%
15%	60%	43%	33%	40%	75%	20%	17%	20%	33%
25%	**67%**	55%	30%	33%	43%	**38%**	40%	25%	47%
35%	**69%**	75%	50%	50%	**40%**	45%	50%	33%	52%
50%	**63%**	**61%**	57%	56%	43%	65%	65%	47%	**70%**

Once we have constructed the golden summary of a discussion, we generate a summary with our own approach. To evaluate the generated summary we calculate the precision, that is, the percentage of units selected by our approach that matches the units in the golden summary.

9.3.2 Preliminary Results

Table 9.1 shows the preliminary results we obtained. We created six golden summaries representing 5%, 10%, 15%, 25%, 35%, and 50% of the information units contained in a Stack Overflow discussion. To have a reference value, we applied the same evaluation approach on the original LEXRANK algorithm, that is, every information unit treated as pure textual information. Best results are reported in bold.

At a first observation, we can distinguish four different scenarios of the two approaches. The first scenario concerns discussions D3 and D5, where results show no difference in terms of performance. The second scenario concerns discussion D1, where the original LEXRANK algorithm performs better than our approach. On the opposite side, we have the third scenario, where for discussion D4, D7, and D8 our approach outperforms the pure textual based approach. In the fourth scenario we have discussions D2, D6, and D9, where each approach performs better than the others depending on the size of the summary. These preliminary results suggest that taking into account the heterogeneity of the information contained in a software artifact could lead to better summarization results, and it is worth being explored in depth.

In general, the results show that our approach generally either outperforms the classic LEXRANK algorithm, and specifically it does so on the shorter summaries. This is what the user in the end finds more desirable, since shorter summaries reduce the information that needs to be taken in by a person. This is also especially true in the case of heterogeneous artifacts, where a person is otherwise forced to comprehend the various distinct pieces of information of different nature.

9.4   Conclusions

Summarizing complex software artifacts is a non-trivial task due to their heterogeneous and multidimensional nature. Different fragments of information (*e.g.,* code, text, xml) co-exist in the same artifacts and contribute differently to the overall knowledge contained in them. Current approaches in summarization do not take this fundamental fact into account, and reductively treat artifacts as if they were purely textual.

We revisited a textual summarization approach like LEXRANK, and we modified it, drafting the basis of HOLIRANK, a summarization approach aimed at considering integrate different aspects of the heterogeneous information contained in an artifact. We obtained promising results in our preliminary evaluation, and we discussed how results suggest that a holistic point of view on the heterogeneous information contained in software artifacts is worth being explored to improve the current state-of-the-art approaches.

Reflections

The chapter presented a first example of *holistic* analysis on the information available within a development artifact like Stack Overflow. The similarity graph built by HOLIRANK captures multi-dimensional aspects of the information shared by two entities, by devising a customized analysis on top of the meta-information system devised in Chapter 7.

HOLIRANK is an example of how an additional layer of abstraction on the information is convenient to manipulate and aggregate information. HOLIRANK is more than a summarization approach, and estimates the centrality of a generic entity within a graph by analyzing the information from an *holistic* point of view, thus abstracting the nature of the entity themselves. In other words, entities (*e.g.,* nodes in the graph) might be entire parts of artifacts, entire artifacts, or collections of artifacts. The abstraction provided by a meta-information model aggregates such entities in one single sink of information.

In the next chapter we reuse HOLIRANK for other purposes than summarization. We change the granularity from the *information unit* level to the entire artifact level, and leverage the analysis performed by HOLIRANK to better understand and exploit the contextual information perused by a developer during a programming task.

10

Supporting Software Developers with a Holistic Recommender System

Developers frequently search the web for the information fragments needed to complete a task [SSE15]. Following an iterative approach [Hol09], they inspect resources until they reach a satisfactory level of knowledge to solve a given task. This process can be described as a foraging loop to seek, understand, and relate information [PC05]. It can also be seen as a treasure hunt, where the map is progressively unveiled as hints are found along the way. Getting new hints to proceed towards the treasure requires one to search in the current zone of the map, facing riddles and tricks to get new pieces of the map, and eventually hunt down the treasure.

Current RSSEs shortcut this process by pointing out a "candidate treasure" (*e.g.,* a Stack Overflow discussion for a given task) using only some small pieces of the map (*e.g.,* by only knowing what the developer is doing in the IDE). However, all pieces of the map are essential to proceed. The unveiled pieces of the map are the developers' *knowledge context*, continuously refined as they peruse new resources or modify existing code. In our vision, a recommender system should provide continuous counseling to developers, guiding their information seeking process, taking into account what they are working on and what they already perused. The recommender should suggest to developers, in a timely fashion, pertinent artifacts given the current context: the developers' knowledge (*i.e.,* the already unveiled pieces of the map).

We propose LIBRA, a *holistic* recommender system that provides developers with real-time support for information navigation in the web browser. LIBRA monitors the developers' activity both in the web browser and in the IDE to track web search results, perused pages, and code written and modified by the developer. LIBRA models the knowledge context of the developer by considering all these resources, and constructs a *holistic* meta-information model of their contents. LIBRA's analysis does not consider the contents of resources as mere text, but takes into account their heterogeneous composition, including code fragments and exchange formats like XML and JSON. By holistically analyzing the contents of the knowledge context, LIBRA assists developers in selecting pertinent results from a web search by considering the *prominence* of a given resource, and the *complementarity* of a result with the gathered knowledge context. We evaluated LIBRA with two different studies to assess its usefulness during development activities, and its applicability in industrial contexts. Both studies showed that a holistic analysis of the developers' information context can offer comprehensive and contextualized support to information navigation and retrieval during development.

Structure of the Chapter

In Section 10.1 we describe LIBRA, its architecture and its user interface. Section 10.2 details the *holistic* approach implemented in LIBRA's core. In Section 10.3 we describe the controlled experiment to evaluate LIBRA's usefulness during development activities. Section 10.4 reports interviews with five industrial practitioners discussing LIBRA's applicability in industrial practice. Section 10.5 concludes the chapter.

10.1 Libra

LIBRA is a recommender system aimed at extending and integrating the two main modern software development tools—the IDE and the web browser—to support information seeking.

10.1.1 User Interface
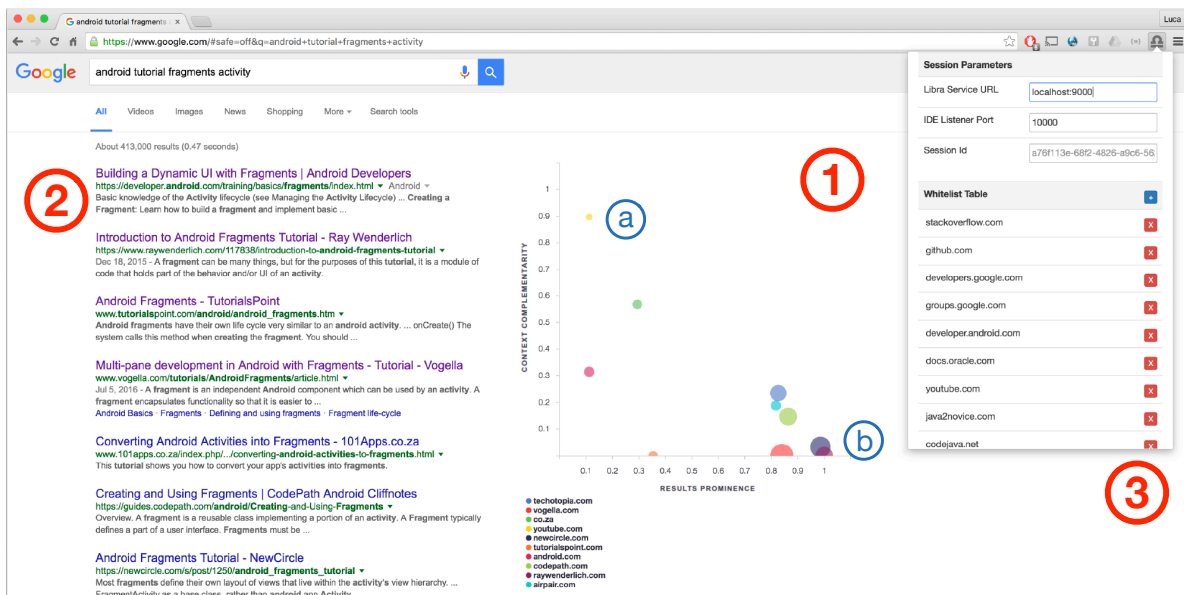
Figure 10.1. The Libra user interface.

Figure 10.1 shows the LIBRA user interface as it appears in the web browser. It includes four components providing features to navigate the information space of the search engine[1].

Whenever a developer writes a query in a browser, a two-axes bubble chart (1) appears on the right side of the web page. Every bubble represents an entry in the results list (2) on the left side. Hovering on a bubble highlights the corresponding entry in the results list, and fades out the others. If a developer hovers over a search result, LIBRA highlights the corresponding bubble, and fades out the others. The developer can access the URL of a search result by clicking on the corresponding entry in the results list or on the related bubble in the LIBRA chart.

The URL is then opened in a new tab, while the chart gets updated with the new context information, and the visited URL becomes part of the developer's context, including all the

---

[1]LIBRA uses Google in its current implementation.

recently navigated resources and the code she recently wrote in the IDE. The chart provides additional support to navigate the information space by visualizing the following information:

Bubble Color: Resources are grouped by their domain and assigned a specific color. The bottom part of the chart contains an interactive legend reporting all domains found in the result set. Developers can click on a domain to highlight its results, fading out the others.

Context Complementarity: The y-axis represents the complementarity of the information provided by a search result with respect to the current context. The higher the position of a resource, the higher its complementarity with the developer's context, who can thus decide between broadening the context or sticking with resources similar to the ones already perused. For example, in Figure 10.1.1 the resource(s) browsed on `youtube.com` (a) has high complementarity (but low prominence).

Result Prominence: The x-axis allows the developer to discriminate among the results returned by the search engine. The more a result is on the right side of the chart, the higher its prominence within the result set[2]. Developers can use this axis to avoid out of scope results, or results whose information is a subset of more prominent resources. For example, the resource(s) browsed on `raywenderlich.com` (b) has high prominence, (but low complementarity).

Bubble Diameter: It represents the quantity of information provided by a resource with respect to the whole result set. Ranging between 10 and 25 pixels, the diameter is normalized on the maximum information content value. For example, the large red (semi)circle on the x-axis refers to resources on `vogella.com`, providing higher quantity of information than other resources shown in small circles, *e.g.,* the yellow `youtube.com` circle.

The developer can take advantage of the features described above to select resources that best suit the next step in the information seeking process. LIBRA also provides an options panel (3) where the developer can access basic information about the state of the application, and manage a white list of domains that can be freely tracked (*i.e.,* that can be part of the developer's context). A demo of LIBRA is publicly available[3].

10.1.2 Architecture

Figure 10.2 shows the architecture of LIBRA and uses two types of arrows to denote two different phases performed by LIBRA: Solid arrows represent the tracking events, while the dashed arrows represent the events caused by the interaction of the developer with LIBRA.

LIBRA is composed of three main components: (1) a plugin for the IntelliJ IDEA integrated development environment that takes care of tracking the modified and accessed source code, (2) a Google Chrome extension that tracks the web pages perused by the developer and augments the Google search result web page with the LIBRA user interface, and (3) a back-end service hosting LIBRA's analyzer as well as its data. The use of the Google search engine, IntelliJ IDEA, and Google Chrome are implementation choices adopted for our convenience.

---

[2]Our exact definition of prominence is discussed in Section 10.2.
[3]`http://libra.inf.usi.ch`
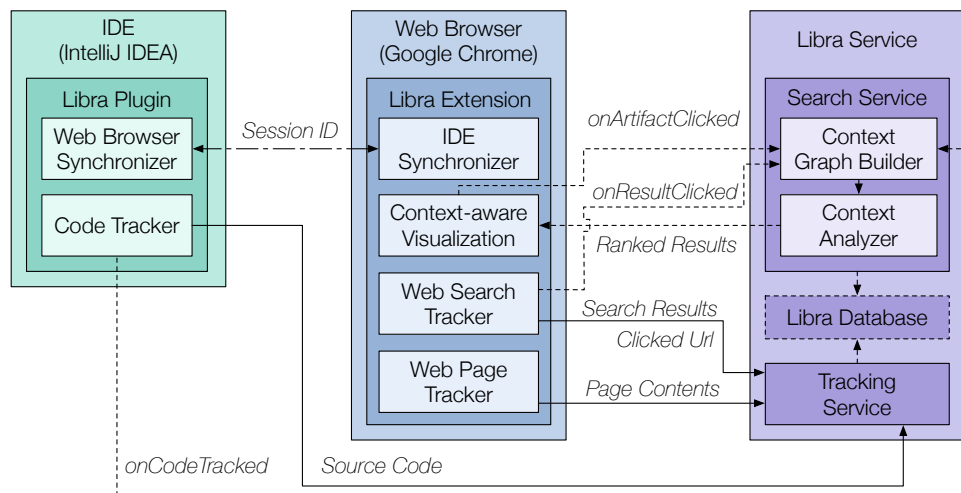
Figure 10.2. The Libra architecture.

Tracking Developer Context

Similarly to MYLYN [KM06], LIBRA aims at tracking the elements that are modified or created during a development task. LIBRA goes beyond the boundaries of the IDE, as it tracks developers' activities both in the IDE and in the web browser, thus targeting source code and web pages respectively. As depicted in Figure 10.2, the *Libra Plugin* is responsible of tracking the code written or accessed by the developer. Whenever a developer opens a text file in the IDE (*e.g.,* Java code, XML files, documentation, or logs), the content is sent to the *Libra Service* to be parsed, modeled, and stored as context resource.

The browser's *Libra Extension* searches for queries performed on the search engine, as well as every other URL opened by the browser. The *Web Search Tracker* is responsible of checking whether a tab in Google Chrome corresponds to a search page, and, in such a case, to monitor new queries. When a search is performed, all URLs composing the result set of the search engine are sent to the *Tracking Service* to be processed and stored in a cache (used for the sake of performance). If the URL points to a document in HTML format, the service renders the page and extracts the text, while it uses Apache Tika to extract textual contents from binary files (*e.g.,* PDF, Word Documents)[4]. In case a URL points to a YouTube video, the service automatically extracts the English audio transcriptions as contents of the page using GOOGLE2SRT[5]. Such transcriptions are either automatically generated or written by the author of the video.

The *Web Page Tracker* keeps track of the URLs opened by the developer, and acts as a "remote crawler" of the service. Indeed, instead of asking the *Tracking Service* to crawl a certain URL, this component sends the whole rendered content to the *Libra Service* so that it can be parsed, modeled, and stored as context resource.

Developers interacting with LIBRA are assigned an ID generated by the *Libra Plugin* or the *Libra Extension*. Both components need to have synchronized IDs to identify the same user: every request sent to the service requires a check between the *IDE Synchronizer* and the *Web Browser Synchronizer* to set the same session ID on both sides (see double arrow in Figure 10.2). This solution allows LIBRA to work with or without the IDE. The *Tracking Service* performs its

---

[4]Supported file types are listed in the Tika website: `http://tinyurl.com/hksl9hr`.
[5]`http://google2srt.sourceforge.net/en/`

operations if and only if the domain of the opened URL matches a white list of domains specified by the developer in the option panel. This limitation does not apply to the *Web Search Tracker*, which tracks any resource opened from a Google search while LIBRA is running. This ensures that the resources visualized in the bubble chart are the same reported in the search results' list.

Interacting with Libra

The *Web Search Tracker* is responsible of instrumenting Google's result page such that LIBRA knows when a result is clicked by the developer. When this happens, the corresponding URL is opened in a new tab, and the URL is sent to the *Tracking Service* to become part of the context resources. The *Web Search Tracker* then notifies the *Search Service* to create and analyze the new context graph, and sends the results to the *Context-Aware Visualization* to display the updated information to the developer. Similarly, either the *Context Aware Visualization* in the browser, and the *Code Tracker* in the IDE, notify the *Search Service* whenever a result is opened from LIBRA's user interface or a new piece of code has been tracked from the IDE. In doing so the visualization is always updated to the last context available.

10.2 Holistic Approach

We detail the approach used by LIBRA to process and analyze the results returned from web searches. We discuss how we parse and model the information within artifacts, and how we employed *HoliRank* to analyze the complementarity and prominence of the results in a holistic fashion.

10.2.1 Content Parsing and Meta-Information Model

Whenever an artifact is sent to the *Tracking Service*, the contents are parsed using the STORMED island parser, capable of identifying complete and incomplete multi-language elements—*e.g.,* written Java, JSON, XML, Stack Traces—immersed in natural language paragraphs, and to model such contents as a Heterogeneous Abstract Syntax Tree (H-AST) that allows visiting and manipulating the results.

In Chapter 7, we developed the concept of meta-information system that models specific aspects of the information. Following the same blueprint, we devised the following meta-information to model the contents of every resource processed by LIBRA:

Types: It represents the set of Java types mentioned in a resource. We consider all the H-AST nodes matching a reference type (either fully qualified or simple name), and primitive types (*e.g.,* int, double).

Variable Declarations: All H-AST nodes matching a variable and class field declaration.

Method Declaration: All H-AST nodes matching a method declaration.

Method Invocations: All H-AST nodes matching a method invocation and the name of the invoked method.

Identifiers: All H-AST nodes matching an identifier that can be visited in any extracted constructs (*e.g.,* full method and class declarations).

XML Elements: All H-AST nodes matching an XML element like a single tag (*i.e.,* `<tagname/>` or `<tagname>`) or a double tag (*e.g.,* `<tagname></tagname>`).

JSON Members: All H-AST nodes matching a JSON member (*i.e.,* `"field": element`).

Natural Language: We complement the meta-information with pure textual information, for example a term frequency map that can be reused to compute, for example, a textual similarity measure like *tf-idf* [MRS08].

10.2.2 Reusing HoliRank

In Chapter 9 we defined HOLIRANK and used it as the core of a summarization approach. In this chapter, we use HOLIRANK at the artifact level instead of using it at the information unit level. The construction of the similarity graph remains the same, where edges between two resources exists if and only if a similarity threshold is surpassed. Also the construction of the similarity vector $V_{x,y}$ remains the same. For convenience, here we report the construction of Equation 9.2 described in Chapter 9. Consider two resources R_x and R_y. Let $T_{x,y}$ be the set of shared types of meta-information between the resources, and let $M(R,t)$ be the meta-information of type t for the resource R. We define the similarity vector $V_{x,y}$ as:

$$V_{x,y} = \langle v_0, \ldots, v_{|T_{x,y}|} \rangle$$
$$\textit{with } v_i = M(U_x, t_i) \sim M(U_y, t_i) \textit{ and } t_i \in T_{x,y}$$

where each element v_i of the vector $V_{x,y}$ represents the similarity value between two homogeneous meta-information, and ranges in the interval $[0,1]$.

The main difference with Chapter 9 lies in the similarity function. Differently from Equation 9.3, we calculate the general similarity between two resources R_x and R_y as the average of the vector $V_{x,y}$:

$$f_{sim}(R_x, R_y) = \overline{V_{x,y}} \tag{10.1}$$

which still gives a value in the range $[0,1]$, and is used to build edges among resources by using a threshold based approach (*e.g.,* > 0.1). Conceptually, there is no difference between the algorithm devised in Chapter 9, since the meta-information model abstracts away the nature of the entity used in the graph. Even for the similarity function, the difference is subtle. Given the absence of a standardized way of evaluating similarity at the heterogeneous level, here we are experimenting a different function to aggregate heterogeneous information that might results in a simpler function than Equation 9.3. Experimenting different functions with the aim of discovering which one better defines a *holistic* similarity, requires deep investigation and experimentation that is left as future work.

10.2.3 Analyzing Context Resources

Our approach is based on the metrics *context complementarity*, *result prominence*, and *information quantity*.

Context Complementarity

The context complementarity measures the information intake provided by a resource in the current context of the developer. We use HOLIRANK to build the similarity graph CG of the

recently used context resources (the code recently written/modified in the IDE and the recently navigated web pages). LIBRA considers as "recent" what the developer dealt with in the past four hours. This is a settable parameter. For each resource R in the search engine result set, we create an additional similarity graph CG_R by adding R to the set of vertices of CG, and for each vertex V_{CG} in CG we add an edge from R to $V_{CG} >$ whose weight is equal to $f_{sim}(R, V_{CG})$. For each graph CG_R, we run HOLIRANK to compute the centrality of the resource R, ranging in $[0, 1]$. The higher the centrality of R in the graph, the lower the context complementarity: a higher centrality implies a tight relationship with many resources of the context, indicating a low information intake of R since R is similar to what is already composing the context. We define context complementarity as:

$$CtxComplementarity = 1.0 - HoliRank(R, CG_R) \tag{10.2}$$

Result Prominence

The result prominence identifies prominent results among the search engine result set. Even though a set of results matched by a query can be more or less relevant, there is often an overlap of the information provided by different artifacts. For example, an artifact is a tutorial on a specific topic, while another artifact tackles a programming problem on the same topic. If a result overlaps with many other results, it probably provides diversified information in its contents. If we model a similarity graph of the result set, a high overlap of the information of a result R with other results in RS, would result in a more prominent (central) position of R in the graph. We build a similarity graph G_{RS} containing all results R in the results set RS. We use *HoliRank* to estimate the centrality of a resource R in the graph G_{RS}:

$$ResultProminence = HoliRank(R, G_{RS}) \tag{10.3}$$

Information Quantity

The information quantity sums up the number of "elements" identified by our meta-information system. For example, for the *Natural Language* meta-information we consider the total amount of terms (after text preprocessing), to which we sum the number of declarators identified by *Method Declarators* meta-information, the ones identified by the *Variable Declarators* meta-information, *etc.* Counting information elements allows discriminating between two resources with the same size but with different contents. Consider for example two resources R_1 and R_2 having the same amount of characters, and –after preprocessing– the same terms. However, R_1 provides just text, while R_2 provides text and code. In this case, we consider the information quantity of R_2 is higher than the one of R_1.

10.3   Study I: Controlled Experiment

The *goal* is to evaluate LIBRA in terms of its (i) ability in correctly assessing for each query search result its prominence and complementarity with respect to the context, and (ii) usefulness to developers during a development or maintenance task. The *context* consists of *participants*, *i.e.,* third-year CS Bachelor students, and *objects*, *i.e.,* a University career management app and four maintenance tasks.

The study addresses the following research questions:

RQ$_1$: *How accurate is* LIBRA *in assessing the prominence and complementarity of query search results?* We investigate if the prominence and complementarity of information computed by LIBRA for a set of query search results Q_r is aligned with the developers' perception of prominence and complementarity.

RQ$_2$: *Does* LIBRA *help developers to complete their tasks correctly?* We investigate if the use of LIBRA helps developers when performing coding activities and to what extent—within an available time frame, and when working with or without LIBRA—they are able to correctly complete development and maintenance tasks.

10.3.1 Context Selection

We ran a controlled experiment with 16 3rd year CS Bachelor students at the end of a "Mobile Apps Development" course taught at the University of Molise (UniMol). Students learned about the design and implementation of Android apps, and were also required to develop an app.

We asked each participant to perform two programming tasks, one with and one without LIBRA. All tasks focused on the source code of `MyUnimol`, an app used by UniMol students to register for exams, visualize their marks, *etc.* All tasks were real implementation tasks performed by the `MyUnimol`'s developers in the past. We extracted these tasks from the app's issue tracker and versioning system (both repositories are private, and the study participants could not access them at any time).

We selected the four tasks from the issue tracker based on their type (two bug fixes and two enhancements) and difficulty (non-trivial, but doable in a limited amount of time). Then, we checked out from the versioning system the four snapshots of the app preceding the commit fixing each of the four issues. Participants worked on the specific version of the app related to the task they had to perform. The four tasks are:

T_1: When the user taps the "Logout" button in the right-up corner of the `MyUnimol` GUI, the app logs out the user without asking for confirmation. You are asked to add a confirmation dialog that pops up when the user taps logout. The dialog asks the user if she really wants to logout the app, providing as possible choices "Yes" and "No". If the user taps "Yes", the app logs her out, otherwise the confirmation dialog disappears. Make sure that the confirmation dialog is legible.

T_2: `MyUnimol` includes an address book with the contacts of all the University employees. A contact can have multiple phone numbers. When visualizing the details of a contact, all phone numbers associated to it are shown in a single string separated by a comma. This does not allow tapping on the phone number to start a call. You are asked to modify the view implementing the contact's details, showing each phone number associated to the contact in a separated field. It must be possible to tap a number to start a call.

T_3: There is a bug in the view allowing students to visualize personal details. The name and the student number shown on screen are not correct (*i.e.,* they are not the ones associated to the logged student who is visualizing her personal details). Also, tapping the "Back" button in this view makes the app crash. Fix the bug.

T_4: When a student logs in, `MyUnimol` loads in the home view a pie chart showing the exams already taken/to take. This also happens when the student comes back to the home view from another section of the app. The pie chart is shown through an animation that glitches, restarting multiple times (instead of loading the pie chart just once). Fix this animation.

10.3.2 Study Design and Procedure

Each participant was assigned two tasks to perform during the controlled experiment. We devised two possible pairs of tasks to assign to each participant: The first pair (T_1, T_2) includes two tasks related to the enhancement of existing features. The second pair (T_3, T_4) includes tasks dealing with bug-fixing activities. We wanted each participant to work on a pair of similar tasks (*e.g.,* two bug-fixing activities or two enhancements) having a comparable difficulty level in order to evaluate the effect of Libra during enhancement and bug-fixing tasks. For the purpose of RQ_2, this highlights the effect of LIBRA on the participant's performance.

Participants were equally partitioned into the eight groups shown in Table 10.1, reporting the study design.

Table 10.1. Study I: Design (L=Libra, NL=No Libra).

	Groups							
	A	B	C	D	E	F	G	H
Session 1	T_1(L)	T_1(NL)	T_2(L)	T_2(NL)	T_3(L)	T_3(NL)	T_4(L)	T_4(NL)
Session 2	T_2(NL)	T_2(L)	T_1(NL)	T_1(L)	T_4(NL)	T_4(L)	T_3(NL)	T_3(L)

The design is conceived in such a way that each participant worked both with and without LIBRA. To avoid learning effects, each participant had to perform different tasks across the two sessions. Different participants worked with and without LIBRA in different order and on two different tasks. When assigning participants to the eight groups, we made sure that their level of experience was (roughly) uniformly distributed across groups. We collected the (claimed) experience of participants via a pre-questionnaire. We also collected information related to the typical sources of information participants consult during coding activities. We carried out a pre-laboratory briefing in which participants were trained on the use of LIBRA through a running example and the laboratory procedure was illustrated in detail. We made sure not to reveal the study research questions. The training was performed on tasks not related to the ones of the experiment to avoid a bias in the results.

Participants had to perform the study in two sessions of 75 minutes each, interleaved by a break of 30 minutes to avoid fatigue effects. During the break participants did not exchange information. Participants were allowed to use whatever they wanted to complete the tasks including any material available on the Internet. At the end of each session, each participant provided the code she implemented and answered a three-part post-questionnaire. The first part, to check for problems with the experimental design, was composed of questions in which participants had to express their level of agreement on a Likert scale going from 1 (absolutely no) to 5 (absolutely yes) to the following claims:

1. *The overall activity to be performed was clear.*

2. *The description of the task to implement was clear.*

3. *There was enough time to perform the task.*

4. *The task was easy to implement.*

The second part was aimed at collecting qualitative information about LIBRA's usefulness. The following questions were only answered by participants who completed a task performed with LIBRA:

1. *How useful were the prominence, complementarity, and information quantity indicators provided by* LIBRA*?* Possible answers used a 5-point Likert scale from 1 (not useful at all) to 5 (very useful) for each indicator. *Why? Please motivate your previous answer.*

2. *How often did you use* LIBRA *in your Web searches?* Possible answers on a five-point Likert scale: 1 (never), 2 (in ∼25% of the searches), 3 (in ∼50% of the searches), 4 (in ∼75% of the searches), 5 (always).

3. *How would you improve* LIBRA*?*

The third part of the questionnaire was aimed at collecting information useful to answer RQ_1 and was only answered by participants who just completed a task performed with LIBRA. We showed to the participant the visualization depicted by LIBRA for the last search she performed. For each of the web documents projected on the LIBRA chart, we asked: *Do you agree with the assessment of the prominence of the document performed by* LIBRA*?* The same question was also asked with respect to LIBRA's assessment of the document complementarity with the context. Both questions were answered with a Likert scale going from 1 (strongly disagree) to 5 (strongly agree).

10.3.3   Variable Selection and Data Analysis

A normality check using the Shapiro-Wilk test indicated a statistically significant deviation from normal distribution (p-value< 0.05); hence we use non-parametric statistics. For all tests we consider a significance level $\alpha = 5\%$.

We answer RQ_1 by showing box plots of the participants' answers to the questions in the 3rd part of the post-questionnaire to evaluate LIBRA's assessment of the documents' prominence and complementarity.

We also statistically check, using the Wilcoxon signed-rank test [She07], whether the average agreement is greater than 3 (*i.e.,* at least weak agreement), by testing the null hypotheses $H_{0pr} : \overline{pr} \leq 3$ and $H_{0cm} : \overline{cm} \leq 3$, where $\overline{pr}$ and $\overline{cm}$ are the average (perceived) document prominence and complementarity.

The dependent variable to answer RQ_2 is *task completeness.* We asked a developer of `MyUnimol` (not involved in the study) to act as "evaluator" by reviewing the code implemented by the participants. The evaluator did not know the goal of the study nor which tasks were performed with-/without LIBRA. We provided a checklist to assign a completeness score to each of the sub-tasks implemented by participants. The completeness percentage of each sub-task was proportional to its difficulty (as estimated by the authors) and complexity. For example, the checklist for task T_1 was: (i) confirmation dialog view implemented and linked to the logout button ($+30\%$), (ii) behavior of the confirmation dialog implemented ($+40\%$), (iii) proper UI theme set ($+30\%$).

The main factor and independent variable is the presence/absence of LIBRA. Other potentially influencing factors are the (possible) different difficulty of the two tasks, the participants' (self-assessed) *Skills* in Java/Android development and years of *Experience* in Java/Android development.

To answer RQ_2, we show box plots of task completeness distributions for the two treatments, and also compare the results using the Wilcoxon signed-rank test. Since we do not know *a priori* in which direction the difference should be observed, we use a two-tailed test. We also assess the magnitude of the observed difference using Cliff's delta (d) effect size [GK05], suitable for non-parametric data. Cliff's d ranges in the interval $[-1, 1]$ and is negligible for $|d| < 0.148$, small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

We check the influence of the co-factors (ability, experience levels, and order in which tasks were performed) and their interaction with the main factor, using permutation test [Bak95], a non-parametric alternative to ANOVA, which does not require normally distributed data. We set the number of iterations of the permutation test procedure to 500,000 to ensure that results did not vary over multiple executions.

To analyze the post-experiment questionnaire results we use descriptive statistics, and tested, using the Wilcoxon signed-rank test, the null hypothesis $H_{0ag} : \overline{ag} \leq 3$ ($\overline{ag}$ is the average agreement level), to assess if there has been a weak or strong agreement.

10.3.4 Study Results

The population involved in this study has 2.8 years of experience in programming on average, with a maximum of 5 and a median of 3.0. They have a median of 2.5 years of Java programming experience (mean=2.2) and 3.5 months of Android development (maximum 1 year, minimum 1 month). Most of the participants learned how to develop an Android application while attending the "Mobile Apps Development" course at the University of Molise. Participants felt to have a good experience in Java programming with a median of three (medium experience), and a low experience in Android development (median=1.5, between very low and low). Concerning the sources of information exploited when programming, participants declared Q&A websites (median=5) as the most exploited, followed by forums (4), video tutorials (4), and official documentation (3.5).

How accurate is Libra in assessing the prominence and complementarity of query search results?

Figure 10.3 reports the level of agreement of the participants with the prominence and complementarity indicators provided by LIBRA for the ten documents retrieved by Google in the last search they did while performing the four tasks with LIBRA, thus adding up to 160 documents.
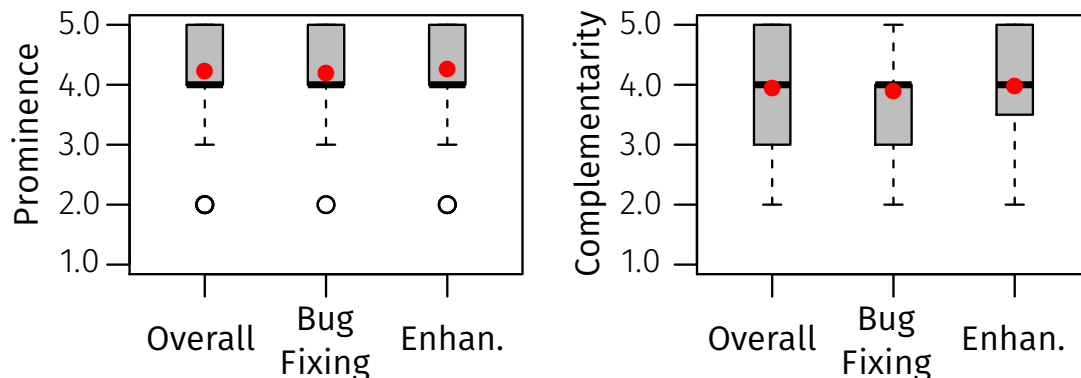
Figure 10.3. Participants' agreement with Libra's indications of prominence and complementarity: 1=strongly disagree, 5=strongly agree.

Participants agreed with the prominence indicator (left side of Figure 10.3) provided by LIBRA (median=4), and H_{0pr} can be rejected (p-value< 0.001). Only for five out of the 160 documents (3%), participants disagreed (Likert scale score=2) with LIBRA's prominence assessment. They agreed (4) for 62 (39%) or strongly agreed (5) for 70 (44%) documents. This is consistent both for

bug fixing activities and enhancing existing features. Concerning the complementarity indicator (right side of Figure 10.3), the agreement was fairly high (median 4), both for bug fixing and enhancement activities (in both cases the median=4). In this case, participants disagreed (2) with LIBRA's complementarity assessment on 8 documents (5%), while they agreed (4) for 73 (46%) or strongly agreed (5) for 43 (27%) documents. Thus, also the complementarity indicator provides precise insights to the LIBRA's users, and H_{0cm} can be rejected (p-value< 0.001) as well.

Does Libra help developers to complete their tasks correctly?

Figure 10.4 shows box plots of completeness achieved by participants with (LIBRA) and without (NOLIBRA) LIBRA.
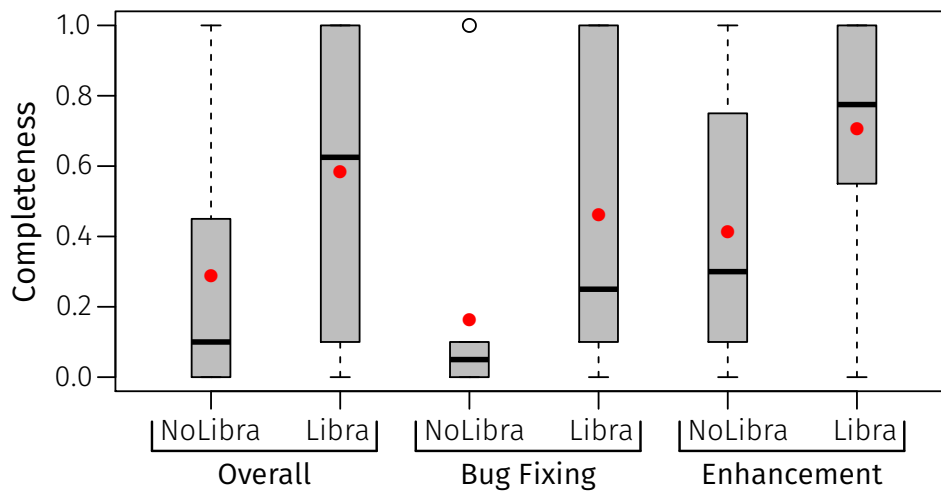
Figure 10.4. Completeness achieved by participants with the two treatments.

Participants using LIBRA achieved a higher completeness. The LIBRA median is 62% (mean 58%) against the 10% median (mean 29%) of NOLIBRA. In other words, LIBRA allowed participants to achieve a median additional correctness of 52% (mean of 29%). The Wilcoxon paired test (Table 10.2) indicates the presence of a statistically significant difference, with a p-value=0.02. Cliff's $d = 0.42$ indicates a *medium* effect size.

Table 10.2. Study I: Wilcoxon p-value and Cliff's d.

Tasks	p-value	d
Overall	**0.035**	**0.42 (Medium)**
Bug Fix	0.170	0.50 (Large)
Enhancement	0.180	0.45 (Medium)

Figure 10.4 and Table 10.2 report the completeness results for bug fixing and enhancement activities. LIBRA helped participants in both types of tasks, increasing the median completeness achieved for bug fixing activities by 25%, and for enhancement activities by 37%. The results of the Mann-Whitney paired two-tailed test indicates that in both types of tasks the difference is not significant (p-value>0.05) and the effect size is medium and large for bug fixing and enhancement activities, respectively. The non-significant results is explainable by the low number of data points

(8 participants for each type of task). Indeed, as explained above, the performance improvement provided by LIBRA is evident from the box plots in Figure 10.4 and statistically significant when considering the dataset as a whole.

The analysis of the post-questionnaires reveals that the usefulness of the information provided by LIBRA was also perceived by participants while performing the coding tasks. Table 10.3 reports the number of participants assessing the usefulness of the three indicators on each of the five levels in the considered Likert scale.

Table 10.3. Study I: Perceived usefulness of Libra's indicators.

Indicator	Not useful at all	Not useful	Neutral	Useful	Very useful
Prominence	0	1	4	7	4
Complementarity	0	2	4	6	4
Information Quantity	0	0	5	5	6

Out of the 16 participants, 11 (69%) found the prominence indicator useful or very useful, 4 remained neutral, while 1 found them not useful. Similar results were achieved for complementarity and information quantity. Participants used LIBRA in 59% of their web searches. 3 participants claimed to have used it only in 25% of their web searches. One of them was the most experienced in Android development, and achieved high completeness both with (100%) and without (90%) LIBRA, *i.e.,* he did not need to look online for help while performing the required tasks. The other two participants simply claimed to have used it only when they were not able to spot the useful web page to open in the search results.

We statistically analyzed the effect of co-factors. Permutation tests indicated that none of the ability/experience factors collected in the pre-questionnaire had an effect on the task's completeness, nor it interacted with the study treatment, *i.e.,* availability of LIBRA (p-values were in all cases way greater than 0.05). Similarly, no significant effect of the task ordering was observed. We also collected from the participants recommendations on how to improve our tool. These improvements mostly concern LIBRA's user interface and are currently being implemented.

Summing up

The study results indicated that both prominence and complementarity indicators reflect developers' perception of such measures, and are considered as useful/very useful indicators. LIBRA helped study participants to achieve a significantly better task completeness than the control group, though differences are not statistically significant when considering task types (*i.e.,* bug fixing and enhancement) separately due to the limited number of data points.

10.3.5   Threats to Validity

Construct Validity

Threats to *construct validity* mainly concern imprecisions in the measurements made. A major challenge is to measure dependent variables related to RQ_1 (agreement with LIBRA) and above all RQ_2 (task completeness). For the former, we relied on developers' perceived agreement with LIBRA's assessment of documents' prominence and complementarity. For the latter we used a checklist-based approach. We are aware that results of such an approach might be influenced by

the evaluator's subjectiveness, as well as by the weights we gave to each task (to account for its complexity).

Internal Validity

Threats to *internal validity* concern confounding factors that could influence the results. First, as explained Section 10.3.2, we have used permutation test to analyze the effect of such factors, and also have been supported by the post-study questionnaires results. All participants strongly agreed/agreed about the clarity of the activity (mean 4.9, median 5, H_{ag} rejected with p-value$<$ 0.001) and tasks (mean 4.8, median 5, H_{ag} rejected with p-value$<$ 0.001). They weakly agreed on time (mean 4.1, median 4.5, H_{ag} rejected with p-value$<$ 0.001), and had mixed opinions about the tasks' difficulty (mean 3.2, median 3, H_{ag} not rejected with p-value=0.27). This should not be considered as a possible threat as it is normal to find people experiencing different difficulty and productivity levels. This indicates the absence of a possible ceiling effect.

Conclusions Validity

Threats to *conclusion validity* concern the relationship between treatment and outcome. The main issue here is the possible presence of Type II errors—due to the limited number of study's participants—every time we could not reject a null hypothesis. In our study this happened when analyzing completeness results for different types of activity separately.

External Validity

Threats to *external validity* concern the generalization of our findings. The controlled experiment has clear limitations (needed to achieve a high level of control) in terms of objects' characteristics and domain, and in terms of participants. To mitigate this threat due to the limited experience of the participants, we have conducted a second, qualitative study with experienced practitioners, described next.

10.4   Study II: Industrial Applicability

A successful technological transfer is the main target objective for each prototype tool. Thus, the *goal* of this second study is to investigate LIBRA's industrial applicability by answering the following research question:

RQ$_3$: *Would practitioners consider exploiting* LIBRA *in their daily coding activities?*

The study *context* consists of the 5 *participants* listed in Table 10.4. We conducted semi-structured interviews to get qualitative feedback on both the tool and the underlying approach. Before each interview, one of the authors performed a demo of LIBRA to show its features to the participants. Then, we let the participant interact with the tool, performing web searches on the topics related to task T_3 of Study I. Each interview lasted ca. 1.5 hours and was based on a think-aloud strategy. After each interview we asked the questions listed in Table 10.5. The interviews were conducted by two of the authors.

Table 10.4. Study II: Participant's.

	Developer	Position
P1	Giuseppe Socci	Project Manager @ Genialapps
P2	Luciano Cutone	Project Manager @ IdeaSoftware
P3	Carlo Branca	Developer @ Capgemini
P4	Giovanni Grano	Senior Developer @ Cedacri
P5	Matteo Merola	Full Stack Developer @ Cleopa

Table 10.5. Study II: Questions for the interviews.

Question	Text
Q1	Mobile Development Experience
Q2	Do you find Libra useful?
Q3	Importance of prominence
Q4	Importance of complementarity
Q5	Importance of information quantity
Q6	Are you willing to use Libra for your activities?

10.4.1 Results

Table 10.6 reports the participants' answers to the questions we asked to drive our interview. Giuseppe and Matteo expressed concerns about LIBRA's usability. In Giuseppe's opinion the graph-based interface provides too many details: "*You could think of a single metric that provides an indication of both complementarity and prominence, which could be used to indicate the overall usefulness of each page returned by Google. In this way it would be immediate for the user to identify which one is the better page for* LIBRA. *Then—and only if necessary—the user can analyze the chart to better understand why* LIBRA *is indicating a specific page.*"

We discarded this option since during a specific phase of a coding activity, a developer might be interested in reading documents that have a high prominence but a low complementarity with the context (*i.e.,* she may want to dig deeper into topics overlapped with her context), while in some other phases she might be interested in highly complementary documents (*i.e.,* she may want to broaden her knowledge). Thus, we do not see prominence and complementarity as direct indicators of "document quality". Similar usability concerns were expressed by Matteo.

Despite some reservations about the LIBRA's usability, both Giuseppe and Matteo expressed their desire to use LIBRA in their daily coding activities. Giuseppe liked the idea behind the tool, and would like to use it more: "*I should use* LIBRA *for much more time to better assess its usefulness. However, from this first experience I can say that* LIBRA *seems to be an interesting tool. I particularly like the idea to add information to the Google ranking. This can be particularly useful when you do not know exactly what you need, i.e., your query is rather generic*".

The other three participants provided enthusiastic comments about LIBRA, and would definitively like to use it while coding. One representative comment is the one by Luciano: "LIBRA *is a very interesting tool. Google provides accurate results in general. However, searching for pages related to software development is more challenging. When I use Google for my daily coding activities, I often need to open almost all documents in the first results' page to identify the most appropriate web page to read. In the hour I spent using* LIBRA, *I noticed that it allowed me to find the most appropriate web page quicker. Instead of analyzing ∼10 pages for each query, I have analyzed 2 or 3 pages, generally the one with the highest prominence, the one with the high-*

Table 10.6. Study II: Participant's answers to the questions explicitly asked.

	P1	P2	P3	P4	P5
Q1	5+ years	5+ years	1+ year	1+ year	1+ year
Q2	Maybe	Absolutely yes	Yes	Absolutely yes	Maybe
Q3	Very high	Very high	Very high	High	Very high
Q4	High	Very high	High	Medium	Medium
Q4	Low	Very high	Medium	Low	Medium
Q5	Absolutely yes	Absolutely yes	Yes	Absolutely yes	Yes

est complementarity and (if needed) another one in between". Giovanni particularly appreciated LIBRA's integration with the development workflow: "*I tried many different tools, but most of them are either hard to use or to integrate in the developer's workflow. The main strength of LIBRA is that it is integrated into the classical developer's workflow, which is programming and searching for information on Google, without adding any complexity: LIBRA does not create any barrier between the developer and her usual working environment; LIBRA just quietly guides the developer to the most useful results*". Carlo also appreciated LIBRA, and positively judged its usefulness and usability.

Participants agreed on the usefulness of prominence and complementarity, but less so for information quantity. Giuseppe explained: "*I do not care about information quantity since in my experience technical web pages are not so long*".

The participants provided several suggestions on how to improve LIBRA. Giuseppe suggested: "*provide an indication on the cohesiveness of the returned page with the query, to see how focused the page is with respect to the query. If you need information on a specific technology, you need a page that is extremely focused on the query, but if you need to learn a new technology, you prefer a less focused page*". Luciano, commenting LIBRA's indicators, explained: "*All the three indicators are crucial. It could be worthwhile to show the social importance of each page, i.e., how many times the page has been shared on social networks and/or how useful was the page for the developers, similarly to the mechanisms in Stack Overflow*". Giovanni expressed concerns about the way LIBRA tracks the web pages; "*I often open web pages of which I read a very limited part. If LIBRA tracks those pages and considers them as part of the context, it could provide misleading information about the documents' complementarity.*" To overcome this issue, Giovanni proposed to track the visiting time and weigh the importance of the pages in the context, ignoring pages visited for brief periods.

Summing up

This study provided positive feedback on the usefulness and practical applicability of LIBRA and its integration in a developer's workflow. Participants provided feedback on how to possibly improve LIBRA, *e.g.*, by providing a simplified user interface. Clearly, tools can always be improved, given sufficient time and human resources.

10.5 Conclusions

A crucial activity in modern software development is the acquisition of the pieces of information. These pieces are located in diverse places and are of a heavily heterogeneous nature. Instead of manually combing through the results of general purpose search engines or heeding the

monochromatic suggestions of the recommender systems proposed so far, we tackled the problem by developing a *holistic* approach, based on a meta-information system, capable of dealing with the heterogeneous nature of web resources. Our approach, implemented in a tool named LIBRA, aids developers to interact with the suggestions, and seamlessly blends into their workflow. The empirical evaluation of Libra provides evidence that a *holistic* analysis of a developer's information context can offer comprehensive and contextualized support to information navigation and retrieval during software development.

Part V

Epilogue

11

Conclusions and Future Work

In this dissertation we asserted that a holistic modeling and analysis of the information provided by development artifacts reaches a higher level of abstraction on the information itself, needed as the basis for a H-RSSE.

We presented the current state of the art in modeling and analyzing information for RSSEs, by highlighting the limitations due to their lack of modeling and their latent reductionism behind such approaches. We devised a set of RSSEs following the current reductionist philosophy, fully relying on off-the-shelf approaches and reductionist analyses.

To move towards a *holistic* interpretation of the information, we devised an approach to parse and model the heterogeneous contents of software into a H-AST that represents, in a unique structure, both textual fragments, formats and languages of an artifact. On top of the H-AST, we devised a meta-information model to organize several types of information contained in the contents of an artifact, thus enabling semantic modeling of its contents.

To validate our thesis we developed a set of applications and analyses by leveraging both the H-AST and meta-information model. We showed how these additional modeling stages favor reusability, and how a higher abstraction provided on the information can be leveraged to even analyze non-textual artifact (*i.e.,* video tutorials). We improved off-the-shelf approaches, by extending them with *holistic* analyses of the information, and laid the foundations for a novel type of recommender systems, H-RSSE, which reported to effectively provide developers with a better support during development.

11.1 Contributions

In this section we briefly summarize the contributions of this dissertation to the current state of the art. We follow the structure of the dissertation, starting from parts, and digging into chapters, to highlight their contributions.

11.1.1 Reductionist RSSEs

In the second part of this dissertation, we devised a set of RSSEs by reusing off-the-shelf approaches, following the reductionist philosophy latently present in the state of the art.

SeaHawk

In Chapter 3 we presented SEAHAWK, a RSSE which leverages the crowd knowledge of Stack Overflow and integrates it in the IDE. SEAHAWK is a prominent example of reuse of off-the-shelf tools from information retrieval (*i.e.,* Apache Solr) to analyze development artifact like Stack Overflow discussions.

Prompter

In Chapter 4 we presented PROMPTER, a RSSE which automatically searches, evaluates, and recommends Stack Overflow discussions in the IDE. PROMPTER uses a multi-faceted ranking model that takes in account textual, code, and community aspects of both the Stack Overflow discussions and the code context in the IDE. The full pipeline of PROMPTER is created by reusing off-the-shelf tools (*i.e.,* Eclipse JDT) to analyze code information in the Stack Overflow discussions. The controlled experiments run with PROMPTER showed that our approach is effective for maintenance tasks.

Stack Overflow Low Quality Post Detection

In Chapter 5 we presented results of an industrial collaboration with Stack Overflow where we developed an approach to evaluate the quality of Stack Overflow discussions at creation time. The model proposed used a combination of several metrics concerning either textual aspects (*e.g.,* readability, characters count) and community related aspects (*e.g.,* popularity). Their combination revealed to be the best quality indicators, outperforming the solution in use at Stack Overflow.

11.1.2   Parsing and Modeling Unstructured Data

In the third part of this dissertation we tackled the problem of modeling the contents of development artifacts to go beyond their pure textual representation.

Automated Multi-Language Parsing and Modeling

In Chapter 6 we presented an approach to parse and model the contents of development artifacts. We devised a multi-lingual island grammar capable of identifying and isolating Java code snippets, interchange formats (*i.e.,* XML, JSON), and stack traces immersed in the narrative of textual artifact. We devised the concept of Heterogeneous Abstract Syntax Tree (H-AST) to model all these heterogeneous elements in a unique structure.

StORMeD: Stack Overflow Ready Made Data

In Chapter 7 we took advantage of the approach devised in Chapter 6 to develop a series of applications and analyses reusing our approach. First of all, we build StORMeD, a dataset modeling the contents of more than 800K Stack Overflow discussions about Java. In StORMeD, we extended the structural model for a Stack Overflow discussion (*e.g.,* question, answer, comments, user information) by devising and including a meta-information model of the contents built on top of the H-AST. We reused StORMeD to perform an analysis to identify actual usages of the undocumented `sun.misc.Unsafe` class on Stack Overflow, unveiling how it attracts the most reputed users.

CodeTube

In Chapter 8 we presented CODETUBE, an approach to split video tutorials into coherent and self-contained fragments, which are automatically categorized in seven different categories (*e.g.,* "theoretical concepts", "code implementation", "working environment setup"). CODETUBE allows developers to retrieve video fragments, with additional related Stack Overflow discussions, by specifying a textual query, and desired category. Several analysis like OCR and ad-hoc image analysis are used in CODETUBE, together with the multi-lingual island parser and its H-AST defined in Chapter 6. The island parser is used to analyze the contents of frames, and decide how they should be merged together to compose fragments according to the code contained in them. We assessed CODETUBE with three studies, showing its effectiveness in splitting and classifying video tutorials into coherent and self-contained fragments.

11.1.3   Holistic RSSEs

In the fourth part of this dissertation we focused on the analysis of information from a *holistic* point of view by building upon the H-AST and meta-information models. The *holistic* interpretation of the information is leveraged to enable a novel type of *holistic* recommender systems.

HoliRank: Holistic PageRank

In Chapter 9 we revised LEXRANK, a textual summarization approach based on PAGERANK, by devising HOLIRANK, a customized LEXRANK using a *holistic* similarity function using the meta-information model to establish edges among elements in a graph.

We performed a preliminary comparative evaluation between a summary built with HOLIRANK and LEXRANK, suggesting that a holistic approach to analyze the heterogeneity of the information might lead to improved summaries.

Libra

In Chapter 10 we presented LIBRA, a H-RSSE that augments the search results of Google to help the developer navigating the information. LIBRA constructs a holistic meta-information model of the resources perused by a developer, and leverages HOLIRANK to estimate the complementarity of a resources within the developer informational context, the prominence of the results within the result set returned by Google. We evaluated LIBRA in a controlled experiment highlighting its effectiveness in providing developers with support in navigating the information.

11.2   Future Work

The last stop of the journey of this dissertation (Chapter 10) terminated with the design of the first H-RSSE: LIBRA. To reach this point, we performed several steps by modeling the contents of development artifacts by leveraging the multi-lingual island parser (see Chapter 6), devising its H-AST model and the meta-information model (see Chapter 7) built on top of it. This additional layer of abstraction provides new possibilities to explore and manipulate the information provided by development artifacts. In this section we describe some potential future work that might be derived from this thesis.

11.2.1 Holistic Data Aggregation

Aggregating heterogeneous data to establish a link between two development artifacts is a non trivial task that requires additional investigation. In Chapter 9 and Chapter 10 we tried to address such challenge by devising two *holistic* similarity function adopting two different solutions based on norm and mean. Several other unexplored solution might be devised involving also machine learning approaches. A deep investigation and experimentation of new approaches to aggregate holistic data is needed to understand which solutions achieve better performance.

11.2.2 Modeling and Assisting Holistic Navigation

LIBRA (see Chapter 10) provided developers with navigation support based on three dimensions: complementarity, result prominence, and information quantity. The solution proposed there left all the navigation decisions to the developer, and LIBRA just acted as *informational map* of the resources, providing indirect guidance. Predicting what the user might need in terms of the metrics used by LIBRA, might definitely be a point of extension of our work. Such an extension point requires the modeling of the behavior of user while navigating the information, by aggregating diverse information (*e.g.,* terms in the query, time spent on a resource) of previous searches performed by the user.

This prediction model might enable additional ways of recommending elements to the developer. For example, one could suggest artifacts depending on their distance between the predicted LIBRA's metrics values of the model behavior, and the actual LIBRA's metrics values of the artifact. In doing do, it is possible to provide an additional push notification system similar to the one used in PROMPTER (see Chapter 4), but relying on informational needs.

11.2.3 Reducing Information Overload

In Chapter 9 we tackled the problem of summarization for development artifacts, with specific focus on Stack Overflow. We used HOLIRANK to select the most prominent parts of a discussion to be included in an extractive summary. Possible extensions of this work might concern two aspects. The former regards a generalization of the type of artifact usable by the summarizer. In the current implementation we only manage Stack Overflow, thanks to the discussions already modeled by STORMED. Therefore, the summarizer might be extended to a general HTML page. For example, web sites like Stack Overflow define a data schema (*i.e.,* QAPage[1]) of the page. Such information might be leveraged to understand which parts to consider.

The latter extension concerns the full fledged integration of the summarizer in a H-RSSE like LIBRA. For example, whenever a developer opens up a result suggested by LIBRA, a summarized, yet interactive, page might be shown instead. The developer could thus require more information on demand in case it is needed. Additionally, HOLIRANK might be used to generate multi-document summaries, including video fragments produced by other approaches like CODETUBE.

11.2.4 Leveraging Developers Interaction

In Chapter 4 and Chapter 10, we needed to perform a minimal tracking of the interaction of the developer with the IDE and web browser. The tracking phase just concerned the code entities and web pages modified or perused by the developer, with the goal of recreating an informational context. A deeper exploration of the developer interaction, in particular in the IDE might be

---

[1]`http://schema.org/QAPage`

beneficial for the recommendation engine in general. For example, Minelli *et al.* [MML15] track events in the IDE, identifying when the developer enters the *"understanding"* phase or navigate the code. Such information can be leveraged to refine the informational context of the developer (on the IDE side), and select the best moment when to fire a notification.

11.3   Closing Words

In this dissertation we showed that modeling and analyzing data from a *holistic* point of view allows analyses to capture and leverage the intrinsic heterogeneity of the contents of development artifacts. We highlighted the importance of modeling as a needed and fundamental step to develop novel analyses and favors reusability.

   This thesis points out the need of new ways of treating information in the context of software engineering, and the need of applications capable of leveraging it to better support developers during programming tasks. A wider set of heterogeneous data that can be modeled and analyzed together to improve the holistic interpretation of the information available to developers, and thus devise a more powerful generation of H-RSSE.

   The only way to achieve this result is to stop being clients using off-the-shelf tools designed outside the software engineering field. We need to remember that, as software engineers, we need to become practitioners capable of understanding and tailoring methodologies to support and advance our own field.

Bibliography

[ACCL00] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of ICSM (16th IEEE International Conference on Software Maintenance)*, pages 40–51. IEEE CS Press, 2000. *(pp. 4, 12)*.

[ACD+08] Eugene Agichtein, Carlos Castillo, Debora Donato, Aristides Gionis, and Gilad Mishne. Finding high-quality content in social media. In *Proceedings of WSDM 2008 (1st International Conference on Web Search and Data Mining)*, pages 183–194. ACM, 2008. *(pp. 69, 71)*.

[AZBA08] Lada A. Adamic, Jun Zhang, Eytan Bakshy, and Mark S. Ackerman. Knowledge sharing and yahoo answers: everyone knows something. In *Proceedings of WWW (17th International Conference on World Wide Web)*, pages 665–674. ACM, 2008. *(p. 15)*.

[Bak95] Rose D. Baker. Modern permutation test software. In *Randomization Tests.* Marcel Decker, 1995. *(pp. 53, 179)*.

[BBL11] Alberto Bacchelli, Lorenzo Baracchi, and Michele Lanza. Remail -blending talk and work in eclipse. In *Proceedings of Eclipse-IT 2011 (6th Workshop of the Italian Eclipse Community)*, pages 303–306, 2011. *(p. 13)*.

[BCLM11] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *Proceedings of ASE 2011 (26th IEEE/ACM International Conference On Automated Software Engineering)*, pages 476–479, 2011. *(pp. 13, 17, 93, 94, 98, 101)*.

[BDWK10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the CHI 2010 (28th International Conference on Human Factors in Computing Systems)*, pages 513–522, 2010. *(pp. 3, 14, 25)*.

[BGL+09a] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Software*, 26(5):18–24, 2009. *(p. 3)*.

[BGL+09b] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of CHI 2009 (SIGCHI Conference on Human Factors in Computing Systems)*, pages 1589–1598. ACM, 2009. *(p. 12)*.

[BLM12] Alberto Bacchelli, Michele Lanza, and Ebrisa Mastrodicasa. On the road to hades– helpful automatic development email summarization. In *Proceedings of TAinSM 2012 (1st International Workshop on the Next Five Years of Text Analysis in Software Maintenance)*, 2012. *(pp. 159, 166)*.

[BLR10]   Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 375–384. ACM Press, 2010. *(pp. 13, 98)*.

[BNJ03]   David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003. *(pp. 4, 137)*.

[BP66]   Leonard E. Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The Annals of Mathematical Statistics*, 37(6):1554–1563, December 1966. *(p. 13)*.

[BP98]   Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of WWW 1998 (7th International Conference on World Wide Web)*, pages 107–117. Elsevier Science Publishers B. V., 1998. *(pp. 159, 160)*.

[Bra97]   Andrew P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997. *(p. 142)*.

[Bre01]   Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. *(p. 137)*.

[BSDL12]   Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content Classification of Development Emails. In *Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, 2012. *(p. 13)*.

[BW10]   R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010. *(p. 115)*.

[BWYS11]   S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining java class naming conventions. In *Proceedings of ICSM 2011 (27th IEEE International Conference on Software Maintenance)*, pages 93–102. IEEE CS Press, 2011. *(p. 98)*.

[BYRN99]   R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999. *(p. 134)*.

[CAG12]   Joel Cordeiro, Bruno Antunes, and Paulo Gomes. Context-based recommendation to support problem solving in software development. In *Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, pages 85–89. IEEE Press, 2012. *(p. 15)*.

[CBHK02]   Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, June 2002. *(p. 142)*.

[CdSdAM16] Eduardo C. Campos, Lucas B. L. de Souza, and Marcelo de A. Maia. Searching crowd knowledge to recommend solutions for api usage tasks. *Journal of Software: Evolution and Process*, 28(10):863–892, October 2016. *(p. 16)*.

[CH00]   Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000. *(p. 103)*.

[CKM03] Michael L Collard, Huzefa H Kagdi, and Jonathan I Maletic. An xml-based lightweight c++ fact extractor. In *Proceedings of IWPC 2003(11th IEEE International Workshop on Program Comprehension)*, pages 134–143. IEEE, 2003. *(p. 5)*.

[CL75] Meri Coleman and T. L. Liau. A computer readability formula designed for machine scoring. *Journal of Applied Psychology*, 60(2):283–284, April 1975. *(pp. 73, 115)*.

[CM03] D. Cubranic and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of ICSE 2003 (25th IEEE International Conference on Software Engineering)*, pages 408–418. IEEE CS Press, 2003. *(pp. 4, 35)*.

[CMSB05] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005. *(p. 13)*.

[Coh60] J Cohen. A coefficient of agreement for nominal scales. *Educational and Psychosocial Measurement*, 20:37–46, 1960. *(p. 127)*.

[Cor89] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, June 1989. *(pp. 11, 12)*.

[CPB+15] Luigi Cerulo, Massimiliano Di Penta, Alberto Bacchelli, Michele Ceccarelli, and Gerardo Canfora. Irish: A hidden markov model to detect coded information islands in free text. *Science of Computer Programming*, 105(C):26–43, July 2015. *(pp. 13, 17)*.

[CS13] Denzil Correa and Ashish Sureka. Fit or unfit: Analysis and prediction of 'closed questions' on stack overflow. In *Proceedings of COSN 2013(1st ACM Conference on Online Social Networks)*, pages 201–212. ACM, 2013. *(p. 16)*.

[CS14] Denzil Correa and Ashish Sureka. Chaff from the wheat: Characterization and modeling of deleted questions on stack overflow. In *Proceedings of WWW 2014 (23rd international conference on World Wide Web)*, pages 631–642. ACM, 2014. *(pp. 16, 69)*.

[CT91] Thomas Cover and Joy Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991. *(p. 72)*.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. In *Proceedings of UML '99 (International Conference on the Unified Modeling Language)*, pages 630–644. Springer, 1999. *(p. 5)*.

[DGLD05] Stéphane Ducasse, Tudor Gırba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. *Tools for Software Maintenance and Reengineering, RCOST/Software Technology Series*, 71:27, 2005. *(p. 5)*.

[DH08] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of OOPSLA 2008 (23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications)*, pages 313–328. ACM, 2008. *(p. 13)*.

[DR12]   Barthélémy Dagenais and Martin P. Robillard. Recovering traceability links be-
          tween an api and its learning resources. In *Proceedings of ICSE 2012 (34th Inter-
          national Conference on Software Engineering)*, pages 47–57. IEEE Press, 2012. *(p.
          13)*.

[DTD01]   Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. Famix 2.1the famoos
          information exchange model, 2001. *(p. 5)*.

[Dum04]   Susan T. Dumais. Latent semantic analysis. *Annual Review of Information Science
          and Technology*, 38(1):188–230, 2004. *(p. 4)*.

[ER04]   Günes Erkan and Dragomir R. Radev. Lexrank: Graph-based lexical central-
          ity as salience in text summarization. *Journal of Artificial Intelligence Research*,
          22(1):457–479, December 2004. *(pp. 159, 160, 161)*.

[Erl00]   Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*,
          2(3):17–23, May 2000. *(p. 11)*.

[FGG97]   Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers.
          Machine Learning, 29(2-3):131–163, November 1997. *(p. 13)*.

[FH82]   R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study:
          Report to Our Respondents. In Girish Parikh and Nicholas Zvegintzov, editors,
          Tutorial on Software Maintenance, pages 13–30. IEEE, 1982. *(p. 11)*.

[Fle48]   Rudolph Flesch. A new readability yardstick. *Journal of Applied Psychology*,
          32(3):221–233, June 1948. *(pp. 73, 115)*.

[For04]   Bryan Ford. Parsing expression grammars: A recognition-based syntactic founda-
          tion. In *Proceedings of POPL 2004 (31st ACM SIGPLAN-SIGACT Symposium
          on Principles of Programming Languages)*, pages 111–122. ACM, 2004. *(p. 94)*.

[GBR15]   Latifa Guerrouj, David Bourque, and Peter C. Rigby. Leveraging informal docu-
          mentation to summarize classes and methods in context. In *Proceedings of ICSE
          2015 (37th International Conference on Software Engineering)*, volume 2, pages
          639–642. IEEE Press, 2015. *(p. 159)*.

[GK05]   Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical
          approach*. Lawrence Associates, 2005. *(pp. 53, 178)*.

[GM09]   Max Goldman and Robert C. Miller. Codetrail: Connecting source code and web
          resources. *Journal of Visual Languages & Computing*, 20(4):223–235, August 2009.
          (pp. 14, 18).

[Gol89]   David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine
          Learning*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1989. *(pp.
          78, 83)*.

[GTGZ14] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Check-
          ing app behavior against app descriptions. In *Proceedings of ICSE 2014 (36th
          ACM/IEEE International Conference on Software Engineering)*, pages 1025–1035,
          2014. *(pp. 137, 154)*.

[Gun52]   Robert Gunning. *The Technique of Clear Writing*. McGraw-Hill, 1952. *(pp. 73, 115)*.

[HAMM10] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of WCRE 2010 (17th Working Conference on Reverse Engineering*, pages 35–44. ACM, 2010. *(pp. 159, 162)*.

[Has09]   Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of ICSE (31st International Conference on Software Engineering)*, pages 78–88. IEEE Computer Society, 2009. *(p. 41)*.

[HB08]   Reid Holmes and Andrew Begel. Deep intellisense: A tool for rehydrating evaporated information. In *Proceedings of MSR 2008 (5th IEEE International Working Conference on Mining Software Repositories)*, pages 23–26, New York, NY, USA, 2008. ACM. *(pp. 4, 13, 18, 35)*.

[HBM+13] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, pages 842–851. IEEE CS Press, 2013. *(pp. 3, 41)*.

[HBO+12a] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. Automatic query performance assessment during the retrieval of software artifacts. In *Proceedings of ASE 2012 (27th IEEE/ACM International Conference on Automated Software Engineering)*, pages 90–99. ACM, 2012. *(p. 41)*.

[HBO+12b] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Evaluating the specificity of text retrieval queries to support software engineering tasks. In *Proceedings of ICSE 2012 (34th International Conference on Software Engineering)*, pages 1273–1276. IEEE CS Press, 2012. *(p. 41)*.

[HDC11]   Björn Hartmann, Mark Dhillon, and Matthew K. Chan. Hypersource: Bridging the gap between source and code-related web sites. In *Proceedings of CHI 2011 (SIGCHI Conference on Human Factors in Computing Systems)*, pages 2207–2210. ACM, 2011. *(pp. 15, 18)*.

[HFW07]   Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: Finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of UIST 2007 (20th Annual ACM Symposium on User Interface Software and Technology)*, pages 13–22, New York, NY, USA, 2007. ACM. *(pp. 14, 17)*.

[HG09]   Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, September 2009. *(p. 71)*.

[HN98]   Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998. *(p. 47)*.

[Hol79]   Sture Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979. *(pp. 53, 66)*.

[Hol09] R. Holmes. Do developers search for source code examples using multiple facts? In *Proceedings of SUITE 2009 (Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation)*, pages 13–16, 2009. *(p. 169)*.

[HP00] Morten Hertzum and Annelise Mark Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing and Management: an International Journal*, 2000. *(p. 3)*.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012. *(p. 103)*.

[KDSH12] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes. Automatically locating relevant programming help online. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 127–134, 2012. *(p. 16)*.

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th International Conference on Software Engineering)*, pages 344–353. IEEE CS, 2007. *(pp. 3, 11)*.

[KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of FSE 2006 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 1–11. ACM, 2006. *(pp. 14, 172)*.

[KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, December 2006. *(p. 11)*.

[LAZCH13] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Proceedings of ESEC/FSE 2013 (9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*, pages 378–388. ACM, 2013. *(p. 44)*.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985. *(p. 11)*.

[Leh69] M. M. Lehman. The programming process. *IBM Res. Rep.*, September 1969. *(p. 11)*.

[Leh78] M. M. Lehman. *Programs, Cities, Students— Limits to Growth?*, pages 42–69. Springer New York, 1978. *(p. 11)*.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707–710, 1966. *(p. 42)*.

[Lia83] Frank Liang. *Word Hy-phen-a-tion by Com-put-er*. PhD thesis, Stanford University, 1983. *(pp. 73, 115)*.

[LM10]   Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer Publishing Company, Incorporated, 1st edition, 2010. *(p. 115).*

[LMC12]   Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the "hurried" bug report reading process to summarize bug reports. In *Proceedings of ICSM 2012 (28th IEEE International Conference on Software Maintenance)*, pages 430–439. IEEE Computer Society, 2012. *(pp. 4, 159).*

[LPO$^+$12] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *Proceedings of ICPC 2012 (20th IEEE International Conference on Program Comprehension)*, pages 193–202. IEEE Computer Society, 2012. *(p. 162).*

[LS80]   Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. *(p. 11).*

[LVD06]   Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, pages 492–501. ACM, 2006. *(pp. 3, 11).*

[MAS$^+$13] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of ICPC 2013 (21st International Conference on Program Comprehension)*, pages 23–32. IEEE Computer Society, 2013. *(p. 159).*

[McL69]   Harry G. McLaughlin. SMOG grading - a new readability formula. *Journal of Reading*, pages 639–646, May 1969. *(pp. 73, 115).*

[MCSD12] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. Ausum: Approach for unsupervised bug report summarization. In *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 1–11. ACM, 2012. *(pp. 4, 159).*

[MHG10]   Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., 2010. *(p. 23).*

[Mit97]   Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., 1 edition, 1997. *(p. 137).*

[MM03]   Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of ICSE 2003(25th International Conference on Software Engineering)*, pages 125–135. IEEE Computer Society, 2003. *(p. 4).*

[MML15]   Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer – an investigation of how developers spend their time. In *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension*, pages 25–35, 2015. *(pp. 11, 193).*

[MMM+11] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hart-
          mann. Design lessons from the fastest Q&A site in the west. In *Proceedings of CHI
          2011 (29th Conference on Human factors in computing systems)*, pages 2857–2866.
          ACM, 2011. *(p. 15)*.

[Moo01]   Leon Moonen. Generating robust parsers using island grammars. In *Proceedings
          of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22.
          IEEE CS, 2001. *(pp. 24, 93, 94, 98)*.

[MPM+15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias
          Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api
          in the wild. In *Proceedings of OOPSLA 2015 (International Conference on Object
          Oriented Programming Systems Languages & Applications)*, pages 695–710. ACM
          Press, 2015. *(p. 7)*.

[MRS08]   Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to
          Information Retrieval*. Cambridge University Press, 2008. *(pp. 4, 30, 43, 150,
          161, 174)*.

[MSB15]   Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. Code, camera,
          action: How software developers document and share program knowledge using
          YouTube. In *Proceedings of ICPC 2015 (23rd IEEE International Conference on
          Program Comprehension)*, pages 104–114, 2015. *(p. 123)*.

[NAA09]   Kevin Kyung Nam, Mark S. Ackerman, and Lada A. Adamic. Questions in, knowl-
          edge in?: a study of naver's question answering community. In *Proceedings of CHI
          2009 (27th International Conference on Human factors in computing systems)*,
          pages 779–788. ACM, 2009. *(p. 15)*.

[Opp92]   A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*.
          Pinter Publishers, 1992. *(pp. 47, 50, 52, 147, 150, 154)*.

[PBD+14a] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and
          Michele Lanza. Mining stackoverflow to turn the IDE into a self-confident pro-
          gramming prompter. In *Proceedings of MSR 2014 (11th Working Conference on
          Mining Software Repositories)*, pages 102–111. ACM, 2014. *(p. 6)*.

[PBD+14b] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and
          Michele Lanza. Prompter: A self-confident recommender system. In *Proceedings
          of ICSME 2014 (30th International Conference on Software Maintenance and Evo-
          lution)*, pages 557–580. IEEE, 2014. *(p. 7)*.

[PBD+16] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and
          Michele Lanza. Prompter: Turning the IDE into a self-confident programming
          assistant. *Empirical Software Engineering*, 21(5):2190–2231, 2016. *(p. 6)*.

[PBL13a] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Leveraging crowd knowl-
          edge for software comprehension and development. In *Proceedings of CSMR 2013
          (17th European Conference on Software Maintenance and Reengineering)*, pages
          59–66, 2013. *(pp. 6, 7)*.

[PBL13b]   Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering), Tool Demo Track*, pages 1295–1298. IEEE, 2013. *(p. 7)*.

[PBM+16a]   Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. Too long; didn't watch! extracting relevant fragments from software development video tutorials. In *Proceedings of ICSE 2016 (38th International Conference on Software Engineering)*, pages 261–272. ACM Press, 2016. *(pp. 7, 146)*.

[PBM+16b]   Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. Codetube: Extracting relevant fragments from software development video tutorials. In *Proceedings of ICSE 2016 (38th ACM/IEEE International Conference on Software Engineering)*, pages 645–648. ACM Press, 2016. *(p. 7)*.

[PC05]   Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*, pages 2–4, 2005. *(p. 169)*.

[PDO+13]   Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, pages 522–531, 2013. *(pp. 44, 137)*.

[PF08]   Nicola Perra and Santo Fortunato. Spectral centrality measures in complex networks. *Phys. Rev. E*, 78:036107, September 2008. *(p. 161)*.

[PMB+14]   Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. Improving Low Quality Stack Overflow Post Detection. In *Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution)*, pages 541–544. IEEE, 2014. *(p. 6)*.

[PMBL14]   Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. Understanding and Classifying the Quality of Technical Forum Questions. In *Proceedings of QSIC 2014 (14th International Conference on Quality Software)*, pages 343–352. IEEE CS Press, 2014. *(p. 6)*.

[PML15]   Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Summarizing complex development artifacts by mining heterogeneous data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, pages 401–405. ACM Press, 2015. *(p. 7)*.

[Pon12]   Luca Ponzanelli. Exploiting crowd knowledge in the ide. Master thesis, Faculty of Informatics, University of Lugano, June 2012. *(p. 31)*.

[PRM15]   Gayane Petrosyan, Martin P. Robillard, and Renato De Mori. Discovering information explaining api types using text classification. In *Proceedings of ICSE 2015 (37th ACM/IEEE International Conference on Software Engineering)*, pages 869–879, 2015. *(p. 12)*.

[PSB⁺17] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Supporting software developers with a holistic recommender system. In *Proceedings of ICSE 2017 (39th ACM/IEEE International Conference on Software Engineering)*. to be published, 2017. *(p. 7)*.

[R C14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. *(p. 53)*.

[RC15] Martin P. Robillard and Yam B. Chhetri. Recommending reference api documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015. *(p. 12)*.

[RMM14a] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE Transaction on Software Engineering*, 40(4):366–380, 2014. *(pp. 4, 159)*.

[RMM⁺14b] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the ICSE 2014 (36th International Conference on Software Engineering)*, pages 390–401. ACM, 2014. *(p. 159)*.

[Rob04] Stephen Robertson. Understanding inverse document frequency: On theoretical arguments for IDF. *Journal of Documentation*, 60:2004, 2004. *(p. 41)*.

[RR13] Peter C. Rigby and Martin P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, pages 832–841. IEEE Press, 2013. *(pp. 13, 17, 98, 102, 104)*.

[RRSK10] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. Springer, 1st edition, 2010. *(p. 12)*.

[RWZ10] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010. *(pp. 4, 12, 22, 23)*.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 625–56, 1948. *(p. 41)*.

[She07] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (4th edition)*. Chapman & All, 2007. *(pp. 53, 178)*.

[SIH14] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. In *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*, pages 643–652. ACM, 2014. *(p. 16)*.

[SM06] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding API components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 195–202. IEEE Computer Society, 2006. *(p. 14)*.

[SM11] Nicholas Sawadsky and Gail C. Murphy. Fishtail: From task context to source code examples. In *Proceedings of TOPI 2011 (1st Workshop on Developing Tools As Plug-ins)*, pages 48–51. ACM, 2011. *(p. 14)*.

[SMJ13] Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. Reverb: Recommending code-related web pages. In *Proceedings of the ICSE 2013 (35th International Conference on Software Engineering)*, pages 812–821. IEEE Press, 2013. *(p. 15)*.

[Smu26] Jan Christiaan Smuts. *Holism and evolution*. The Macmillan company, 1926. *(p. 5)*.

[SSC+14] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. The (r) evolution of social media in software engineering. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 100–116. ACM, 2014. *(p. 15)*.

[SSE15] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian G. Elbaum. How developers search for code: a case study. In *Proceedings of ESEC/FSE 2015 (10th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering)*, pages 191–201, 2015. *(p. 169)*.

[SSS67] E. A. Smith, R. J. Senter, and Air Force Aerospace Medical Research Laboratory (U. S.). *Automated Readability Index*. AMRL-TR-66-220. Aerospace Medical Research Laboratories, 1967. *(pp. 73, 115)*.

[STvDC10] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FoSER 2010 (FSE/SDP Workshop on Future of Software Engineering Research)*, FoSER '10, pages 359–364, New York, NY, USA, 2010. ACM. *(p. 15)*.

[TBS11] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web? (nier track). In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, pages 804–807. ACM, 2011. *(p. 15)*.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of the International Symposium on Principles of Software Evolution*, pages 154–164. IEEE, 2000. *(p. 5)*.

[TR16] Christoph Treude and Martin P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of ICSE 2016 (38th International Conference on Software Engineering)*, pages 392–403. ACM, 2016. *(p. 16)*.

[TX07] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the ASE (22nd IEEE/ACM International Conference on Automated Software Engineering)*, pages 204–213, New York, NY, USA, 2007. ACM. *(p. 14)*.

[USL08] Medha Umarji, SusanElliott Sim, and Crista Lopes. Archetypal Internet-Scale source code searching. In Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP The International Federation for Information Processing*, pages 257–263. Springer US, 2008. *(pp. 3, 12, 14)*.

[VPDC14] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. Codes: mining source code descriptions from developers discussions. In *Proceedings of ICPC 2014 (22nd International Conference on Program Comprehension)*, pages 106–109. ACM, 2014. *(p. 60)*.

[VSDF14] Bogdan Vasilescu, Alexander Serebrenik, Premkumar T. Devanbu, and Vladimir Filkov. How social Q&A sites are changing knowledge sharing in open source software communities. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 342–354. ACM, 2014. *(pp. 12, 15)*.

[WF05] Ian Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann Publishers Inc., 2005. *(p. 75)*.

[WF11] Ian Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*. Morgan Kaufmann Publishers Inc., 2011. *(p. 137)*.

[WHJK13] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of ESEC/FSE 2013 (9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*, pages 455–465. ACM, 2013. *(p. 44)*.

[Wil01] Laurie Williams. Integrating pair programming into a software development process. In *Proceedings of CSEET 2001*, pages 27–36. IEEE, 2001. *(p. 36)*.

[WM05] Richard Wettel and Radu Marinescu. Archeology of code duplication: recovering duplication chains from small duplication fragments. In *Proceedings of SYNASC 2005*, pages 63–70, 2005. *(p. 43)*.

[WT04] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of ICPC 2004 (12th IEEE International Workshop on Program Comprehension)*, pages 194–203. IEEE, 2004. *(pp. 138, 141)*.

[WYT13] E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of ASE 2013 (28th IEEE/ACM International Conference on Automated Software Engineering)*, pages 562–567. IEEE Press, 2013. *(p. 16)*.

[ZBY12] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *Proceedings of the RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, pages 38–42. IEEE Press, 2012. *(p. 16)*.

[ZSG79] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon. *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference, 1979. *(p. 11)*.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572. IEEE CS Press, 2004. *(pp. 4, 13, 35)*.