Supporting a Developer's Discovery of Task-Relevant Information

by

Arthur de Sousa Marques

B. Computer Science, Universidade Federal de Campina Grande, 2013

M. Computer Science, Universidade Federal de Campina Grande, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF SCIENCE

(Computer Science)

The University of British Columbia

(Vancouver)

August 2022

Abstract

The information that a developer seeks to aid in the completion of a task typically exists across different kinds of natural language software artifacts. In the artifacts that a developer consults, only some portions of the text will be useful to a developer's current task. Locating just the portions of text useful to a given task can be time-consuming as the artifacts can include substantial text to peruse organized in different ways depending on the type of artifact. For example, it might be easier to locate information in artifacts with structured headings such as tutorials whereas artifacts consisting of developer conversations might need to be read in detail.

To aid developers in this activity, given the limited time they have to spend on any given task, researchers have proposed a range of techniques to automate the identification of relevant text. However, this prior work is generally constrained to one or only a few types of artifacts. Integrating artifact-specific approaches to allow developers to seamlessly search for information across the multitude of artifact types they find relevant to a task is challenging, if not impractical.

In this dissertation, we propose a set of generalizable techniques to aid developers in locating a portion of text that might be useful for a task. These techniques are based on semantic patterns that arise from the empirical analysis of the text relevant to a task in multiple kinds of artifacts, leading us to propose techniques that incorporate the semantics of words and sentences to automatically identify text likely relevant to a developer's task.

We evaluate the proposed techniques assessing the extent to which they identify text that developers deem relevant in different kinds of artifacts associated with Android development tasks. We then investigate how a tool that embeds the most promising semantic-based technique might assist developers while they per-

form a task. Results show that semantic-based techniques perform equivalently well across multiple artifact types and that a tool that automatically provides task-relevant text assists developers in effectively completing a software development task.

Glossary

Q&A question-and-answer

ERB Research Ethics Board

UBC University of British Columbia

DS$_{synthetic}$ Synthetic tasks dataset, comprising six synthetic tasks with annotations from 20 participants of text deemed as relevant in 20 associated artifacts with natural language text

DS$_{Android}$ Android tasks dataset, comprising 12,401 unique sentences annotated by three developers and originating from artifacts associated to 50 software tasks drawn from GitHub issues and Stack Overflow posts about Android development

DS$_{Python}$ Python tasks dataset, containing 28 natural language artifacts where 24 participants indicated text containing information that assisted them in writing a solution for three programming tasks involving well-known Python modules

SO Stack Overflow, a question and answer website for software developers

STDV Standard deviation

SDK software development kit

API application programming interface

NLP Natural Language Processing

IR Information Retrieval

LDA Latent Dirichlet Allocation

POS part-of-speech

ML Machine Learning

RNN Recurrent neural network

DL Deep Learning

API Application Programming Interface

WWW World Wide Web

BERT Bidirectional Encoder Representations from Transformers

TARTI Automatic Task-Relevant Text Identifier, our proof-of-concept semantic-based tool, which uses BERT to automatically identify and show text relevant to an input task in a given web page

Chapter 1

Introduction

When performing software tasks in large and complex software systems, software developers typically consult several different kinds of documents, or artifacts, that assist them in their work [131, 174]. For example, when incorporating a new software library needed for a new feature, a developer might consult official application programming interface (API) documents for the library [156, 181] or question-and-answer developer forums [143, 169].

Many of the artifacts that developers consult contain unstructured text; for the purposes of this thesis, we refer to artifacts with unstructured text as natural language artifacts. To utilize information in natural language artifacts, a developer must read the text to find the information that is relevant to the task being performed [30]. However, the sheer amount of information in these natural language artifacts may prevent a developer from comprehensively assessing what is useful to their task [136]. Within just one kind of artifact, API documentation, studies have shown that it can take 15 minutes or more of a developer's highly constrained time to identify information needed to perform certain task [62, 131]. Considering that a developer must typically consult multiple kinds of documents when performing a task, the time investment to find the needed information can be significant. A developer that fails to locate all, or most, of the information needed will have an incomplete or incorrect basis from which to perform a software task.

Finding information that assists a developer to complete a task can be a time-consuming and cognitively frustrating process [32, 156]. Therefore, we posit the

need for approaches that assist developers in locating information in the different natural language artifacts sought as part of a software task.

1.1 Scenario

To illustrate challenges in locating information useful for a task, let us consider an Android mail client application[1]. Figure 1.1 shows a task—in the form of a GitHub issue[2]—that indicates that app notifications are not working as expected in Android 7.0.

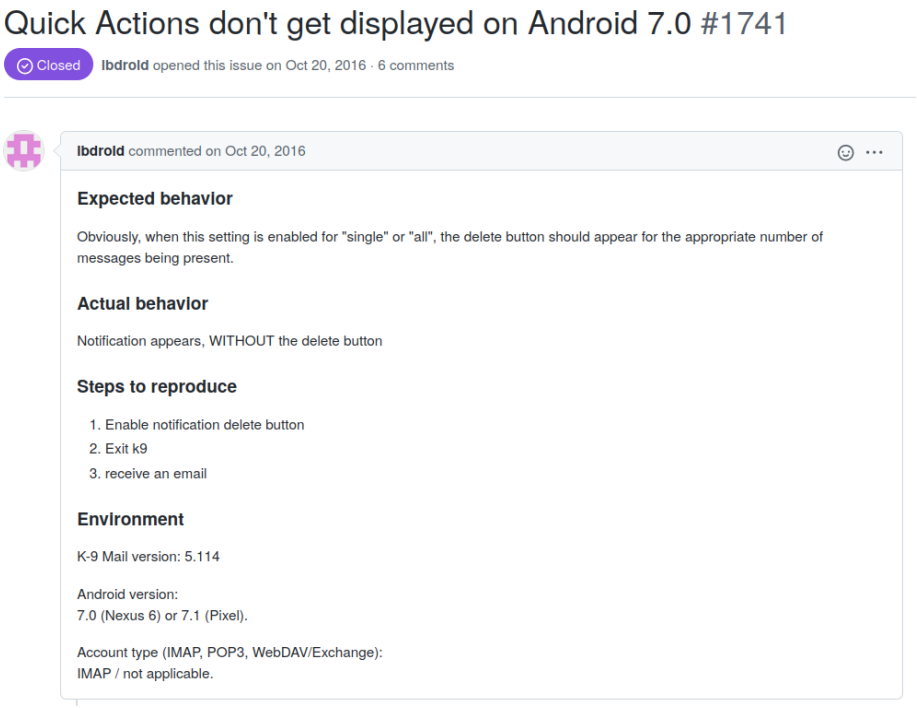

Figure 1.1: k-9 mail GitHub issue #1741 indicating that quick actions don't get displayed on Android 7.0

A developer assigned to this issue might not be familiar with how Android notifications work and thus, they will likely need additional knowledge to under-

[1] https://github.com/k9mail/k-9
[2] https://github.com/k9mail/k-9/issues/1741

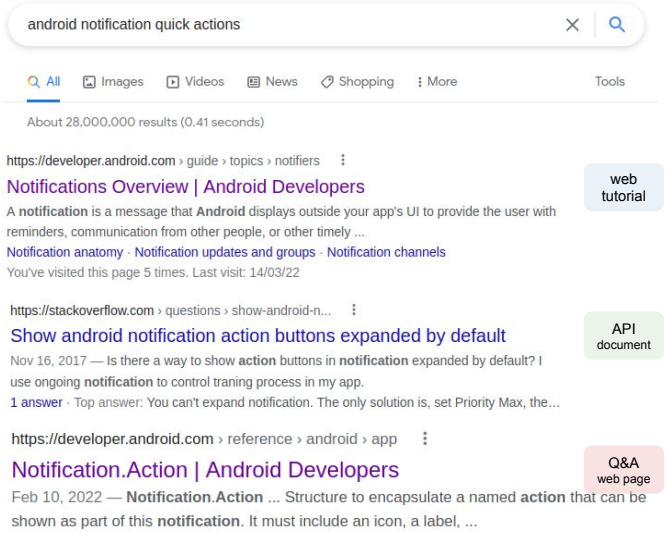

Figure 1.2: Search results showing artifacts of potential interest to the Android quick actions issue

stand and resolve this task [99, 106, 168]. Often, this knowledge can be acquired from a developer's peers [170]. However, the fragmented and distributed nature of software development may prevent the developer from accessing their peers [99], instead they seek online web resources for information that may assist them in completing the task-at-hand [153, 189].

A common way to find software artifacts pertinent to the developer's task is through a web search engine [38, 106]. Figure 1.2 shows the artifacts resulting from a developer's search about android notifications for the task in Figure 1.1. Each artifact can be assigned a type. For instance, the artifacts returned represent web tutorials, API documents and question-and-answer artifacts, each of which can be considered a different type. Each type of artifact has different associated challenges for locating information useful to a task within them.

The first artifact in Figure 1.2 is a web tutorial, a document intended primarily to teach users how to use a technology [20] through a series of structured topics that progressively explain concepts about the technology [85, 86]. Figure 1.3 shows a portion of this artifact's content. The Android tutorial contains approximately 200

Notification actions

Although it's not required, every notification should open an appropriate app activity when tapped. In addition to this default notification action, you can add action buttons that complete an app-related task from the notification (often without opening an activity), as shown in figure 9.

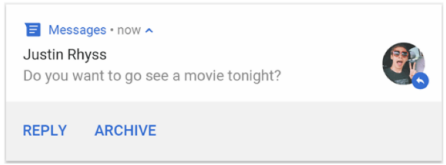

Figure 9. A notification with action buttons

Starting in Android 7.0 (API level 24), you can also add an action to reply to messages or enter other text directly from the notification.

Starting in Android 10 (API level 29), the platform can automatically generate action buttons with suggested intent-based actions.

Adding action buttons is explained further in Create a Notification.

Notification updates and groups

To avoid bombarding your users with multiple or redundant notifications when you have additional updates, you should consider updating an existing notification rather than issuing a new one, or consider using the inbox-style notification to show conversation updates.

However, if it's necessary to deliver multiple notifications, you should consider grouping those separate notifications into a group (available on Android 7.0 and higher). A notification group allows you to collapse multiple notifications into just one post in the notification drawer, with a summary. The user can then expand the notification to reveal the details for each individual notification.

The user can progressively expand the notification group and each notification within it for more details.

On this page

Figure 1.3: Snapshot of the official Android notifications tutorial with highlights relevant to the GitHub issue #1741

sentences and, on the page's right-hand side, we find that it has nine sections, each with sub-sections of their own. Reading all sections of this document could potentially take 10 minutes or more[3] of a developer's time. To save time, a developer would likely try to find the sections of the document most closely related to their task [106]. For example, using a web browser's search to find content that mentions the '*actions*' keyword, a developer would find eight different matches spread across four different sections. Not all of these matches will be relevant, requiring the developer to peruse each match and assess relevance. For instance, the '*Notification actions*' (under focus in the figure), explains notifications for both Android version 7.0 and 12.0, but given that the developer's task is related to the former version, only the text highlighted (in orange) might be of relevance to the task in Figure 1.1.

An API contains software elements (e.g., classes and methods) that other developers are expected to use for a certain purpose [135]. Instructions about an API usage are typically available in the API's reference documentation. Figure 1.4 shows part of the reference documentation for one of the classes of the Android API, namely '*Notification.Action*'. In the right-hand side of the figure, we find common information in this type of artifact, including a brief summary, constants and fields available, the class's constructor, and each of the functions or methods that this particular class provides, for a total of 35 distinct elements. To find the portions of the text relevant to their task, a developer unfamiliar with this document would face challenges similar to the ones already described in the web tutorial. Additionally, we note that complex APIs, such as the ones exposed by the Android operating system, require combining several classes and method calls [156] to properly perform some instruction, which would mean inspecting at least three other classes with equally complex documentation to find all the text explaining the API elements used in the quick actions menu[4].

The third artifact resulting from the developer's search about android notifications is a post in a question-and-answer (Q&A) web platform, Stack Overflow[5]. Figure 1.5 shows an example of the information found in this type of artifact. A

[3] Using a standard reading metric of 200 words per minute [90].

[4] https://github.com/k9mail/k-9/pull/1755/files

[5] https://stackoverflow.com/

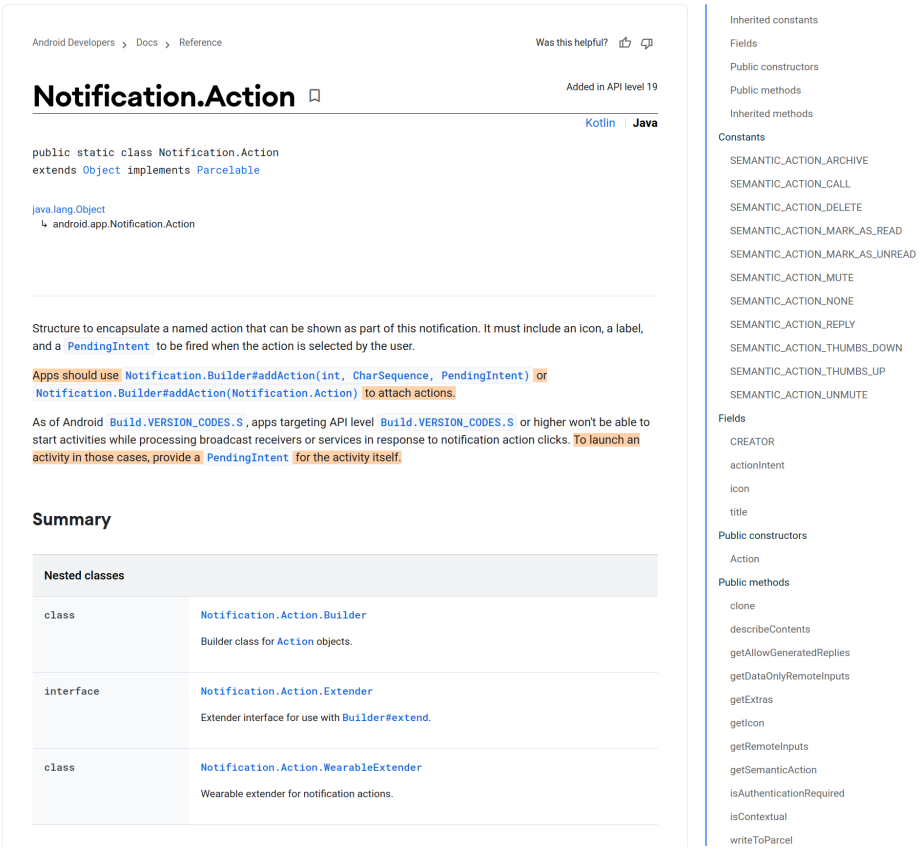

Figure 1.4: Snapshot of the Android Notification.Action API reference documentation with highlights relevant to the GitHub issue #1741

question usually contains both text and code snippets and it includes a set of tags about the problem's programming language or technology [180]. Each answer contains similar content and the counter that appears on left of a question and of each answer represents how many other users found them helpful (or not). The user who asked the question can also accept an answer (green check mark) if it correctly solved their problem. We also find a list of similar or related questions at the right portion of the page. This more structured format often assists a developer in navigating through the content in this type of artifact [137], for example, a developer could read the problem and then the accepted answer to quickly find information that might be helpful to their task. Nonetheless, a series of factors make finding

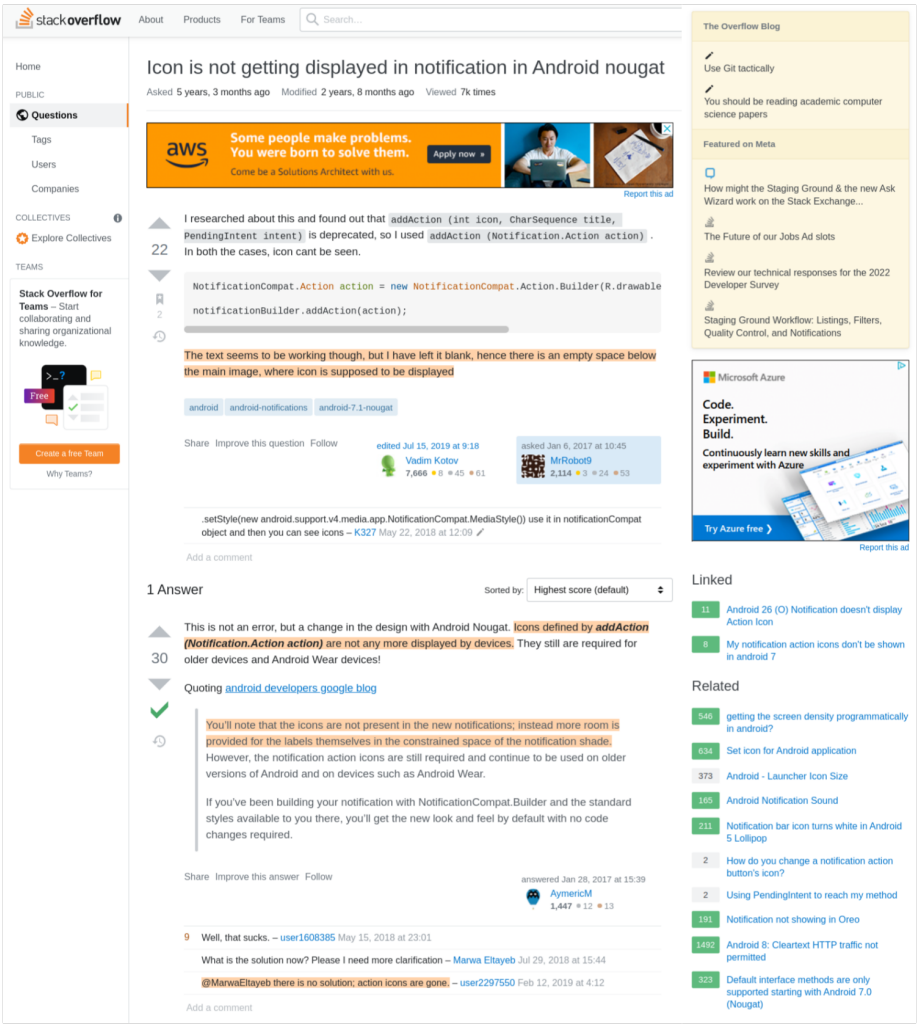

Figure 1.5: Snapshot of a Stack Overflow question about Android notifications with highlights relevant to the GitHub issue #1741

useful information challenging. For instance, only half of the Android questions on Stack Overflow have an accepted answer [143] and millions of questions have more than one answer [137]. Technologies also evolve and answers become obsolete [16]. Hence, despite the fact that structured data might assist a developer, finding task-relevant text in this type of artifact is also not trivial.

At this point, it is clear that if no tool support is provided, much of the process of locating text relevant to a task in a natural language artifact falls on the developer's shoulders [41, 77, 98]. Given how quickly developers progress to use new kinds of technology to record pertinent information, the aforementioned challenges are not exclusive to the three types of artifact that we have discussed, rather they are common to many kinds of natural language software artifacts [106, 174].

1.2 State of the art

To assist developers in discovering task-relevant information, software engineering researchers have proposed a number of techniques or tools to automate the identification of text. Although effective in specific contexts, this support is generally constrained to certain types of artifacts and integrating the many existing artifact-specific approaches to allow users to seamlessly search for information is challenging, if not impractical.

Artifact-specific approaches often use assumptions on the type of content and the meta-data available in an artifact to automate the identification of relevant text. Gaining insight into how developers produce and consume such artifacts to design artifact-specific techniques requires significant effort from the research community, where several empirical studies have investigated natural language artifacts and techniques using Information Retrieval (IR), Natural Language Processing (NLP), or Machine Learning (ML) to automatically identify relevant text [19, 97, 121, 142].

However, designing and evaluating such approaches does not follow the same pace with which developers have started to record pertinent information in a specific type of artifact [74]. By the time an artifact-specific tool might be available [75], it is possible that developers have progressed to using new kinds of technology, as observed in the recent shift from development mailing lists to instant communication platforms such as Slack [46, 112].

Contrasting emerging technologies, there are also long-lasting types of artifacts (e.g., API documents or bug reports) that a developer might consult to satisfy some information need. Nonetheless, due to the number of tools that aid developers in locating text relevant to different activities associated with these artifacts, it might be difficult for a developer to familiarize themselves with all of the existing tools and to decide which tool to use in a given situation.

These barriers might be lifted by a tool that could serve as a single point of entry to the automatic identification of relevant text regardless of the kind of artifact a developer consults. Such a tool could integrate the many existing tools (or their underlying techniques) and decide which one to use in which context. Nonetheless, due to an ever-growing the number of technologies used to record information, integrating such tools would be equally difficult. For example, in an approach that uses a bug report's meta-data [110, 119], a unified tool would have to consider how to extract such data from different products such as GitHub Issues[6] or Jira[7]. As another example, different programming languages might have very different documentation styles [62] and a unified tool would have to either integrate approaches tailored to each style or ensure that an approach for this kind of artifact consistently extract text from the myriad of styles under use [156].

A second approach for general techniques could consider ways to relate the text in a task to the text in a natural language artifact and state-of-the-art tools (e.g., [169, 190]) attempt to accomplish this using semantic techniques. These tools have similar constraints due to also using an artifact's meta-data and other limitations arise from how they make use of semantics, i.e., they default to a specific semantic approach [193] and do not investigate the range of semantic techniques [57, 67, 134] that might apply to text in software engineering artifacts.

These limitations suggest the need for a more generalizable technique that can evolve to identify relevant text in potential new kinds of artifacts and that is easier to deploy/install than many. If one technique could identify relevant text across all kinds of artifacts a developer encounters, the technique could apply in all situations and may be more adoptable in industry as a result.

[6] Microsoft GitHub
[7] Atlassian Jira

1.3 Thesis Statement

The scenario presented earlier in this chapter demonstrates the many kinds of artifacts a developer may consult to help complete a task and the challenges a developer faces in identifying text relevant to the task at hand within these artifacts. Existing techniques that support a developer in the automatic identification of relevant text are constrained to working on one or only a few types of artifacts. This thesis aims to overcome this constraint. We posit that:

> *A developer can effectively complete a software development task when automatically provided with text relevant to their task extracted from pertinent natural language artifacts by a generalizable technique.*

The design of a generalizable technique might be more possible if the text relevant to a task follows similar patterns across different types of artifacts [95]. Hence, we ask what are common properties, if any, in the text deemed relevant to a software task found across different types of artifacts? To investigate this question, we performed a formative study that examines text that twenty developers deemed relevant in artifacts of different types associated with six software tasks. By *characterizing* task-relevant text found in different kinds of artifacts, this study complements and adds to previous research that has examined text relevant to particular tasks and one kind of artifact [44, 97, 154, 155]. Notably, our analysis of natural language text in bug reports, API documents and Q&A websites inspected as part of this study show consistency in the meaning, or *semantics*, of the text deemed relevant to a task by a developer, suggesting that semantics might assist in the automatic identification of task-relevant text and in the design of a more generalizable technique.

Approaches that interpret the meaning of the text have been successfully used for a variety of development activities, such as for finding who should fix a bug [191], searching for comprehensive code examples [169], or assessing the quality of information available in bug reports [45]. Nonetheless, we are not aware of the usage of the meaning of text in techniques that assist developers in discovering task-relevant text over different types of natural language artifacts. To this end, this thesis describes the investigation of a design space of six possible techniques that

incorporate the semantics of words [57, 134] and sentences [67, 128] to automatically identify text likely relevant to a developer's task. An empirical assessment of these techniques on a corpus of tasks and artifacts reveals that semantic-based techniques achieve recall comparable to a state-of-the-art technique aimed at one type of artifact [190] while supporting multiple artifact types.

With our formative study, we found consistency in the text that is relevant to a task across different kinds of artifacts. With our empirical assessment, we found that a semantic-based techniques can automatically identify such text automatically across many artifact types. However, these studies do not address whether a semantic-based approach can *assist* a software developer while they work on a task. Hence, we introduce TARTI[8], a web browser plug-in that uses the most promising semantic-based technique identified in our empirical assessment to automatically identify and highlight text relevant to a developer's task. We evaluate TARTI with a controlled experiment that investigates the presence (or absence) of benefits that are brought by using TARTI. We report how participants performed two Python programming tasks when assisted or not by TARTI, where experimental results indicate that participants found the text automatically identified by the tool useful in two out of the three tasks of the experiment. Furthermore, tool support also led to more correct solutions in one of the tasks in the experiment, providing thus initial evidence on the role of semantic-based tools for supporting a developer's discovery of task-relevant information across different natural language artifacts.

1.4 Contributions

This thesis makes the following contributions to the field of software engineering:

- It details a formative study that characterizes task-relevant text across a variety of natural language artifacts, including API documentation, Q&A websites, and bug reports that are pertinent to six software tasks;
- It introduces six possible techniques that build upon approaches that interpret the meaning, or semantics, of text to automatically identify task-relevant text across different kinds of software artifacts, ~~where~~:

[8] Automatic Task-Relevant Text Identifier

- It shows how the most promising semantic-based approaches that we have explored have accuracy comparable to a state-of-the-art approach tailored to one kind of artifact [190], i.e., Stack Overflow.
- It presents an empirical experiment that provides initial evidence on how a semantic-based tool, TARTI, assists a software developer in completing a software task.

This dissertation also contributes with three different datasets (*DS*) that can be used for replication purposes and future research in the field:

- $\text{DS}_{synthetic}$ provides a unique corpus of 20 natural language artifacts that include annotations from 20 participants of text deemed relevant to the six tasks in our study on characterizing task-relevant text;
- $\text{DS}_{android}$ is a dataset with 50 Android tasks and associated natural language artifacts with annotations from three developers of the text relevant to these tasks;
- DS_{python} contains three Python tasks and it includes annotations from 24 participants that indicated which text assisted them in writing each task's solution.

1.5 Structure of the Thesis

In Chapter 2, we describe background information and previous approaches that seek to assist developers in finding useful information in natural language artifacts. The chapter details the empirical analysis of text in software artifacts, existing approaches and tools that assist in the automatic identification of text, and how these tools fit under the umbrella of studies that seek to improve a developer's work.

Chapter 3 presents our empirical study to characterize task-relevant text. We provide details on the tasks and artifacts that we have selected for this study and then, we present our findings on the text considered relevant, how we observe consistency on the semantics of the task-relevant text, and what are common challenges faced by developers trying to locate task-relevant text.

Chapter 4 describes the groundwork for producing the corpus ($DS_{android}$) that we use to evaluate the techniques detailed in the chapter that follows. It describes the selection of tasks, and artifacts pertinent to each task, as well as how three human annotators identified relevant text in each of the artifacts gathered.

Chapter 5 details the semantic-based techniques we investigate for automatically identifying task-relevant text. The first two techniques that the chapter presents use word embeddings to identify likely relevant text via semantic similarity and via a neural network. A third sentence-level technique filters (or not) the output of the word-level techniques according to frame semantics analysis. We combine these techniques for a total of six possible techniques, assessing the text that they automatically identify for the tasks and artifact types available in the $DS_{android}$ corpus, where we find that the neural network and the frame semantics techniques are the most promising ones for automatically identifying task-relevant text.

Chapter 6 details our empirical experiment investigating whether TARTI—a tool embedding a semantic-based technique—assists a software developer in locating information that helps them complete Python programming tasks. We begin by detailing experimental procedures and then, we report results from the experiment.

In Chapter 7, we discuss challenges and decisions made throughout our work, implications from our findings, and potential future work.

Chapter 8 concludes this work by reflecting on the contributions in the thesis.

Chapter 2

Related Work

The process through which an individual obtains knowledge often experiences paradigm shifts [149]. At times, an information need would be directed to a library clerk who would suggest books for the person's search [161]. Then, when the World Wide Web (WWW) [34] became popular, individuals started to rely on search engines to seek information now digitally stored in web pages [141]. Searching for pertinent artifacts, be them books or web pages, is one of the first steps to address an information need, which we detail in Section 2.1.

In possession of a potentially relevant artifact, careful inspection of its content might lead a person to find the information that they sought and researchers from several disciplines have considered means to help in this activity. In Section 2.2, we detail general approaches that mine information from natural language text. Then, in Section 2.3, we hone in on approaches that automatically identify text containing information likely relevant to an input software task.

Given how natural language artifacts have become intrinsically tied to software development [181], we conclude this chapter with an overview of other applications that make use of textual data to help developers across many of the tasks in their daily work (Section 2.4).

2.1 Finding Pertinent Artifacts

We start by considering how an individual finds artifacts, or documents, which might address an information need. For this, information foraging theory [149] explains how a person navigates through a search space looking for information patches based on a set of cues about the effort and gains that a patch provides to them [149].

Searches happen between patches (e.g., consulting different sources) or within a patch (e.g., over the many sections in a web page) and *explicit* or *implicit* factors affect judging the relevance of a patch [161]. For example, consider an electronic library catalog and a search for '*android notifications*'. Figure 2.1 shows two hypothetical results for this search. An explicit factor might represent how the keywords in the '*subject and genre*' field directly match one of the keywords used in the person's query. An implicit factor might represent a person's background or previous knowledge, such as how they might know that the first book targets a more general population and that it might not be helpful for a software developer.

1. **Android for Dummies**

   Author: Gookin, Dan

   Location: Britannia Branch Library

   LC Call No.: 004.167 A57G6ad

   Subject and genre: Android (electronic resource), Smartphone, Mobile computing

2. **Learning Android**

   Author: Gargenta, Marko

   Location: Champlain Heights Branch Library

   LC Call No.: 005.44 A5G2L1

   Subject and genre: Android (electronic resource), Application software (development), Mobile computing

Figure 2.1: Sample library catalog with two results associated with the keyword 'android'

A similar search could be performed on a web search engine[1] and for both library catalogs or web searches, researchers have investigated what affects how a person finds pertinent results and their decision to inspect them. For example, in electronic library catalogs, Hildreth observed how keyword searches were used more than any other type of search (e.g., by author or by title) [81] while other studies observed impatience and near-random search habits in novice librarians [140]. In contemporary web search engines, researchers have also investigated how individuals formulate queries [33, 78], how prior knowledge helps them to more efficiently perform searches [55], and what search results they inspect, going to the lengths of using eye-tracking technology for this purpose [52, 125].

With the prominence of web search engines, other search problems also gained attention. Most notably, Carbonell and Goldstein investigated the relevance and the novelty of search results [42]. That is, on the one hand, prioritizing relevance may lead to a scenario where all results contain redundant information, leaving some information needs unanswered. On the other hand, it might be difficult for an individual to find information that corroborates and consolidates some knowledge if search results are too diverse [48]. This has led researchers to investigated algorithms that balance the relevance and the novelty of the results retrieved by a web search [138, 152, 183] and commercial search engines have since considered how to strike a balance between relevance and novelty, providing diverse yet relevant results for a web search.

Across these general studies, and also in software engineering-specific studies [38, 55, 174], a common trend is that the identification of good search terms is as, or even more important than the search algorithm itself [93]. Nonetheless, identifying good search terms is often challenging and as a consequence, many searches are unsuccessful or retrieve resources that a person loses time inspecting only to find that they are not pertinent to the task at hand [79, 140]. This indicates that individuals usually inspect multiple and potentially different types of artifacts before finding relevant information, further motivating the work presented in this thesis.

[1] e.g., Google or Microsoft Bing

2.2 Techniques for Mining Unstructured Text

In this section, we provide background information on general approaches used by software engineering researchers to identify text in natural language software artifacts [30]. We focus on automated approaches that might assist a person in discovering text that addresses an information need, presenting both core concepts needed to understand an approach and seminal work that has used an approach in the context of natural language software artifacts.

2.2.1 Pattern Matching

Pattern matching approaches use regular expressions describing a sequence of tokens that represent the text to be identified [30]. We have shown how pattern matching identified books whose subject contained the keyword '*android*' (Figure 2.1) and the same principle could assist in finding parts of an artifact that contain some keyword, as many web browsers allow via a '*find in page*' feature (Figure 2.2). Nonetheless, this support is very limited and users would have to manually inspect each match to determine if they are relevant or not.

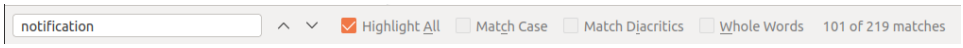

Figure 2.2: Find in page feature with results for the '*notification*' term

More automated approaches have also been built using pattern matching, as when Panichella et al. matched class elements mentioned in development mailing lists to source code files to assist developers in finding information useful to understanding the source code [142]. Although the heuristics and regular expressions used in this and other studies [121, 137] are lightweight and effective [30], pattern-matching approaches are often specific to certain kinds of domains and types of artifact [73], limiting their use in the design of a generalizable technique.

2.2.2 Information Retrieval

Information Retrieval (IR) comprises techniques or approaches for finding entities (documents, paragraphs, sentences, etc.) that satisfy an information need [123]. At its most basic level, IR uses the frequency or co-occurrence of words (or phrases)

to determine the relevance of an artifact with regards to an information need, often expressed as a query.

By counting how frequent is a word in an artifact and across the entire collection of artifacts, IR can be used to query which artifacts contain that word, or where in an artifact it appears. Using this principle, multiple schemes have been proposed to identify text based on how not all the words in some vocabulary have the same relevance (*TF-IDF*) [87, 120], how to represent sentences (*VSM*) [160], or how to account for different words that appear in a similar context (*LSI*) [59].

In natural language software artifacts, information retrieval has been used as part of approaches that automatically identify sentences that a developer would first read in a bug report when pressed with time [119], or in approaches that help cluster software components to aid program comprehension of a software system [126]. In this dissertation, we combine information retrieval and word embeddings to identify relevant text across different kinds of artifacts, as Chapter 5 further details.

2.2.3 Natural Language Processing

Natural Language Processing (NLP) relies on the lexical or syntactical analysis of the text [89] and such analyses might assist in the automatic identification of text that satisfies some information needs. To illustrate some of the elements obtainable using NLP, we consider a short sentence instructing how to perform a file system operation: "*you can use io.StringIO*". Figure 2.3 shows elements extracted for this sentence using two NLP techniques, namely part-of-speech (POS) tagging and dependency parsing. The formed assigns tags (*PRP*, *VP*, *NNP*, etc.) to each word in a sentence [178] while the latter identifies relationships between them, such as how '*io.StringIO*' is the direct object (*obj*) associated with the verb '*use*'.

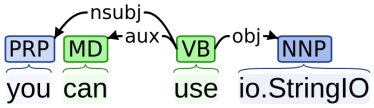

Figure 2.3: Syntactic elements for a sentence giving instructions about how to perform some file system operation

As an example of the application of NLP to software engineering text, Robillard

and Chhetri proposed a tool (Krec) that uses POS tagging to automatically identify text with threats and directives on how to use some API element [155]. As another example, DeMIBuD is an automatic approach that uses dependency parsing to automatically detect sentences discussing a bug's expected behaviour [44]. In this work, we use NLP as part of our analysis of the text deemed relevant to a set of software tasks (Chapter 3).

2.2.4 Machine Learning

Several other studies have investigated the text of various kinds of natural language artifacts and the meta-data available on them [19, 97, 121]. Findings arising from these empirical studies indicated regularities in the text and meta-data of an artifact which could be engineered into features that a Machine Learning (ML) technique could use to automatically identify or classify text that addresses certain information need [30].

Machine learning techniques can be categorized according to how they make use of available data. Supervised learning approaches use a set of features and labeled data to train classifiers for detecting entities of interest (e.g., text, images, etc.) while unsupervised learning approaches do not require labeled data and identify entities according to properties inferred from the data.

Supervised Learning

With regards to supervised learning approaches, we consider the many families of classifiers that determine *(1)* if some entity belongs or not to a given class, as in binary classification *(2)* which class, out of many, some input entity belongs to, as in multinomial classification, or *(3)* which classes can be assigned to a given input, as in multi-label classification [17], investigating their application to software engineering problems.

Among other applications, binary classifiers have been used to classify text in bug reports that describes (or not) steps to reproduce a bug [43] or to classify paragraphs which contain explanations about an API element in web tutorials [146]. In turn, multinomial classifiers have been mostly used to identify a category, out of many, that represents the type of information associated with some text. For in-

stance, Arya et al. identified 16 categories of information available in open source GitHub issues (e.g., workarounds, solution discussion, task progress, etc.) and they proposed a multinomial classifier to automatically identify such categories [19]. In a similar manner, Fu and colleagues consider five types of decisions that might arise during email exchanges (e.g., design, testing, management decisions, etc.) and they proposed a multinomial classifier to automatically identify all the decisions present in a development mailing list [72].

Regardless of the type of classifier, these approaches are typically built with training data containing human-engineered features and the cost and effort of hiring skilled workers to produce the labeled data for these and other supervised learning approaches has been a major limitation to their usage in software engineering research [18, 65].

Unsupervised Learning

Common applications of unsupervised learning to natural language software artifacts comprise text summarization [76] and data clustering, which might help an individual both in understanding the key information in an artifact and also in deciding which information patches are worth of their time [119].

Unsupervised summarization approaches are often based on variations of the PageRank algorithm [141], and these approaches identify the most central sentences in an artifact based on a set of relationships established between all the sentences in that artifact. Among other natural language artifacts, extractive summarization techniques have been applied to Stack Overflow posts [150], coding tutorials [109], or bug reports, as when Li et al. summarized key elements required to understand a bug and its resolution [110].

A second family of unsupervised techniques focuses on clustering data. These techniques use properties or features in the data to identify subsets with similar characteristics. Techniques that cluster data, such as Latent Dirichlet Allocation (LDA) [35], have been used by software engineering researchers to both bootstrap the categorization of information in natural language artifacts and as part of tools that identify topics in the text. For instance, Allamanis and Sutton applied LDA to gain insight into the types of questions asked on Stack Overflow [16] while FRAPT

is a tool that uses LDA to identity topics in a web tutorial, which are then used to identify key sentences in each of the topics identified [86].

Although unsupervised techniques lift the need for labeled training data, they still use human-engineered features, which are often specific to certain types of artifact. Therefore, both supervised and unsupervised ML have limitations that might prevent their usage in the design of a generalizable technique.

2.2.5 Deep Learning

In contrast to the human-engineered features, Deep Learning (DL) approaches allow the automatic extraction of features from training data through a set of mathematical transformations [56, 194]. This makes deep learning an interesting approach to uncover regularities that are not obvious or easily identified by software engineering researchers.

A common application of DL in software engineering is the usage of neural, or word, embeddings [134] for information retrieval purposes. Neural, or word, embeddings produce vector representations in a continuous space, where words with similar meanings are typically close in the vector space [80, 133]. Researchers have found that word embeddings mitigate lexical mismatches in the text found across different natural language software artifacts, as shown by Ye et al.'s evaluation of word embeddings for bug localization [193] or Huang and colleagues' study on the usage of word embeddings for API recommendation [83]. Guided by these findings, Chapter 5 describes how we use word embeddings in the identification of task-relevant text.

Many other DL studies in software engineering [65, 110] use neural network architectures for the same range of problems discussed in Section 2.2.4. As explained by Watson et al. [185], neural network architectures are composed of several layers that perform mathematical transformations on data passing through them. A set of parameters controls these transformations and adjust the model being trained so that it can predict the right outcome for any given input and researchers have proposed several architectures (e.g., RNNs [157, 176], encoder-decoders [21], Transformers [182], etc.) suitable for different problems.

As examples of architectures applied to software engineering artifacts, Li et

al. used an auto-encoder [114] to produce more accurate and diverse summaries for bug reports [110] while Fucci et al. used a recurrent neural network (RNN) to identify the types of information available in API documentation [73].

Similar to supervised ML approaches, deep learning approaches typically require large amounts of data for training purposes. Nonetheless, recent advancements on the usage of pre-trained models have lifted some of these limitations [63]. In Chapter 5, we investigate how we can use a state-of-the-art architecture, namely BERT [57], to identify relevant text across different types of artifacts.

2.3 Task-specific Techniques

Although certainly valuable, the techniques mentioned thus far do not extract text tailored to a specific software task. In other words, they identify a set of sentences in an input artifact regardless of the developer's task. In contrast, this section focus on task-specific techniques.

A small number [115, 169, 190] of studies consider specific types of artifacts and they formulate the problem of identifying task-specific text as a special case of query-based summarization [76]. That is, instead of producing a summary for the entire content of an artifact, these studies first use information retrieval techniques to identify a subset of sentences relevant to a task and then, a summary of these key sentences is produced.

AnswerBot [190] is an example of a query-based summarization tool. It uses word embeddings to find which sentences in a Stack Overflow answer are most similar to a query representing a developer's task. The tool uses semantic similarity in conjunction with Stack Overflow's meta-data to decide which text it should include in the summary that it will produce. A second query-based summarization tool by Liu et al. parses the content of API documents to build a knowledge graph of the information within this type of artifact. Their tool, KG-APISumm [115], uses this graph to find nodes with text that matches the text of an input task, producing a task-specific summary for this type of artifact.

However, these techniques target one type of artifact and they also use an artifact's meta-data to assist in the automatic identification of task-relevant text, limiting their use across the many different kinds of artifacts developers encounter daily

in their work. For example, AnswerBot uses the number of votes an answer has and if it is the accepted answer to decide which text it will use to produce a summary [190] while KG-APISumm uses code blocks, HTML anchor links and other properties which are exclusive to API documents to build the knowledge graph used as part of its summarization approach [115]. We differ from these studies by not using assumptions specific to certain kinds of artifacts and by considering how to identify task-relevant text in a more generalizable manner.

2.4 Improving Developers' Productivity

The tools and approaches presented in Sections 2.2 and 2.3 are examples of studies that help developers in locating text that might address an information need. These studies fit in the bigger context of software engineering research facilitating or improving the quality of a developer's work [92, 131, 164].

Software developers engage in many sensemaking and decision-making activities [168] and several studies have investigated how to provide means to better assist developers in performing such activities [27, 116, 117]. For instance, to assist a developer in deciding which web pages to inspect next, researchers have proposed tools that monitor a developer's search history and show how similar or not the results of a new web search are in comparison to already visited pages [151].

Other tools focus on assisting a developer make sense of the code that they might have to change as part of a task. For example, Deep Intellisense [82] displays code changes, filed bugs, and forum discussions mined from an organization's database so that a developer can understand the rationale behind the code. Other tools such as CueMeIn [175] use online resources for this same purpose, finding excerpts from web tutorials that might contain explanations for the classes and methods that a developer inspects.

Researchers have also been interested in making knowledge bases that developers working on a task can benefit from. Hipikat [51] is a seminal tool that exemplifies this concept in action. It takes a query explicitly prompted by a developer or implicitly based on the code that the developer has inspected and it recommends artifacts from a project's archive that are pertinent to the query. This archive, or project memory, is produced based either on relationships between the artifacts in

a software project or based on the files changed or accessed as part of a bug fix or feature request [51]. Other tools like Strata summarize knowledge produced by developers while they navigate on the web so that other developers can use this knowledge to have a head start when performing similar tasks [117].

This dissertation builds upon many of the research procedures outlined in these and many other studies in the field. For example, in the evaluation of Strata, Liu et al. describe procedures considering how a control and tool-assisted group complete information-seeking tasks [117], which assisted in the design of our experiment for evaluating an automated approach to task-relevant text identification (Chapter 6).

Chapter 3

Characterizing Task-relevant Text

In the last chapter, we described several studies that analyze text in software engineering artifacts and many approaches that attempt to automatically extract relevant information from these natural language artifacts. However, we have described that most of these approaches focus on one or only a few types of artifacts leaving open the question of how to identify text relevant to a task regardless of an artifact's type.

To address this questions, this chapter presents an *empirical study* in which we investigate whether there are common cues in the text that software developers identify as relevant to a task across different kinds of artifacts. To investigate such cues, we asked 20 participants with software development experience to identify relevant text within selected artifacts for six distinct software development tasks. We analyze the text that participants considered relevant to gain insight into properties of the text that are indicative of its relevance to a task [54, 89] while also reporting the participants' reasoning process for determining relevance, gathered through interviews that we use as part of our analysis on their information-foraging process.

We start by presenting the research questions that guide the design of our study (Section 3.1). We then detail experimental procedures (Section 3.2) and results (Section 3.3), concluding the chapter with a summary of our findings (Section 3.4).

3.1 Motivation

In Chapter 2 we described limitations of existing approaches for identifying task-relevant text across different types of artifacts. To investigate how to relax these limitations, we consider three questions:

RQ1 *How much agreement is there between participants about the text relevant to a task?* With this question, we seek to understand the amount of information sought for task completion and whether participants see the same text as relevant to a task.

RQ2 *What are common cues to the relevancy of text to a task?* With this question, we seek to determine if the rules governing how natural language information is constructed can guide us to information relevant to a task [95]. We consider if particular syntactic structures are cues to relevant text and whether there are patterns in the meaning of text identified as relevant.

RQ3 *How do developers determine if text is relevant to a task?* With this question, we seek to identify common themes in developers' identification of task-relevant text. For instance, what are common challenges or factors that affect a developers' judgment on the relevancy of the text to a task?

The first research question seeks to characterize text in documents relevant to a task. The second research question refers to the text itself. It analyzes possible syntactic or semantic predictive cues that may originate from commonalities in any of the text that developers deem relevant to the tasks we present them. The last research question considers qualitative aspects that might provide further insights to how developers identify relevant text.

3.2 Experiment

To investigate which text within a natural language artifact software developers deem as relevant to a task, we decide for a controlled experiment as it allows us to investigate the behavior of multiple developers over a consistent set of artifacts.

In the experiment, detailed in the following subsections, we asked 20 participants to highlight portions of a curated set of artifacts relevant to completing six

different development tasks; highlighting was followed by post-experiment interviews to understand the strategies that participants employed to identify relevant text, providing us both quantitive data (in the form of the highlights produced) and qualitative data (in the form of the interview transcripts) which we use to answer the research questions presented earlier in this chapter.

UBC ERB approved this experiment under the certificate *H18-02104*. The experiment's supplementary material is also publicly available [**?**].

3.2.1 Tasks

The six tasks in our experiment provide for a variety of information-seeking activities observed from earlier studies, which include when a developer refers to API documentation or Q&A websites for API usage purposes [156, 171, 181]; evaluates a possible reusable library to be incorporated into their implementation [181]; or confirms a system's behaviour referring to past discussions in community forums or in the system's documentation [119, 171, 181]. To design tasks for these activities, we applied criteria described by Petrosyan and colleagues [146], namely that tasks must encompass common application domains and that the tasks' artifacts must be publicly available.

Table 3.1 outlines the tasks designed. For each task, the table provides a brief description and the task name, which refers to an identifier pertaining the topic or the system related to the task. From this point forward, we will refer to the tasks according to these identifiers. `Bugzilla` and `Yargs` are API usage tasks; `Networking` and `Databases` ask participants to evaluate and decide upon the adoption of a certain technology or framework; and finally, `GPMDPU` and `Lucene` describe scenarios where a participant has to understand the system's behaviour.

Our decision to use a number of different systems employing a variety of technologies aims to minimize the effects that might result from a participant being well-acquainted with a particular system or technology [55, 186].

3.2.2 Artifacts

For each task, we also needed to choose a set of artifacts for a participant to peruse for relevant information to the task. Based on observations about the most common

Task	Description
Bugzilla	Locate information about Bugzilla's REST API custom fields and how they can be included as part of the `GET /rest/bug` payload
Databases	Review Q&A forums and decide between the adoption of ORM or JDBC for a system's database being migrated from C to Java
GPMDPU	Locate constraints or limitations about the GPMDPU shortcuts feature in order to work in a patch for this feature
Lucene	Review the Lucene documentation and bug reports to understand how it computes similarity scores during indexing, particularly the BM25 score function, such that you can address a change request
Networking	Review the MDN documentation and Q&A forums to decide between the adoption of Server-sent events or WebSockets technologies for a notification system being developed using Javascript
Yargs	Review the Yargs documentation and bug reports to check whether the API provides support for parsing mutually exclusive arguments, which is a requirement for a command line tool being developed in Python

Table 3.1: Tasks overview

information sources sought in development tasks [106, 151], we focused on API documentation, Q&A websites, and bug reports. For each task, a participant should consider on average three artifacts extracted from one or more information sources. We describe selection criteria for the artifacts and information sources as follows.

To select the artifacts, for each task, we performed searches and inspected the top 10 results retrieved by Google search engine, Stack Overflow, and in a system's bug tracking system to select a set of candidate artifacts that could assist in task completion. For some tasks, the list of candidate artifacts contained a single kind of artifact (e.g., bug reports). For others, multiple kinds of candidate artifacts applied (e.g., API documentation and Q&A websites). Due to the varied availability of artifact types per task, we ensured that across the six tasks, each kind of artifact appeared in at least half of the tasks.

To produce the curated set of artifacts for each task, we then carefully read each candidate artifact. Selection criteria considered an artifact's content and possible effects that could arise from interacting with a piece of text: (1) strengthening an assumption, (2) contradicting or eliminating an assumption, or (3) combining multiple pieces of text to yield a conclusion [48]. As examples of these criteria in action, the `Lucene` artifacts chosen contain complementary sources where sentences in different artifacts pertain to the same topic while the artifacts chosen for

Task	# participants	# Artifacts	API	Bugs	Q&A	# Sentences
Bugzilla	18	3	3	0	0	459
Databases	17	3	0	0	3	232
GPMDPU	18	3	0	3	0	291
Lucene	15	3	2	1	0	170
Networking	17	5	4	0	1	313
Yargs	18	3	1	1	1	409
Total	20	20	10	5	5	1874

Table 3.2: Corpus overview

the `GPMDPU` task contain contradictory information; two bug reports argue that a feature is not supported while a third states otherwise.

We also asked two other software engineering researchers to provide a list of candidate artifacts for each task based on the task's description. The Fleiss's Kappa coefficient score indicates a moderate agreement level ($\kappa = 0.45$) [68, 102] on the list of provisioned resources suggesting that a developer searching for pertinent artifacts for such tasks would likely obtain a similar list of artifacts for inspection.

Table 3.2 reports on the 20 artifacts used for the experiment: 10 are documents describing APIs, 5 are bug reports, and 5 are Q&A forum entries. These documents comprise nearly 1874 sentences and the average reading time [90] of all artifacts of each task is equivalent to 18.3 minutes ($\pm$5.5 minutes).

3.2.3 Participants

Twenty participants (2 self-identified as female and 18 as male) were recruited through mailing lists for computer science graduate students and through personal contacts. To ensure participants were familiar with concepts integral to the experiment, a demographic survey included questions about the use of software documentation and bug tracking systems by a participant. No participants were excluded based on answers to these questions. Participants were compensated ($20) for their participation. At the time of the experiment, 5 participants were working as software developers and 15 were graduate students, of which 73% have previous professional experience. On average, participants self-reported 8.9 years of programming experience ($\pm$ 4, ranging from 3 to 20 years) and 16 out of the 20

participants reported professional experience with an average of 4.7 years ($\pm$ 3.7, ranging from 1 to 15 years).

3.2.4 Procedures

Each experimental session lasted no more than two hours; this length of time was selected based on three pilot sessions. We describe the final experimental set-up, referring to changes made based on the pilot sessions.

Each session began with a short tutorial explaining the experiment's purpose and describing how to use a provided tool (in the form of a browser extension) to highlight text relevant to a given task. The concept of relevance was left to a participant's judgment as any definition or explanation could introduce bias. Each participant was able to practice use of the tool on an example that was separate from the experimental tasks and artifacts. We added this description of the tool and ability to practice with the tool after the first two pilot sessions.

Participants worked on the assigned tasks in a given order and were not required to complete all tasks. We randomized the order of presentation of each task across participants while also balancing the order of tasks [187], such that every task was first seen by an equal number of participants. Participants took as much time as they needed to complete a task.

For each task, a participant was presented with a description of the task in a PDF viewer. The description included links to the artifacts a participant was asked to consider as part of the task: when a link was followed, the artifact would appear in a browser window. A participant could choose to consider the artifacts in any order and was asked to highlight any text the participant considered relevant to complete the task.

When the participant indicated completion of the task, a short questionnaire was provided, specific for the task, that gathered background information about a participant's knowledge related to the task and whether they had likely gained appropriate knowledge of how to complete the task. As an example, Figure 3.1 provides an excerpt of the questionnaire presented for the `Networking` task.

Each session concluded with a debrief session where we discussed the correctness of every task, and also conducted a semi-structured interview that discussed

On a scale of 1 to 5, how familiar are you with the task's technologies:
(not at all familiar) 1 - 2 - 3 - 4 - 5 *(extremely familiar)*

On a scale of 1 to 5, what was the level of difficulty of the task:
(very easy) 1 - 2 - 3 - 4 - 5 *(very hard)*

Please evaluate the following sentences and mark only correct statements:
☐ SSE are bidirectional and it can be used for pushing notifications to the bill-sharing system;
☐ SSE works over HTTP with no additional components;
☐ WebSockets are bidirectional and it can be used for pushing notifications to the bill-sharing system;
☐ There are no data type limitations for SSE messages;
☐ There are no data type limitations for WebSockets messages;

☐ WebSockets should be adopted for the notification system;
☐ SSE should be adopted for the notification system;

☐ After reviewing the documentation, I don't have enough knowledge to complete this task

Figure 3.1: Questionnaire for the Networking task asking about previous knowledge, expertise and likely solutions for the task

how a participant reasoned about relevance and what features of the text, in their opinion, helped in deciding whether information in the text was relevant or not. To ensure that the session would finish in the allotted time, we shortened the debriefing and interview components for certain participants (e.g., as when a participant finished 4 tasks near the allotted time) such that the study would not take longer than the stipulated 2 hours time period. Overall, 11 out of the 20 participants completed all the tasks. Table 3.2 discriminates the number of participants who completed a task.

3.2.5 Summary of Procedures

We have described experimental procedures that allow us to inspect the text that 20 participants deemed relevant to different kinds of artifacts associated with six information-seeking tasks. Figure 3.2 illustrates the data gathered with this experiment. It shows a portion of a task description used in the experiment (left-side) and it provides an overview of highlights (right-side) selected by participants using our

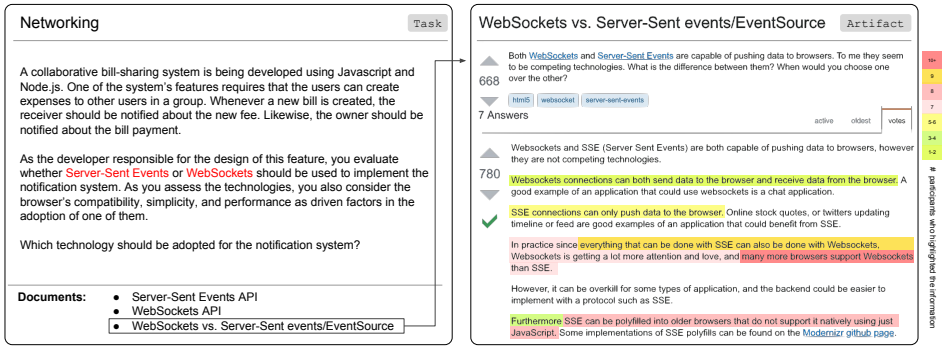

Figure 3.2: An example of a task description and a depiction of collected data, shown as a heatmap of text that participants highlighted as relevant to the task

highlighting tool.

3.3 Results

In this section, we present experimental results. First, we describe the data produced and then, we analyze properties of the text considered relevant. We also report results from our interview analysis.

3.3.1 Relevant Text in Natural Language Artifacts

To characterize the task-relevant text found in natural language software artifacts (*RQ1*), we analyze the participants' produced highlights. We focus our analysis on highlights participants made of natural language text. We do not consider highlights participants made of source code snippets or in tables, leaving the consideration of this information to future work.

We define a **H**ighlighted **U**nit (HU) as the full sentence containing any highlight by a participant. We use sentences as the unit of analysis because this was the most common unit considered by the participants. That is, of the 2,463 distinct highlights created by participants, 1,777 are sentences (72%), 621 are portions of sentences (25%) and 65 are combinations of consecutive sentences (3%). Thus, if a participant highlighted just a phrase in a sentence, we consider the full sentence as an HU or if a participant highlighted more than one portion of a sentence we

still consider the full sentence as one HU.

How much text in an artifact is deemed relevant to a task?

We first ask how much text within an artifact participants found relevant to a task since if almost all of the text is relevant then there would not be a need for identifying text within an artifact. Figure 3.3 presents the ratio of relevant and non-relevant text in each type of document. We compute the average ratio of HUs and text for each artifact from each type of document. We report the mean of the ratio artifact-wise to prevent misinterpretations due to outliers, i.e., a lengthy document with few HUs or a document concentrating almost all the HUs

Considering all HUs, between 1% to 20% of an artifact's text is considered relevant, depending on its type. API documents have the smallest ratio of relevant information (mean of 3%) and for this kind of artifact, at most 6% of all sentences were considered relevant. This result is not surprising as API documents may describe many API features or methods of which only a few are likely to apply to a particular task [156]. Similarly, only a small portion (4% on average) of bug reports are considered relevant. This result is also not surprising as bug reports can contain long discussions encompassing several topics [39, 154]. Even the kind of document with the highest percentage of highlights, Q&A entries with, on average, 11% of the sentences being relevant, contain a substantial amount of information not relevant to a specific task.

> *Finding task-relevant information in a bug report, API document and Q&A documents require filtering to less than a 20% of the documents' text.*

How much agreement is there between participants about the text relevant to a task?

With this question, we seek to identify if there is a set of key sentences that participants unanimously consider relevant to a task. We use Nenkova and Passonneau's pyramid method [139] to investigate the degree to which participants agree upon the text relevant to a task. This method was originally used to quantify the content of summaries produced by different annotators. The more annotators who include the same content in their summaries, the more consistent the view of information

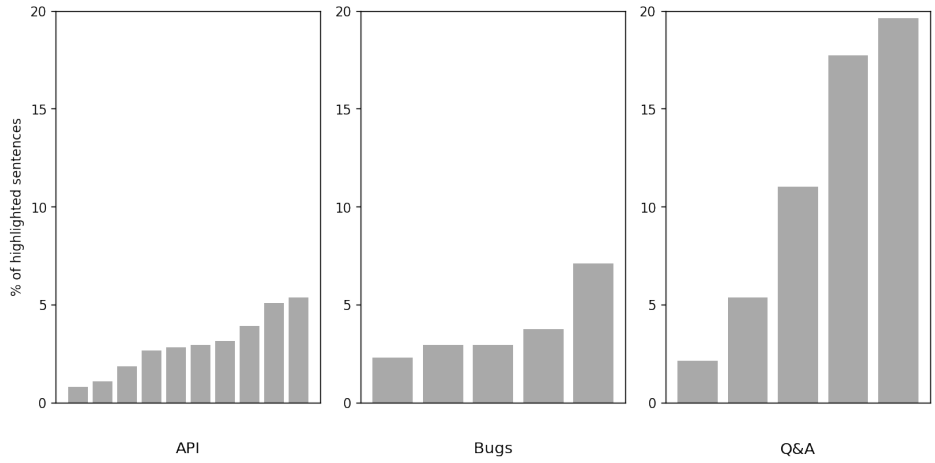

Figure 3.3: Percentage of highlighted sentences per artifact grouped by artifact's type

relevant to a summary and the more weight given the selected content. We apply the same rationale to assess the level of agreement of which HUs are relevant to a task and whether agreement relates to how correctly a participant completes a task.

Figure 3.4 shows the distribution of HUs over the pyramid we produce. To facilitate interpreting the results, we take into consideration how other studies provide categories for the relevance of text (i.e., [146] and [86]) and we aggregate HUs selected by a range of participants into three tiers. Table 3.3 presents HUs per tier. Starting from the bottom tier, the pyramid represents the perceived relevance of information from less to more relevant. With this information, we test if the number of HUs identified at each tier affects how correct were the solutions provided by the participants (as measured by the questionnaires applied for each task).

Since have three independent variables (i.e., tiers) and one outcome variable (i.e., scores), we use a multivariate analysis of variance to test this hypothesis [187] Results indicate a significant differences (p-value < 0.01) between participants' score and their HUs. We then conducted univariate tests on each tier, finding that the number of HUs identified at the mid and top-tiers positively affect participants scores, what suggests that HUs from these two tiers contain key information for completing a task. As expected, HUs at the bottom tier vary more per participant.

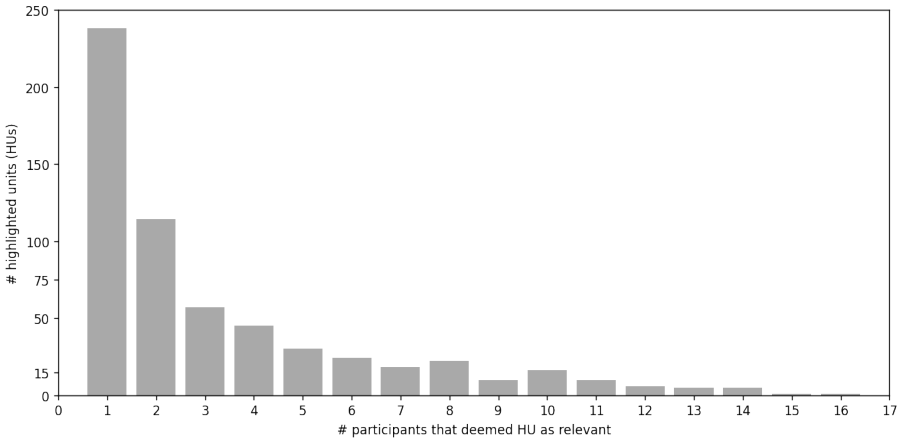

Figure 3.4: Distribution of the number of participants who deem an HU as relevant

Task	# HUs	bottom tier†		mid tier†		top tier†	
Gpmdpu	43	23	(53%)	10	(23%)	10	(23%)
Bugzilla	103	51	(50%)	22	(21%)	30	(29%)
Yargs	74	59	(80%)	12	(16%)	3	(4%)
Lucene	95	54	(57%)	24	(25%)	17	(18%)
Databases	159	84	(53%)	41	(26%)	34	(21%)
Networking	128	81	(63%)	37	(29%)	10	(8%)
Total	602	352	58%	146	24%	104	17%

† The bottom tier includes HUs highlighted by 0–2 participants the mid tier includes HUs highlighted by 3–7 participants, and the top tier includes HUs highlight by at least 8 participants.

Table 3.3: Distribution of highlights and their respective percentages per tier for every task

Due to such variability, there is no clear indication that bottom tier HUs are from participants from a certain group, such as all those unfamiliar with a particular technology or who successfully completed a task and those who did not.

> *Text perceived as relevant by more participants likely relates to key information for completing a task whereas information highlighted by a few participants may be more dependent on the knowledge of individuals.*

3.3.2 Textual Analysis

We examine syntactic and semantic properties of the highlighted text so that we might identify common cues to the relevancy of text to a task (*RQ2*). Since we intend to use these cues in the design general technique, we examine the text across all the kinds of artifacts available for all the tasks.

Does syntactic structure provide cues to the relevancy of text to a task?

For our syntactic analysis, we follow procedures from related work (Section **??**). We start by inspecting the elements that compose a sentence (i.e., nouns and verbs) and then, we analyze possible patterns that may arise from the syntactic structure of the text [155], investigating if the extracted entities co-occur in multiple HUs.

Among noun phrases, we observe that 65% of the HUs contain acronyms or coding elements. Existing approaches that rely on these elements to identify relevant text (e.g., [155] or [85]) would miss the remaining 35% of the HUs in our corpus. This value may seem acceptable at first; however there are no guarantees that the identified 65% HUs hold all the crucial information for task completion. As an example, some of the HUs from the mid and topmost tiers do not contain obvious code elements that could signal their relevancy, such as one of the sentences in the `Bugzilla` task indicates the need for "*authentication to allow retrieving non-public data*".

With regards to verb phrases, we observe a substantial overlap (81%) with verbs observed in Ko and colleagues linguistic analysis of bug report titles [97]. The most common verbs in the HUs include conjugations of verbs such as *use*, *get*, *set*, *be*, or *do*, but with the exception of *use*, *get*, and *set*, the remaining top common verbs are in English stop words lists [89]. As a result, many NLP techniques would discard them as part of their pre-processing steps [30].

As for syntactic patterns, we did not observe a large set of patterns for the variety of tasks and artifacts in our experiment, where the prominent patterns identified (e.g., {*nsubject, do, negation*}) reflect common constructs of the English language rather than cues that we might explore for the relevancy of text. There may be multiple explanations for these results, such as the fact our corpus contains a small number of natural language artifacts. Due to this reason, we also checked whether

patterns from existing related work (i.e., [44, 155]) applied to the text in our HUs, but the small number of matching patters raises caution on their generalizability.

> *We did not find prominent syntactic cues to identify task-relevant information. Our analysis of highlights demonstrates: 1) limitations of existing techniques that rely on code elements and acronyms, 2) missed information that may occur due to the prevalence of verbs that appear in English stop word lists, and 3) the absence of patterns derived from the syntactic structure of the text.*

Do semantics provide cues to the relevancy of text to a task?

For our semantic analysis, we analyze the meaning of the text in the HUs using frame semantic parsing [67, 89]. Semantic frames are centered around events, labelled frames, which abstract both the event as well as relationships, entities or participants related to that event [22, 67].

We explain semantic frames by considering two distinct sentences extracted from the `Databases` and the `Lucence` tasks, respectively. For each sentence, Figure 3.5 shows an excerpt of the frame analysis for the sentences. The frames of each sentence (in grey) represent a triggering event and the frame elements *(fe)* (in red) are arguments needed to understand the event. The enclosing square brackets mark all lexical units, or words, associated with either a frame or a frame element. Observing the frame elements captured by the verbs '*see*' and '*understand*', both sentences have the common meaning of describing a '*phenomenon*'. However, the frame elements that capture the meaning of each verb differ: the former represents a '*perception of experience*' while the latter represents a cognizer '*grasp*'ing her knowledge over the phenomenon.

As multiple sentences might have similar meanings, we analyze whether there are common frame elements that provide cues to the relevancy of text. For this analysis, all frame elements were extracted automatically using the *SEMAFOR* toolkit [54], where we extract the frames of every HU, resulting in 3,719 frames across the 602 HUs. Only 346 distinct frames appear across these 3,719 frames parsed. The proportionally small number of distinct semantic frames occurring suggests that different HUs share frame elements.

Table 3.4 details the most frequent frames identified per tier, filtering to show

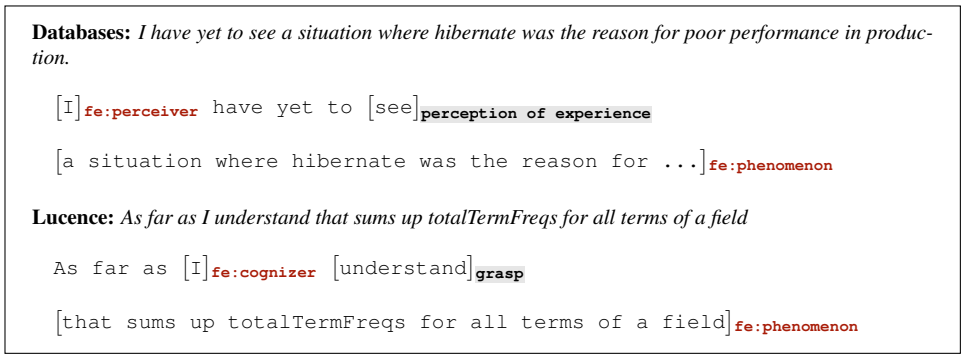

Figure 3.5: Example of frames and frame elements

the most frequent frames in a tier that have not appeared in lower tiers. In the topmost tier, the most frequently identifying distinguishing frames denote the '*cause*' or '*likelihood*' of a phenomenon. These frames are found in sentences that explain a system's behavior, which are often crucial for task completion, as in a sentence that provides a cause for the loss in performance when using Hibernate: *"if you need to process lots of objects for some reason, though it can seriously affect performance*". Other common frames *quantify relationships*, as when a sentence describes the minimum elements needed to perform an operation, e.g., *"to create a flag, at least the status and the type_id or name must be provided*".

In the middle tier, we observe frames for actions performed by some entity (*intentionally act*) or facts regarding a topic (*statement*). Other common frames relate to methods or attributes and the result of some operation such as a method call, which may be useful for identifying code related entities. For instance, this sentence from the `Bugzilla` tasks contains both the '*being returned*' and the '*fields*' frame, *"You can specify to only return custom fields by specifying _custom or the field name in include_fields"*.

The bottom tier contains frames that are common to all HUs. The most frequent frame in the bottom tier has a semantic meaning of '*using*'. HUs with this frame are often sentences detailing how to use a method or a framework to achieve some goal, what might also explain the second most frequently occurring frame, i.e., '*purpose*', which denotes an achievable goal. These two frames could be used to filter sentences that contain the means to use a technology or API with certain

	Frame	Freq	Description	Example
top tier	Likelihood	8%	Denotes the likelihood of a hypothetical event occurring	However, it can be overkill for some types of application, and the backend could be easier to implement with a protocol such as SSE.
	Causation	8%	An effect is observable due to a cause	By default this is true, meaning overlap tokens do not count when computing norms.
	Relational Quantity	7%	Denotes a quantifiable relationship between any two dependent entities	It is much faster to get to something working with Hibernate than it is with JDBC
mid tier	Statement	12%	Verbs and nouns that communicate the act of a speaker to address a message	Custom fields are normally returned by default unless this is added to exclude_fields
	Intentionally act	11%	An act performed by an entity	The data field could, of course, have any string data; it doesn't have to be JSON.
	Fields†	11%	Denotes mentioning an object attribute or field	computeNorm(FieldInvert state) - computes the normalization value for a field
	Being returned†	11%	Denotes results from a particular operation or method call	You need to be aware of this behaviour otherwise you will get cryptic errors
bottom tier	Using	17%	An agent uses an instrument in order to achieve a purpose	By using JDBC, resource leaks and data inconsistency happens as work is done by the developer.
	Purpose	15%	Denotes a goal or target to be achieved	Object Relational Mapping empowers the use of ''Rich Domain Object'' which are Java classes
	Capability	14%	An entity does or does not meet the preconditions for some event or action	Can't detect anything outside letters, arrows, ctrl, alt and shift

† Frame name was modified because its semantic meaning is specific to software engineering;

Table 3.4: Most common semantic meanings across all the HUs; frames are presented per tier, from the topmost tier to the bottom tier

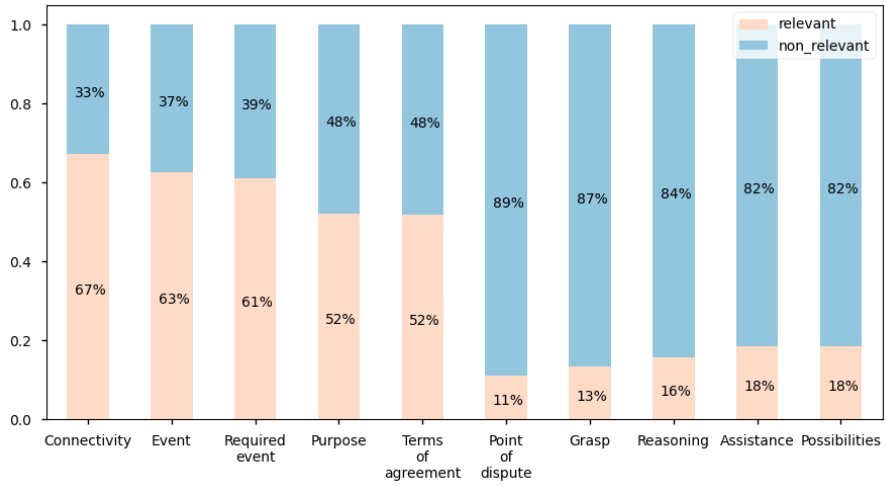

Figure 3.6: Distribuion of semantic frames over the text; the figure depicts the top five frames most commonly observed in relevant and non-relevant sentences, respectively

intention, as this sentence explaining usage scenarios for WebSocket and Server-Sent Events in the `Networking` task: "*One is synchronous and could/would be used for near real-time data xfer, whereas the other is asynchronous and would serve an entirely different purpose*".

To provide further evidence supporting these findings, we also compared the frequency of the frame elements identified in relevant text (i.e., HUs) and non-relevant text. Figure 3.6 provides insight in the distribution of frames across relevant and non-relevant sentences. For example, frames that represent a '*required*' event are more prominent in the relevant text as found in a sentence that describes how to circumvent errors in the `Bugzilla` REST API due to invalid tokens: "*An error is thrown if you pass an invalid token; you will need to log in again to get a new token*". On the other hand, frames that relate to user discussions and that do not draw conclusions or provide facts about an API or technology, such as '*point of dispute*' or '*reasoning*' are often found in non-relevant text, as when users discuss community's procedures in the `GPMDPU` task: "*Open a new issue following the template so we can have more details on your device*".

While certain frames are not indicative of relevance when found on their own,

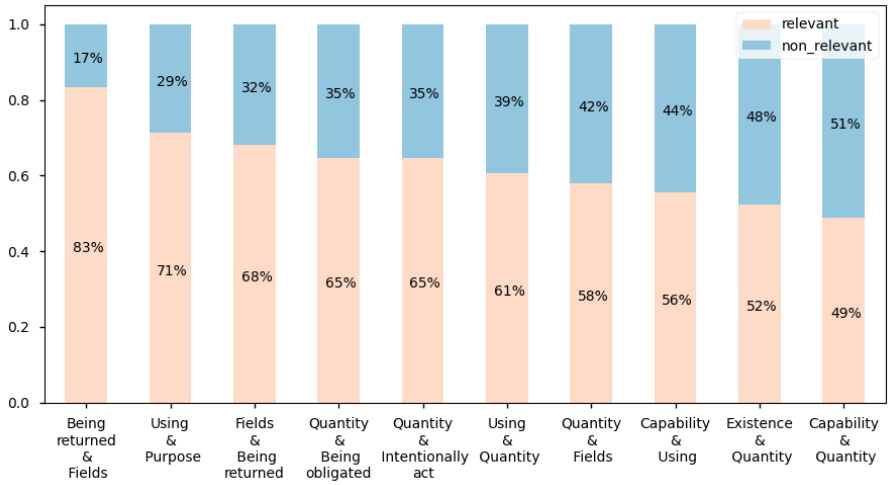

Figure 3.7: Distribution of co-occurring frames over the text

we also observe that the co-occurrence of certain frames in a sentence increase the likelihood of the sentence's relevancy. For instance, '*purpose*' and '*using*' occur almost evenly across relevant and non-relevant text while their co-occurrence is more frequent in relevant text. Figure 3.7 shows the most frequent frames that co-occur and their ratio on relevant and non-relevant text.

A Wilcoxon signed-rank test [187] over the distribution of frames in our data both in individual frames and pairs of frames shows statistical significance (p-value $\leq$ 0.05) on the prominence of certain frames in relevant or non-relevant sentences.

Our analysis on the semantics of relevant text indicates that there exists some common aspect to the different sentences deemed as relevant.

> *There are recurring semantic meanings in relevant sentences, suggesting commonalities in their conveyed information and indicating that text might be identified through its semantics.*

3.3.3 Interview Analysis

To understand how participants identified relevant text (*RQ3*), we analyzed the participants' interview responses using a card-sorting approach [173]. We created index cards containing the interview questions and the participants' responses.

The cards were sorted into meaningful themes and then grouped into abstract categories. We refined themes over three iterations of the data. At each iteration, we grouped cards containing responses on similar topics, analyzed the emerging theme, and evaluated whether there was a broader theme that incorporated two or more of the existing themes. To reduce individual bias, the first and second authors independently annotated a subset of the cards at every iteration. Disagreements occurred when more than one theme could apply to a sentence. The annotators discussed and resolved disagreements refining the set of themes in a subsequent iteration, where annotators had substantial agreement ($\kappa = 0.82$).

From participants' comments, we identified nine themes that we group under two major categories. Table 3.5 details observed categories, themes, and the number of participants who made a comment pertaining to that theme. We discuss results per category illustrating comments that best exemplify a theme (grey highlight).

How do developers locate relevant information?

Developers often use a mix of *keyword-matching* and *skimming* [98, 174] to find relevant information within an artifact. Some participants said they use these search strategies for all types of artifacts, while others said they use knowledge of a document's structure to assist their searches (*document-guided*).

> "*Definitively [my strategy] wasn't always the same. Going through a StackOverflow question, I would obviously read the first response. For API docs, keywords were the go to. Bug reports are hard. Comments are ordered chronologically and the first ones are sometimes not the most relevant ones*"—P12

Participants were also aware of the shortcomings of some artifact types and were less willingly to use faster but less accurate strategies like skimming or keyword-matching for these artifacts when the tasks were difficult. In these cases, participants mentioned that they used a *scrutnizing* strategy:

> "*I usually read every comment [in Stackoverflow]. Obviously, they are ranked, but, in general, every single comment could have something important*"—P11

Regardless of strategy, participants mentioned using implicit criteria to decide when to (not) read some text. Such implicit criteria often relate to information

	Theme	Definition	#
Searching Strategy	document-guided	Knowledge about the structure of the document influences how a developer locates relevant text in that particular document, e.g., the fact that StackOverflow has accepted answers, or that bug reports might have status changes such as when a user closes a bug marking it as resolved	15
	keyword-matching	The use of keywords and search filters to locate relevant text	13
	skimming	The act of quickly reading a portion of a document to locate relevant text. It also represents the act of scrolling from a document's sections and deciding to stop at certain points	11
	scrutinizing	The act of carefully reading a document/section due to some reason	10
	summarizing	The desire to first have an overview of a document or section's content before deciding which portions are worth investigating	9
Challenges	missed-information	Acknowledgement that some information deemed relevant was missed due to the developer's search strategy	11
	familiarity	How familiar is a developer with the domain of the task and/or technology and how his expertise affects her foraging process	10
	statement-checking	The text contains facts that are not accurate and information might be ambiguous or contradictory	7
	verbosity	The structure of the text is verbose or extensive hindering readability	5

[#] Indicates number of participants who have quotes related to the theme.

Table 3.5: Key themes relating to developers' assessment of relevancy

foraging theory and how an individual follows some *information scent* for judging the value, or cost of investigating some text according to available cues [149], such as the presence of bullet-points, bold text, or the conciseness and readablilty of the text.

What challenges do developers face while searching for relevant information?

Participants also commented on factors that made assessing the relevancy of text difficult. The most common challenge was *missing information* that other participants deemed relevant.

> "*This [highlight] is a valid alternative if you don't want to use the conflicts [method]. I basically didn't see this because of how I was searching*"—P2

We consider missing relevant information as a major threat to task completion because it can lead to an incomplete or incorrect solution [136]; usually, participants missed information due to their workflows, as *P15* explains:

> "*It's hard to say that I would have picked [those highlights] without being directed to that. I believe that I knew that there were specific questions coming, but even if I was doing it for myself, I would probably skim first and then, when I start to code, it would be a natural thing to get back and dive into the details*"—P15

Participants also missed information due to text *verbosity*. Bloated text makes it harder for developers to locate task-relevant information [156] and is more cognitively demanding to read:

> "*When I looked at the API documentation, there was too much text. I was mentally exhausted just looking at it*"—P6

Participants noted that their *familiarity* with the task domain also affected how they located relevant information in the text.

> "*The easiest one was the one about ORM/JDBC [databases] because I was so familiar with these technologies*"—P02

Finally, when presented with *ambiguous or contradictory* information, participants had to spend time seeking out additional information to resolve the ambiguity.

> "*I think it was in this one [highlight]. They said that* `cmd` *works in the same way as* `Ctrl`*, but I went with the one that says otherwise. [...] it was actually helpful to have two other highlights so I had a bit more reliance on the thing that was mentioned by more users.*"—P19

Our analysis on the participants' comments on their search strategies provide further insights into how developers forage for information in software development natural language artifacts. One of our key observations is that developers use mixed strategies to locate task-relevant information. In doing so, developers often miss some information that might be relevant for task completion.

> *Developers use mixed strategies to locate task-relevant information, often missing some information that might be relevant to complete a task completely and correctly.*

3.3.4 Summary of Results

The results we obtained from the analysis of highlights produced by developers inspecting artifacts pertinent to six software tasks indicate that between 1% to 20% of the text of an artifact was considered relevant to a task. While we failed to observe syntactic cues that could indicate the relevance of text to a task, we observe consistency in the meaning of the relevant text—as captured using frame semantics—suggesting that semantic-based approaches may be more appropriate for the automatic identification of task-relevant text.

3.3.5 Threats to Validity

In this section, we discuss threats from our examination of the relevant text (HUs) produced by participants who inspected natural language artifacts from six information seeking tasks.

The design of the six tasks in our experiment represent a threat to *construct validity*. To mitigate this threat, we used, as input to the task design, results from studies discussing the search behaviour of developers [106, 181, 189]. For each task, our goal was to provide a set of artifacts so that a participant could gather enough knowledge to complete the task, which we confirmed through pilot studies. The provision of artifacts was necessary to enable a systematic analysis of

relevance through a controlled experiment. We mitigated threats on artifact selection by asking external researchers to provide their own list of candidate artifacts, which was similar to our own list.

The *external validity* of the experimental results is affected by the participants, the tasks and the artifacts. Considering the subjective nature of relevance, the knowledge and particularities of our participants may not generalize to other software developers. Furthermore, the structure and linguistic characteristics of the artifacts considered may not extend to other kinds of artifacts or the same kind of artifacts in of other domains. We tried to mitigate this threat by selecting different kinds of artifacts such as API documentation, bug reports, and Q&A pages while ensuring that each kind was present in at least 50% of the tasks. However, our results are bound to the domains and artifacts we evaluated and may not generalize. Future work should confirm our findings on a wider range of artifacts and tasks.

There are also threats to the validity of our *conclusions*. Experimental procedures encouraged participants to work on the tasks using their normal workflow. However, at least one participant (*P15*) indicated that they would normally revisit a document as different information needs arise as part of a task. As a result, our study may miss relevant information. In addition, the text participants highlighted may not have always aligned with the text that contained information useful for completing the task. To investigate this alignment further, we would have had to create an oracle containing the relevant text. Unfortunately, creating the oracle is challenging because it would need to encompass different perceptions of relevance [139, 149, 161] and usefulness [70, 71]. Thus, to avoid the subjective and imprecise process of judging alignment, we simply consider any text highlighted by participants as containing information useful to completing the tasks.

While the size of our corpus might also affect our conclusions, we emphasize that the HUs from mid and top-tiers in our data comprise 250 sentences, which is similar to the amount of data studied previously. For instance, the McGill [146] and Android [85] datasets contain 238 and 141 relevant text fragments, respectively.

To extract the meaning of text through parsed frames, we used a general frame semantics database rather than one specific to software engineering. This led to frames that were *out-of-context*, causing us to adapt either the frame's name or its description. Frames adapted to software engineering (e.g., *being returned* and

fields) represent a small fraction of all the extracted frames. As a result, we argue that all the sentences containing them still have similar semantic meanings and our conclusions are not affected by these out-of-context frames. However, having specific software engineering frames could improve our results.

3.4 Summary

In this chapter, we address the problem of locating relevant information to a particular software development task *within* a natural language artifact. To better understand how relevant information is encoded in natural language artifacts, we detailed an empirical study investigating what text is perceived as relevant and whether there is consistency in what 20 participants with software development experience deem relevant to a particular software development task.

Our study comprised the analysis of a set of 20 artifacts originating from API documentation, bug reports, and Q&A documents. We observe that finding task-relevant information in such artifacts requires filtering to less than a 20% of the documents' text and that there is substantial variability in what information participants perceive as relevant. Nonetheless, there are commonalities shared through most of the identified relevant textual pieces. These commonalities are found in the semantic meanings extracted from the text suggesting that the semantics of natural language artifacts might can be used in automatic approaches for automatically identifying task-relevant text.

Chapter 4

Android Task Corpus

Developing techniques that can automatically identify text relevant to a task requires a means of evaluating whether a proposed technique works well. In this context, "working well" refers to whether the text identified by an automatic technique is similar to text considered by humans as being relevant to a task at hand. To support the evaluation of proposed techniques, there is a need for a corpus that include tasks to be performed on systems, artifacts representing different kinds of information that would be useful to humans performing the tasks and an identification of text in those artifacts that is helpful to the humans.

As we have described earlier in this thesis, while there exist techniques tuned to identifying relevant text in specific artifact types, there are not techniques able to identify text across a range of artifact types. The corpa that exist consist of tasks with only single, or a small number, of types of artifacts. As a result, there is a need to create a corpa that included multiple artifacts types for a task. This chapter describes the development of a corpus that overcomes the limitations of existing corpa, bringing together tasks from an existing well-known domain, Android development, with relevant text for solving that task from multiple artifact types.

Figure 4.1 summarizes the process we use to create the corpus, which we call $\text{DS}_{android}$. We randomly sample a set of 50 Android development tasks from two common task sources, GitHub[1] and Stack Overflow[2]. For each of these tasks, we

[1] https://github.com/

[2] https://stackoverflow.com/

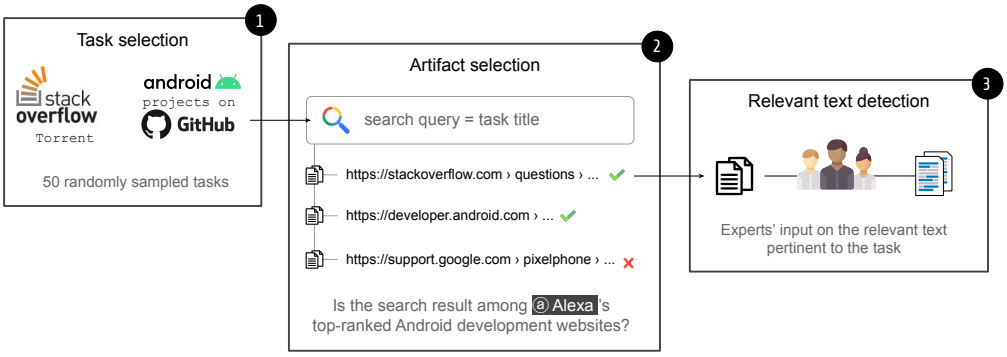

Figure 4.1: Summary of procedures for corpus creation

use the Google search engine to find potential artifacts that likely contain relevant information for that task in top-ranked Android development websites. We then gather task-relevant sentences from these artifacts from experienced developers. We detail each of these steps in turn in Sections 4.1, 4.2 and 4.3. We provide a summary of the final created corpus in Section 4.4.

We publicly share the corpus to help future research in the field [13].

4.1 Software Tasks

We start corpus creation by identifying software tasks for which a developer will likely benefit from the use of additional information to complete. We scope task selection to *Android development* because the Android software development kit (SDK) evolves constantly due to functionality, security and performance-related improvements [108, 129]. These improvements impact its development community, requiring them to often seek information regarding changes in the SDK [31, 113, 130]. For example, over 35,000 developers have used Q&A forums to discuss tasks covering 87% of the classes in the Android API [143].

Two common places where Android task can be found are:

- the description of an issue (e.g., a bug or feature request) reported in an issue tracking system; or in
- a post in a community forum, development mailing lists, and others.

Several studies have used such sources for software tasks [19, 23, 137, 190] and, following the lead of these studies, we select GitHub issues and Stack Overflow (SO) posts on Android development as the two sources for the tasks in our corpus.

GitHub tasks

To select tasks from GitHub, we are guided by studies that use stars [36, 37] as a proxy for a project's popularity [66, 188]. We selected 14 projects, ranging from mail clients[3] to development frameworks[4], by filtering the list of top-starred projects in GitHub to those with the *Java* and *Android* tags. We then randomly selected 25 distinct issues originating from these starred projects as the GitHub tasks of our corpus (average of 1.78 issues per project). While selecting issues, we took care to check that they had at least one follow-up comment and that the issue title did not contain certain words, e.g., `test` or `ignore`, as these words indicate issues created automatically by scripts or bots—a common pitfall that researchers must be aware of when mining GitHub [91].

Figure 4.2 shows an example of a GitHub task in our corpus. Although the expected behaviour is that the app controls should be visible even with the screen locked, a user reports that the app screen is missing. A developer addressing this issue might need to review the Android lock task documentation [5] or refer to examples of applications that use the Android lock screen [12]. For the remainder of this chapter, we use the lock screen task as a running example.

Stack Overflow tasks

We consider Stack Overflow posts as software tasks because to answer a post, a developer often needs to provide references supporting their answer [192]. Finding these references in a timely manner and writing the key information that helps a user understand the provided solution encompasses many of the activities found in a developer's daily work, e.g., work-related browsing, coding, debugging, and reading/writing documentation [131]. For example, Figure 4.3 depicts a task where

[3] https://github.com/k9mail/k-9
[4] https://github.com/libgdx/libgdx

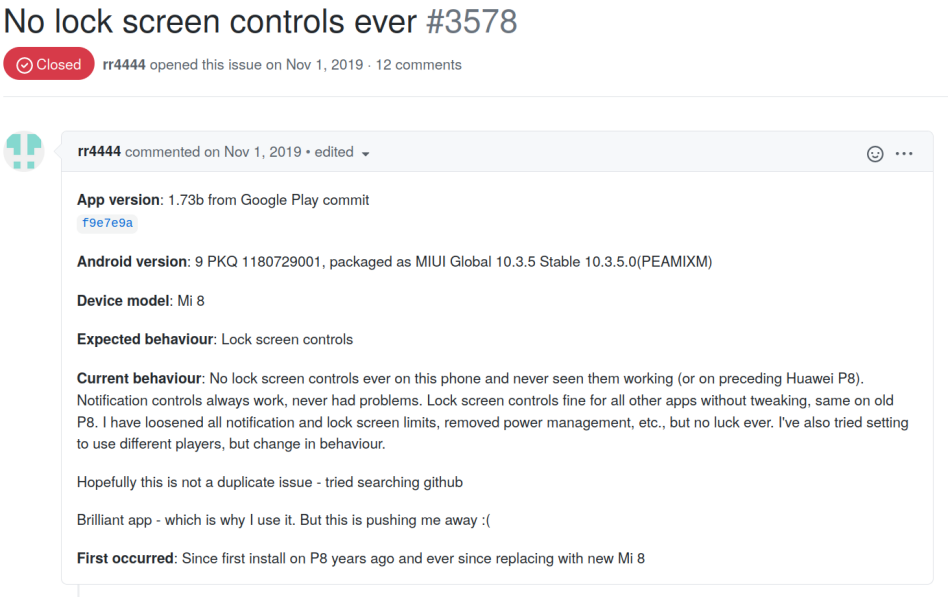

Figure 4.2: Sample GitHub task from our corpus

a developer describes her struggles using the Android WebView component. To answer this question, a developer will not only provide a code snippet, but also explain key points of the Android WebView API [6] and how they were used in the solution provided to the task, as presented in Figure 4.4.

We randomly select 25 Stack Overflow posts from a curated list about Android development [26]. This list was built by Baltes et al. using the Stack Overflow dump published on June 5, 2018 [24, 25] and it contains 209,536 unique posts with the *Java* and *Android* tags.

4.2 Artifact Selection

When selecting artifacts pertinent to a task in our corpus, we seek to simulate everyday practices on how developers search the Web [153, 189]. We formulate a query for each task and use a Web search engine to retrieve artifacts that are pertinent to that task, as described below.

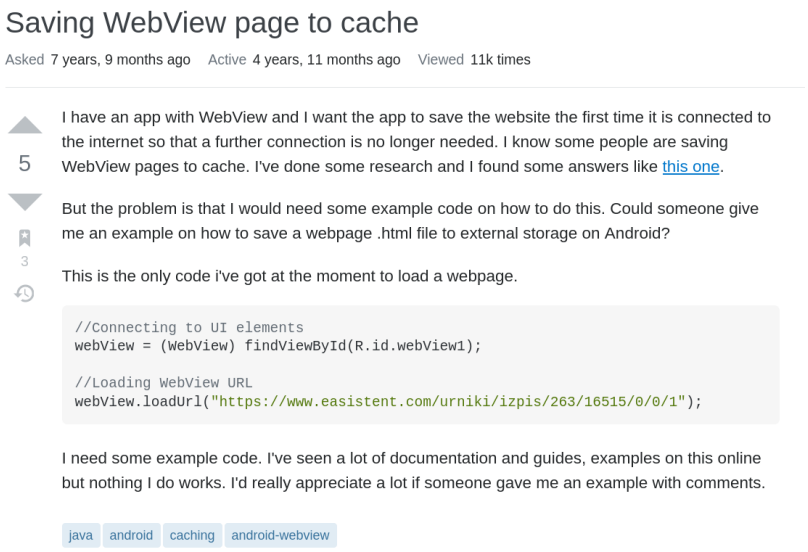

Figure 4.3: Sample Stack Overflow question

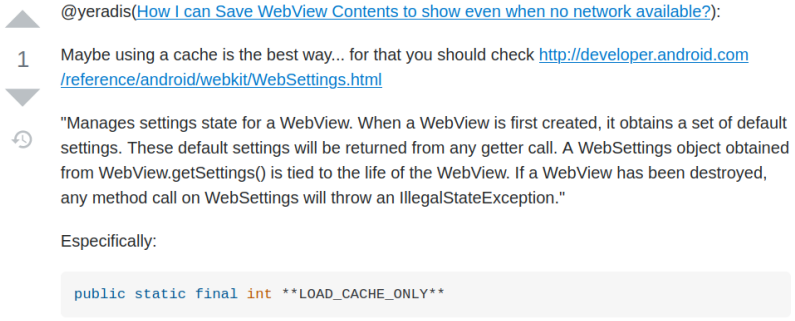

Figure 4.4: Excerpt of a Stack Overflow answer

Artifact sources

The artifacts sought to find useful information or knowledge for completing a task depend on the type of task a developer performs. For example, when using a new framework or library, a developer refers to official API documentation [106, 156] while, for debugging or error diagnostic tasks, community-based sources are preferred [39, 106]. Despite such variability, researchers have observed that Web blogs, API documentation, and community forums are sources commonly used

by developer to forage information that assists task completion [88, 106].

We use this knowledge to restrict artifact selection to well-known and studied artifact types within these sources [93, 106, 174], namely Android and Java SE API documentation, GitHub issues, Stack Overflow answers, and Web tutorials or blog posts on Java and Android development.

Query formulation

Coming up with proper search terms is a critical step of any search [79] and, ideally, we should be able to formulate a query with terms able to retrieve the most pertinent artifacts for a software task. However, studies have shown that developers perform poorly in identifying good search terms [93, 103, 174] and thus, using a task's title as an educated approximation to terms that a developer might use is a common procedure adopted by other studies in the field (e.g., [190] or [169]). Hence, we use a task's title (i.e., SO question or GitHub issue title) as the seed to search for pertinent artifacts.

Search results

We use `googlesearch` API [11] to request up to 5 resources per query adding `site:domain` to search for artifacts only in (or outside) a given web domain[5]—procedures similar to [190].

From the results returned, we include up to one API document, one GitHub issue discussion, one Stack Overflow answer, and two miscellaneous web pages in the final artifact set for a task. When selecting results, we exclude any entry that does not appear in the Amazon Alexa [4] web traffic statistics for Java and Android development in the period from April 2020 to March 2021. We apply this filter to include software development artifacts and remove results such as a tutorial about "*stock swap*" operations which was initially fetched for a task discussing "*left and right-hand swap*". Table 4.1 shows one search result per artifact source for the GitHub task introduced in Section 4.1

Limiting the number of artifacts up to a maximum of 5 artifacts per task relates to the time-consuming [15] and cognitively demanding [148] nature the final

[5] e.g., `site:stackoverflow.com` for a query searching for Stack Overflow artifacts

step in the dataset creation, i.e., asking annotators to carefully read, understand and identify the text within the fetched artifacts that relevant to a given task, as Section 4.3 further details.

No lock screen controls ever	
API documentation	Lock task mode - Android Developers
Github issues	Lock screen controls disappear on Android 11
StackOverflow answers	Media Control on Lock Screen like Google Play Music in android?
Miscellaneous	Create A React Native App - Which works on Lock Screen (Android)

Table 4.1: Sample of artifacts obtained for a Github task [9]

Artifact's content

Last, we need to extract the natural language text within an artifact so that techniques that automate the identification of text relevant to a task can be built using our corpus. This step requires processing an artifact's content into a sequence of individual sentences, what prompted us to follow common procedures for processing the artifact types found in our corpus [19, 137]. That is, given a search result `URL`, we use use a series of python APIs[6] to fetch the artifacts' content and then, we use the Stanford CoreNLP toolkit [1] to identify individual sentences in the artifacts' content.

4.3 Relevant text detection

Next, we need to determine which of the text in the gathered artifacts could provide information that assists a developer in solving her task. In our corpus, this text represents *golden data* that one can use to design and evaluate automatic tools that assist developers in the identification of information useful to their tasks. To produce it, we ask experienced developers to mark the text that they deem useful and that provide information for tasks assigned to them [127, 137, 155].

[6] `BeautifulSoup` [8], `StackAPI` [3] and `PyGithub` [2]

4.3.1 Annotation process

Our intention is that golden data reflect text that instructs developers to perform important actions to accomplish their task [119, 155]. To this end, we describe the annotators' background, annotation procedures, and the the text inspected by the annotators.

Annotators

We recruited 3 graduate students with professional programming experience to produce *golden* data for our corpus. Annotators had to have experience with Java development and they also had to be familiar with the types of artifacts they would encounter throughout the annotation process. On average, annotators self-reported 3.0 years of professional programming experience (stdv 1.63, ranging from 1 to 5 years).

Annotation procedures

The annotation process started with two tasks, other than the ones in our dataset, so that annotators could familiarize themselves with annotation procedures. For each task, annotators had the task description and links to artifacts pertinent to the task at their disposal. We asked annotators to write a short (250 words max [154]) with instructions that a developer could follow to complete the task successfully. The purpose of the plan was to ensure that annotators built enough context about the task. While perusing artifacts, annotators also had to manually highlight sentences that they deemed useful and that provided information that assisted task completion—instructions similar to the ones used for the creation of the data in Nadi and Treude's study [137] or in the $\text{DS}_{synthetic}$ corpus [127].

Annotation was facilitated by a tool we created for this purpose. Figure 4.5 shows a screen shot from the tool in action, which works as a browser plug-in. The top-right corner panel in the figure shows the browser extension. When an annotator clicked the `highlight` button, the tool instrumented the HTML of a page identifying individual sentences. The tool then allowed annotators to hover over identified sentences and to select them as relevant by clicking on the hovered text. For example, in the first paragraph, an annotator selected the sentence "*Call*

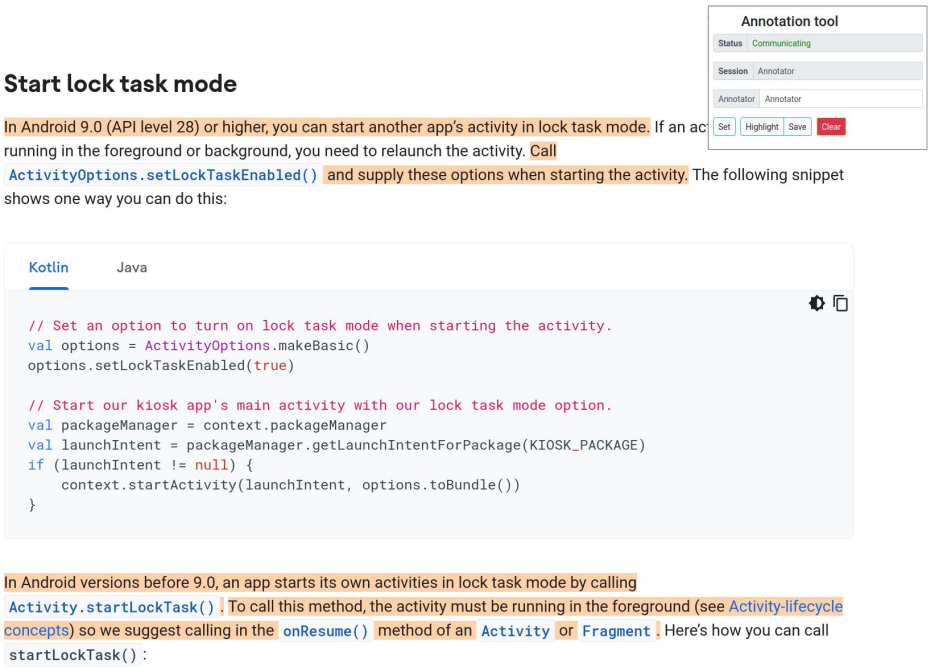

Figure 4.5: Annotation tool and relevant sentences marked by an annotator

ActivityOptions.setLockTaskEnabled() ... when starting the activity" as relevant to the Android lock task (Figure 4.2).

We discussed results from these two tasks with all annotators to ensure consistency over responses to any of the questions annotators had. No changes arose from the two introductory tasks. After this step, we presented the tasks in the $\text{DS}_{android}$ corpus to the annotators, dividing them into batches of 10 tasks each. Annotation procedures for these tasks were analogous to the introductory tasks.

Text inspected

Annotators had to inspect a total of 12,401 sentences originating from API documentation, Stack Overflow answers, GitHub issue discussion and miscellaneous Web pages. These sentences comprise natural language written text—in this case, English—and do not involve source code snippets that may appear alongside the text.

Table 4.2 gives insight into the number of sentences inspected per artifact type. We observe that API documents and miscellaneous Web pages contain the highest number of sentences in our corpus. This is not surprising because API documents often contain boilerplate text and all the information needed for the usage of an API element is usually found in a single document [156]. Miscellaneous Web pages comprise blogs or tutorials, which often provide step-by-step instructions and accompanying examples, what likely explains the large number of sentences for this type of artifact [20, 85].

The content of GitHub issues mostly resembles conversations [154] and, beyond code snippets and minimal structured fields, the 1,890 sentences inspected in this type of artifact comprise the description of a reported bug or a feature request as well as questions, answers, and discussion from community members who are interested in resolving the issue at hand [196].

For SO answers, the fewer number of inspected sentences (1,420) potentially relates to the fact that Stack Overflow offers little incentive for further discussion once a post has been answered. Nonetheless, it is worthy noting that sentences with crucial information are not limited to the ones in an accepted answer [137] and that later comments can be equally or more informative, often providing notes about updates in an API component or framework [195].

	# of sentences		
	total	mean	stdv
API documentation	4,915	109.22	96.77
GitHub issues	1,890	43.95	33.69
SO answers	1,420	28.40	28.48
Miscellaneous Web pages	4,176	70.78	56.64
Overall	12,401	62.95	66.25

Table 4.2: Summary of sentences in the $\text{DS}_{android}$ corpus

4.3.2 Corpus Description

Overall, the resultant $\text{DS}_{android}$ corpus has 50 tasks and 133 artifacts, divided as 33 API documents, 45 Stack Overflow answers, 20 GitHub issues, and 35 other web pages. Tasks in the dataset have an average of 3 associated artifacts each and we discriminate the text marked by each annotator per artifact. Overlap between the

text marked by the annotators represents 30% of the entire data and *Krippendorff's alpha* [100] indicates good reliability[139] over the text marked as relevant (or not) by the annotators ($\alpha = 0.69$).

We structure tasks and annotation results in a format similar to the one shown in Table 4.3. Each task has a title, link, description, and a set of pertinent artifacts. In turn, each artifact has a type, title, link, and content, where each sentence within an artifact's content is preceded by the set of annotators (i.e., `none`, `A1`, `A2`, or `A3`) who marked the sentence as useful to the task. As an example, in a Stack Overflow answer (artifact_2), all three annotators marked the sentence "*Have you checked RemoteControlClient?*" as relevant to the lock mode task.

Overall, annotation required a total of ≈ 45 hours of manual work and it was done throughout the course of 5 weeks. Table 4.4 provides summary statistics for the text marked by the annotators over all the artifacts inspected as well as on an artifact type basis according to the text marked by any annotator. On average the text deemed useful to a software task in the artifacts inspected comprises 8.93 sentences per artifact per annotator. We observe that the highest number of sentences marked originate from miscellaneous artifacts while the lowest come from GitHub issue discussions. The high number of sentences marked in the former might relate to the more didactic nature of web tutorials [20, 86]. For the latter, the content on GitHub may be too project-specific what might prevent a developer performing a task in a different context from benefiting from most of the content found in this type of artifact. For example, discussions about whether a certain design is the most appropriate [184] might not extend to other projects, but instructions on how to use an API element might.

Task	
No lock screen controls ever	link
Description	
... **Expected behaviour:** Lock screen controls **Current behaviour:** No lock screen controls ever on this phone and never seen them working (or on preceding Huawei P8). Notification controls always work, never had problems. Lock screen controls fine for all other apps without tweaking, same on old P8. I have loosened all notification and lock screen limits, removed power management, etc., but no luck ever. I've also tried setting to use different players, but change in behaviour. ...	
Artifact$_1$ - API documentation	
Lock task mode Android Developers	link
Content	
none	Android can run tasks in an immersive, kiosk-like fashion called lock task mode.
A1	Only apps that have been allowlisted by a device policy controller (DPC) can run when the system is in lock task mode.
none	A DPC must allowlist apps before they can be used in lock task mode.
A1, A2	Call DevicePolicyManager.setLockTaskPackages() to allowlist apps for lock task mode as shown in the following sample
	...
Artifact$_2$ - Stack Overflow answer	
Media Control on Lock Screen like Google Play Music in android?	link
Content	
A1, A2, A3	Have you checked RemoteControlClient?
A2, A3	it is used for the Android Music Remote control even if the App is in Lock mode.
	...

Table 4.3: Example of how data is structured in the $\text{DS}_{android}$ corpus. Each task has a title, link, description, and a set of pertinent artifacts. Each artifact has a title, link, and content. For each of the sentences in the content, we store the set of annotators (i.e., none, A1, A2, or A3) who marked the sentence as useful to the task

	# of sentences marked			% of sentences marked by		
	total	mean	stdv	1 annot.	2 annot.	3 annot.
API documentation	327	9.62	7.88	76%	16%	7%
GitHub issues	146	4.87	3.37	74%	20%	6%
SO answers	330	7.33	5.08	57%	22%	21%
Miscellaneous Web pages	590	12.55	9.88	75%	17%	8%
Overall	1393	8.93	7.78	71%	19%	10%

[annot] annotator(s);

Table 4.4: Summary statistics for the text deemed useful by annotators across the artifacts inspected in the $\text{DS}_{android}$ corpus

Table 4.2 also reports the percentage of sentences marked by one, two or the three annotators who inspect the artifacts in the corpus. Our rationale to provide this data is that individuals might use different criteria to assess the usefulness of a sentence to a given task [28, 29]. Out of 1393 unique marked sentences, 10% were marked by all annotators, 19% by two of them and the remainder—996 sentences—by a single annotator. These ratios follow Nenkova and Passonneau's empirical findings on content selection [139] and are a similar to the ratios in the $\text{DS}_{synthetic}$ corpus, where sentences marked by a single annotator comprise the majority of the data. Although we leave the in-depth analysis of how an individual's background plays a role in what they perceive as relevant for future studies, evaluation metrics should outline how the differences between the text marked are taken into account when using the golden data provided in our corpus for evaluation purposes.

Based on the text marked by the annotators, we also ask if the amount of information required to solve a task was similar in GitHub or Stack Overflow tasks. Answering this question will allow us to know whether the corpus can be evenly used for evaluation purposes or whether we must distinguish tasks according to their origin. To answer this question, Table 4.5 shows the number of sentences marked by any annotator according to a task's origin. We use a Wilcoxon-Mann-Whitney test [122] to check if there are any differences between the average number of sentences marked per type of artifact in GitHub and Stack Overflow task. Results show there is no statistically significant difference between the number of sentences sentences marked. Hence, the corpus can be used for evaluation purposes

without the need to distinguish tasks based on their origin.

	GitHub # of sentences marked			Stack Overflow # of sentences marked		
	total	mean	stdv	total	mean	stdv
API documentation	144	9.60	5.83	180	10.00	9.29
GitHub issues	78	4.88	3.66	60	5.00	3.00
SO answers	112	5.60	3.87	218	8.72	5.50
Miscellaneous	181	12.14	9.55	420	12.73	10.01
Overall	515	7.75	6.61	878	9.98	8.47

Table 4.5: Number of sentences marked per type of task

4.4 Summary

In this chapter, we introduced the need for corpora for the development of automatic techniques able to identify relevant text to solve a task in artifacts pertinent to the task. Since no such corpora existed, we detailed a set of structured procedures for its creation. The DS$_{android}$ corpus consists of 12,401 unique sentences originating from artifacts associated with 50 software tasks drawn from GitHub issues and Stack Overflow posts about Android development. We found that three annotators with professional experience indicated that, out of these 12,401 unique sentences, 1,393 of them were relevant to a particular task and that multiple annotators agreed of the relevance of 29% of the sentences, which lead us to provide recommendations on how our data can be used for evaluation purposes. Ultimately, we expect that the DS$_{android}$ corpus provides a foundation for studies that explore relationships between software tasks and text found across different types of artifacts that a developer might seek information on and that are pertinent to the developer's task.

Chapter 5

Identifying Task-Relevant Text

The information a developer seeks to help aid the completion of a task typically exists across a range of artifacts. To aid developers in identifying, from the large amount of text in these documents, just the fraction of text relevant to the task-at-hand, prior work has used textual properties—alongside an artifact's meta-data—to identify likely relevant text (Chapter 2). Although effective, these techniques target specific types of artifacts, limiting their use across the many different kinds of artifacts developers encounter daily in their work.

In this chapter, we consider the semantic patterns that arise from the empirical analysis of the task-relevant text (Chapter 3) and we explore a design space of possible techniques building on approaches to interpret the meaning, or semantics, of text to identify task-relevant text across different kinds of software artifacts. We introduce six possible techniques that incorporate the semantics of words and sentences. We show that some of the proposed semantic-based techniques compare to existing artifact-specific techniques while applying to a broader set of types of artifacts.

We start by outlining the hypotheses that motivate the techniques that we explore (Section 5.1) followed by detailed descriptions of the techniques (Section 5.2). We show how the six techniques compare against state-of-the-art artifact-specific techniques and their accuracy across different types of artifacts (Section 5.3). Section 5.4 summarizes our key findings.

5.1 Problem Statement

In Chapter 3, we observed consistency in the meaning of text of different types of artifacts that participants deemed relevant to six software tasks. Guided by these findings and aslo by recent success in studies that use techniques that interpret the meaning, or *semantics*, of text for a variety of development activities, such as for finding who should fix a bug [191], searching for comprehensive code examples [169], or assessing the quality of information available in bug reports [45], we ask:

> *to which extent can semantic-based techniques identify task-relevant text across different kinds of software artifacts?*

To investigate this question, we introduce six possible techniques that incorporate the semantics of words and sentences to identify textual information likely relevant to a developer's task.

5.2 Techniques

We introduce six novel semantic techniques that might aid developers in identifying task-relevant text in software artifacts. Three of the techniques use word2vec [134] as a base for determining a ranked list of similar sentences between an artifact and a task description and three use BERT [57]. Each of these technique takes as input a task description and an artifact. The technique then identifies sentences in the artifact pertinent to the task. The output from the technique is a ranked set of sentences from the artifact from most likely to be helpful for solving the task to the least helpful.

Table 5.1 provides an overview of the six techniques. Two techniques simply returns the results of *word2vec* or *BERT*. The four others apply *frame semantics* [67] to filter each of the sentences returned by the base techniques. A first filter—*frame-elements*—retains only sentences that include certain frames. The second filter—*frame-associations*—keeps sentences with certain co-occurrences of frame semantics between the task description sentences and the artifact sentence. We describe word2vec, BERT and the filters in turn to explain the six techniques.

Base technique	Filters	Description
	no filter	Uses the Skip-gram model to identify relevant sentences as the sentences most semantically similar to a task
`word2vec`	*w/ frame-elements*	Modifies the output of the `word2vec` according to whether sentences contain meaningful frame elements
	w/ frame-associations	Modifies the output of the `word2vec` according to whether sentences contain meaningful task-artifact frame pairs
	no filter	Fine-tunes BERT to predict the sentences that are likely relevant to an input task
`BERT`	*w/ frame-elements*	Modifies the output of the `BERT` according to whether sentences contain meaningful frame elements
	w/ frame-associations	Modifies the output of the `BERT` according to whether the sentences contain meaningful task-artifact frame pairs

Table 5.1: Summary of semantic-based approaches for automatically identifying task-relevant text

5.2.1 Identifying Task-relevant Text with word2vec

To form the base of the first three techniques, we use word semantics via word2vec in conjunction with information retrieval approaches. Information retrieval techniques are usually applied to look for documents relevant to a query prompted by a user [30], where the relevance of a document to a query is based on some function that indicates how similar the document's content is to that query [124]. Given the right *similarity function*, information retrieval techniques can be also applied to identify sentences within a document, artifact in our problem formulation, that are relevant to a task.

We use the *Skip-gram* model [134], also referred to as *word2vec*, to define such a similarity function. The *Skip-gram* model exploits Harris' distributional hypothesis [80]—which states that words that appear in a similar context tend to have similar meanings—and builds vector representations, namely *word embeddings*, for each of the words in a text corpus. With a significantly large text corpus, the model associates similar vector embeddings to words that are similar in meaning [193].

Since *word2vec* represents words through a vector space, the similarity of two words i and j can be obtained computing the cosine similarity [124] between the

corresponding vectors of each word, i.e., w_i and w_j:

$$sim(w_t, w_a) = \frac{w_t^T w_a}{\|w_t\| \|w_a\|} \tag{5.1}$$

In turn, the similarity of two sentences can be by first computing a vector representation for the entire sentence and then, by measuring the cosine angle between the obtained sentence vectors. To obtain sentence vectors, we follow Conneau et al.'s guidelines [49] and we average the sum of the embeddings for each word in a sentence.

Following these procedures, we use a Skip-gram model with word embeddings trained for the software engineering domain [61] to compute the semantic similarity, or relevance, of the sentences $\{a_1, a_2, \ldots, a_n\}$ within a pertinent artifact A and the sentences $\{t_1, t_2, \ldots, t_m\}$ in a task T. As both entities have multiple sentences, the semantic similarity of a sentence $a_i \in A$ is the maximum value obtained for this sentence and a vector representing the average of the embeddings of the text in a task.

After we compute the semantic similarity of all the sentences in an input artifact with regards to an input task, we sort the obtained values from highest to lowest, outputting the top-n sentences with highest similarity as the ones likely relevant to an input software task.

5.2.2 Identifying Task-relevant Text with BERT

In the Skip-gram model, context refers to the sentences used to train the model. This approach to context does not allow the model to disambiguate words based on their surrounding text. In other words, a Skip-gram model will have a single vector representation for a word such as '*company*' even when it can have different meanings, i.e., a business organization or being with someone. In contrast, the Bidirectional Encoder Representations from Transformers (BERT) [57] provides different representations for the same word based on the sentence in which a word appears. This additional layer allows the model to perform more complex operations, leading to state-of-the-art results in several tasks [57].

Typically, BERT is trained on a massive amount of data. During training, a percentage of the tokens in a sentence—usually 15%—are replaced with a special

token and the model is optimized to predict these replaced, or *masked*, tokens. The model relies on a mechanism, called *attention* [182] that correlates and weights all non-replaced tokens in the input so that the model maximizes predictions. To fully train a BERT model, one first creates a base model using a large amount of text and the general token prediction task. Then, this base model is fine-tuned to specific tasks, such as text classification, using a dataset specific to the fine-tuning steps. The procedures of *training* and *fine-tuning* allow using the model even when training data is scarce and the model transfers what it learned during training steps to the fine-tuning steps [57].

We posit that BERT's attention mechanism and fine-tuning procedures can be used to train the model to classify sentences in a natural language artifact as likely relevant to software task.

The second group of three techniques we introduce uses a BERT model to classify if a sentence is relevant or not to a task. To this end, we take the sentences $\{a_1, a_2, \ldots, a_n\}$ within a pertinent artifact A and the sentences $\{t_1, t_2, \ldots, t_m\}$ in a task T and we feed them to the model as task-artifact input pairs alongside binary labels representing whether that sentence is relevant. Using training data, the model's attention mechanism will learn associations between each pair such that it can predict if a sentence is relevant for the task provided. We use up to 10 epochs to train BERT, setting both *batch size* and a *sequence length* to 64. Cross-Entropy is our loss function, and the model is trained to minimize it using the Adam optimizer at a learning rate of *1e-5* with early stopping.

Following these procedures, we take a fully trained model and we compute the relevance of a sentence $a_i \in A$ when we pair it to to a vector that represents all the sentences in a task T. Similar to the word2vec approach, we predict the relevance of all the sentences in an input artifact with regards to an input task and we use prediction probabilities to select the top-n sentences with highest probability as the relevant ones.

5.2.3 Filtering Task-relevant Text with Frame Semantics

Although word2vec and BERT have provided significant improvements to a diverse set of natural language tasks, they still capture semantics at the word level. If

we seek to infer a sentence's meaning, an alternative would be to consider *frame semantics* (Chapter 3).

We define two filters using frame semantics that can be applied with either the word2vec or BERT base techniques as a post-processing step to identify sentences potentially relevant to a task based on the frame elements that the *SEFrame* tool identifies [127]. For that, we implement a set of filters that can be applied to the previous techniques as a post-processing step [124]. A filter takes the output of a technique O and produces output O' where a sentence must match the filter's relevance criteria. For both filters, we use SEFrame [128] to assign frames relevant to software engineering text.

Frame-elements

This filter considers sentences identified with frames giving instructions, describing required events, explaining system procedures, and others as likely to contain information of interest. The rationale for the set of frames identified is based on findings from related work that have shown that the meaning of the text is an indicator of its relevance to a software task [127]. This filter takes a sentence $a_i \in A$ and checks whether any of the frame elements of this sentence match one meaningful frame $f \in F$ from a pre-established set[1].

$$F = \{\text{being obligated}, \text{causation}, \ldots \text{required event}, \text{using}\} \tag{5.2}$$

$$\textit{frame-elements}(a_i) = \begin{cases} 1, & \text{if } \textit{frames}(a_i) \cap F \neq \varnothing \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

Frame-associations

This filter considers that the relevance of a sentence depends on the intentionality of a task. For example, for a task that requires diagnosing an error, sentences in a bug report that describe success or failure of an action are likely relevant. This rationale is represented by a set of associated frame pairs, where each element represents a

[1] See full list in our replication package [13]

frame originating from a task and another frame originating from a sentence. This filter checks if a task-artifact frame pair obtained from an input artifact and input task appear in a pre-established set of pairs[1].

$$P = \{(\text{execution}, \text{being obligated}), \ldots (\text{questioning}, \text{using})\} \tag{5.4}$$

$$\textit{frame-associations}(a_i, T) = \begin{cases} 1, & \text{if } \textit{pair}(a_i, T) \cap P \neq \varnothing \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

We identify frame-pairs $p \in P$ using association rule mining [14], which identifies frame elements that co-occur in a task's title and in the text marked as relevant in the artifacts associated to each task. We mine pairs using the apriori algorithm [7] with a minimum support level—the minimum number of times the frame element pairs must appear across the data—of 0.10. Section 5.3 further details the data used to identify rules.

5.3 Evaluation

The goal of our evaluation is to assist the design of tools that can apply to multiple kinds of software artifacts so that software developers can rely upon one technique instead of many. To be useful, the one technique that applies to multiple artifacts must perform similarly to techniques specialized to particular kinds of artifacts.

We first evaluate whether any of the six techniques we introduce performs similarly to a technique specialized to one kind of artifact. We chose to compare the performance of our techniques to that of AnswerBot on Stack Overflow answers because it is the technique closest to our work (Chapter 2). This evaluation helps answer:

> *How do our six semantic-based techniques compare to a state-of-the-art technique that is specific to a particular artifact type?*

We then consider whether the semantic-based techniques can perform equivalently well across multiple artifact types. This evaluation helps answer:

> *Which semantic-based technique provides the best results?*

We use *precision* and *recall* metrics [123] to measure what portion of the text identified by human annotators in the $\text{DS}_{android}$ corpus the techniques automatically identify. In this context, we believe recall to be the most important metric since failure to identify text that is relevant to a task means that a developer will have an incomplete or partial view of the information needed, what can lead to faults [136].

5.3.1 Experimental Setup

To evaluate the six techniques and determine how much of the task-relevant text in the $\text{DS}_{android}$ corpus they are able to identify, we use the following experimental setup.

Techniques' Output

We configure each technique to identify a target number of 10 relevant sentences for each task-artifact pair. This decision is based on the fact that no more than 15% of the content of any artifact in our dataset is deemed relevant to a task, which,

on average, accounts for 8.93 sentences (Chapter 4). Researchers have also used the same target number of 10 sentences when evaluating techniques (e.g., [190] or [119]) able to identify relevant text for certain kinds of artifacts, what will also facilitate comparing our results to related work.

Training & Testing Data

In addition to configuring the techniques' output, two of our techniques require training data for fine-tuning purposes (`BERT`) and to derive task-artifact frame pairs (`frame-associations`). We ensure that no data used to evaluate these techniques is also used to train them by splitting the dataset into two portions, one for training and another for testing, each with an equal number of tasks. Due to our comparison with AnswerBot, we also ensure that all tasks used for testing purposes have associated Stack Overflow artifacts. That is, we create our test set randomly selecting 25 tasks in the dataset that contain a Stack Overflow artifact among its associated artifacts.

Metrics

We compute values for *precision* and *recall* metrics based on the sentences deemed relevant to a task by *at least two* human annotators. To minimize risks from a scenario where one of the techniques and their underlying approaches has a peak or bottom performance due to training procedures or due to factors beyond our control, we compute results for each technique over 10 distinct executions over the test data, reporting the average of each metric over all runs.

We compute values for *precision* and *recall* metrics based on the sentences deemed relevant to a task by *at least two* human annotators. To minimize risks from a scenario where one of the techniques and their underlying approaches has a peak or bottom performance due to training procedures or due to factors beyond our control, we compute results for each technique over 10 distinct executions over the test data, reporting the average of each metric over all runs.

For a detailed definition of each metric, we refer to the evaluation outcomes in Table 5.2, where columns represent labels provided by the annotators and rows, the text identified as relevant or not by a technique.

	Relevant	**Not-relevant**
Identified as relevant	true positive (TP)	false positive (FP)
Identified as Not-relevant	false negative (FN)	true negative (TN)

Table 5.2: Evaluation outcomes

Precision Precision measures the fraction of the sentences identified that are relevant over the total number of target sentences identified, as shown in Equation 5.6.

$$Precision = \frac{TP}{TP+FP} \tag{5.6}$$

Recall Recall represents how many of all the annotated sentences are identified by a technique (Equation 5.7).

$$Recall = \frac{TP}{TP+FN} \tag{5.7}$$

Precision means identifying only text that is relevant, whereas recall means identifying all relevant text. Ideally, we would aim for a technique with high precision and high recall. Unfortunately, this is often not possible and we must reach a compromise. As described earlier, our goal is to support developers to locate text that might be relevant to their task, and not locating all the relevant text may lead to incomplete or incorrect solutions, thus the reason why we favour recall.

5.3.2 Comparison to AnswerBot

For a fair comparison between our techniques and AnswerBot (`AnsBot`), we must ensure that we obtain measurements for all the approaches under the same circumstances. This could mean applying our techniques to the tasks and artifacts used in AnswerBot's original evaluation or applying both our semantic-based techniques and AnswerBot to our test data. It would be interesting to report results for both scenarios, but golden data in AnswerBot represents how human evaluators judged the target sentences outputted by the tool. This comprises only a portion of the entire text available in a Stack Overflow answer, which can have more text that these evaluators could have judged as relevant. Since we do not have access

	no filter		*w/ frame-elements*		*w/ frame-associations*	
technique	**precision**	**recall**	**precision**	**recall**	**precision**	**recall**
AnsBot	0.59	0.63	-	-	-	-
word2vec	0.43*	0.48*	0.49*	0.45*	0.45*	0.50*
BERT	0.58	0.63	0.53*	0.58*	0.58	0.62

* AnsBot performs better with a large effect size ($\alpha = 0.1$, *Cohen's D* ≥ 0.8);

Table 5.3: Comparison to AnswerBot

to AnswerBot's original judges to construct this data, we evaluate our techniques and AnsBot only on the tasks and Stack Overflow artifacts in the portion of the DS-android dataset that we use for testing.

Since AnsBot uses both textual properties and meta-data, we also evaluate how much of the tool's accuracy relies on properties such as the number of votes an answer got on the platform or whether the answer has been accepted. For that, we disable any of the features used by AnsBot that rely on meta-data and we compute evaluation metrics once more. We refer to this configuration in our results as $\texttt{AnsBot}_{\texttt{text}}$.

Results

Table 5.3 shows values of precision and recall for AnsBot and the semantic techniques that we explore. In the table, rows provide details about a specific technique while columns discriminate precision and recall values and which post-processing filters were applied. We also mark results from paired Wilcoxon-Mann-Whitney tests [122] that check if the results between each technique and AnsBot are statistically different.

Overall, we observe that AnsBot achieves 0.59 precision and 0.63 recall—values explainable by the fact that the tool is tailored specifically to Stack Overflow. When we compare AnsBot to our techniques, we observe that the base BERT approach and the one using frame-associations achieve result comparable to AnsBot. Interestingly, the frame-elements filter does not provide significant improvements to the base approach. The lack of differences may be explained by the fact that, to determine relevance, the attention mechanism used by BERT already correlates the text in a task and in an artifact (Section 5.2.2), and that the

	no filter		*w/ frame-elements*		*w/ frame-associations*	
technique	**precision**	**recall**	**precision**	**recall**	**precision**	**recall**
$\texttt{AnsBot}_{\texttt{text}}$	0.53	0.53	-	-	-	-
`word2vec`	0.43*	0.48*	0.49*	0.45*	0.45*	0.50*
`BERT`	0.58†	0.63†	0.53	0.58†	0.58†	0.62†

† `BERT` performs better with a large effect size ($\alpha = 0.1$, *Cohen's D* ≥ 0.8);
* $\texttt{AnsBot}_{\texttt{text}}$ performs better with a large effect size ($\alpha = 0.1$, *Cohen's D* ≥ 0.8);

Table 5.4: Comparison to $\text{AnswerBot}_{\text{text}}$

vector representations in the model are able to infer contextual information, which may serve as an implicit way to identify a sentence's meaning. For `word2vec`, we observe that the technique's results are significantly lower than `AnsBot`, which suggests that it fails to identify relevant text identified by `AnsBot`.

Without meta-data, results (Table 5.4) indicate that all `BERT` techniques achieve significantly higher recall than $\texttt{AnsBot}_{\texttt{text}}$. This suggests that, to determine relevancy, a neural network might implicitly use some of the properties of text that appears in highly voted or accepted answers.

Results from our comparison to a state-of-the-art approach, namely AnswerBot, indicate that text-based semantic techniques achieve comparable accuracy when identifying relevant information in Stack Overflow artifacts.

5.3.3 Evaluation Across all Artifact Types

Provided that we can identify task-relevant textual information with accuracy comparable to a state-of-the-art approach, we measure how much of the text that is relevant to a task (within an artifact) can our semantic-based techniques identify. To this end, we compute precision and recall metrics over all the artifacts in the test data. To assist comparisons, we also report results for the text identified by a standard VSM lexical similarity approach. Several studies [71, 98, 127] have shown that developers often use keyword-matching as a first search strategy to locate text that might contain information relevant to their tasks. Thus, this `baseline` might assist us in interpreting how much of the task-relevant text a developer would identify by themselves when inspecting an artifact at first glance.

We also consider precision and recall metrics per type of artifact, i.e., API

documentation, Stack Overflow answers, GitHub issues, and miscellaneous web pages. We measure how much each metric varies across these artifacts reporting their standard deviation. The less variation there is, the more a technique consistently identifies task-relevant textual information regardless of an artifact type.

Results

Table 5.5 summarizes the average of precision and recall metrics when identifying task-relevant textual information for all of the test data. Based on the overall results for each technique, we observe that recall scores range from 0.43 to 0.58. We also note that applying sentence-level filters improves precision and recall values for `word2vec`. Notably, `word2vec` with the `frame-elements` filter achieves up to 0.49 recall. For `BERT`, recall values range from 0.56 to 0.58 and, similar to our evaluation with Stack Overflow artifacts, we observe that sentence-level filters do not provide substantial changes.

When we compare evaluation metrics to a `baseline` using VSM, we find that the baseline achieves precision and recall scores of 0.30 and 0.33 for the same data. Although this result is not surprising, it suggests that a developer using keyword-matching could miss much of the task-relevant textual information in an artifact.

Table 5.5 also details evaluation metrics artifact-type wise. Certain techniques, such as `word2vec` with `frame-elements`, have better results in specific types of artifacts, such as GitHub issues. However, `word2vec` results are significantly lower for other types, such as API documentation or Stack Overflow answers. Without filters, `BERT` performs better at Stack Overflow. For this technique, we also find that filters decrease the amount of variation in the evaluation metrics. Notably, `BERT` with `frame-associations` is the technique with the highest recall and second lowest standard deviation suggesting that the technique performs well across different types of artifacts.

When we compare techniques without any filters, differences between `word2vec` and `BERT` may be explained by how each of these two approaches compute relevance. That is, all the words in a sentence contribute equally to `word2vec`'s identification of task-relevant text via its semantic similarity function. If an artifact lacks much of the text that appears in a task description, this technique may not

be able to identify the text deemed relevant. `BERT` shortens this gap because relevance is computed via its attention mechanism, which might assign more weight to words key to determining that the text is relevant to a task thus, explaining why the technique has better evaluation metrics.

With sentence-level filters, there are improvements to most of the types of artifacts evaluated. However, in certain techniques, the filters decreased the text identified, as when using the `frame-elements` filter in Stack Overflow artifacts. It is possible that the text in these artifacts lacks any of the frames that we defined as meaningful, and thus, the reason why this filter reduced the amount of text identified.

Results from our evaluation across all artifact types indicate that semantic techniques can find up to 58% of the task-relevant textual information in an artifact. Some of our techniques also show consistent results over different types of artifacts, suggesting that semantic techniques generalize across a variety of software artifacts.

		no filter		*w/ frame-elements*		*w/ frame-associations*	
	artifact	**precision**	**recall**	**precision**	**recall**	**precision**	**recall**
word2vec	API documentation	0.47	0.39	0.47	0.40	0.53	0.52
	GitHub issues	0.40	0.36	0.60	0.54	0.44	0.36
	Stack Overflow answers	0.43	0.48	0.49	0.45	0.45	0.50
	Miscellaneous pages	0.49	0.49	0.50	0.45	0.50	0.47
	overall	0.45	0.43	0.51	0.46	0.48	0.46
	standard deviation	0.03	0.05	0.05	0.05	0.03	0.06
BERT	API documentation	0.52	0.55	0.51	0.57	0.52	0.55
	Github issues	0.52	0.56	0.53	0.55	0.52	0.56
	Stack Overflow answers	0.58	0.63	0.53	0.58	0.58	0.62
	Miscellaneous pages	0.52	0.56	0.51	0.54	0.53	0.57
	overall	0.54	0.58	0.52	0.56	0.54	0.58
	standard deviation	0.02	0.03	0.01	0.01	0.02	0.02

Table 5.5: Evaluation Metrics Over All Artifacts and Artifact-wise

5.3.4 Threats to Validity

We rely on the golden data for $\text{DS}_{android}$ to evaluate techniques, which impacts the conclusions we draw. This golden data in turn relies on the human annotations of text in artifacts that is related to a task-at-hand (Chapter 4). There is a possibility that this text marked by the annotators might not contain information associated with the solution for the task, or that an annotator missed highlighting text that they deemed relevant. We minimize this threat by considering only the sentences marked by at least two annotators as the golden data used in the evaluation of techniques. We also asked annotators to write a short set of instructions for the tasks they annotated and provided them an in-house tool that streamlined annotation procedures. These procedures help mitigate risks to the validity of our conclusions. We note that there are no objective means to quantify which text assists task-completion [162, 163].

The internal validity of the evaluation may be impacted by the differing number of types of artifacts in the data set. While most of the tasks in the data set have associated Stack Overflow artifacts, not as many have artifacts from GitHub. These differences may affect the training and testing of each technique. These risks are mitigated in that the content of Stack Overflow and GitHub artifacts is similar in length in the data set.

Other internal threats arise from the BERT pre-trained model we used. There are base models trained on text from Wikipedia [57], online news [145], or source code [64], to cite a few, and we could have explored how usage of these different models could impact the accuracy or our techniques. Nonetheless, we decided to use the BERT base model as published on the model's original paper [57] to establishing a baseline for future comparisons. After fine-tuning, the model we used outperformed the word2vec technique with software engineering embeddings [61] and we believe that using the BERT base model did not represent a risk to our study. Nonetheless, future research should consider how this model compares with other general or software-specific pre-trained models.

A second model-related threat concerns how we divided our data for training and evaluation purposes. We split the Android tasks and their associated artifacts in two equal parts using 25 tasks for training and 25 other tasks for evaluation.

This division might have affected BERT's overall accuracy and it would have been interesting to explore other data splits. We refrained from investigating other data splits because we were interested in testing our approach over a number of tasks with Stack Overflow artifacts so we could compare it to AnswerBot (Section 5.3.2) and other splits could have affected this evaluation.

The selection of tasks in the Android development domain could affect the generalizability of our work. Most notably, aspects such as programming languages, frameworks, associated technologies, and others [23] influence the information sought by a developer as well as what they find relevant. We mitigate this threat by reporting how one of the most most promising semantic-based techniques identified as part of our evaluation applies to unseen tasks and artifacts associated with the Python programming language (Chapter 6). The three annotators that we recruited to construct the $\text{DS}_{android}$ dataset are also not representative of the entire developer population.

5.4 Summary

In this chapter, we introduced six semantic-based techniques that incorporate semantics of words and sentences to identify task-relevant text across a range of natural language artifacts. We compare our proposed techniques to a state-of-the-art technique, AnswerBot, specific to Stack Overflow artifacts and we evaluate them using a dataset that comprises 50 software tasks about Android development for which human annotators identified pertinent text per task across a variety of kinds of software artifacts. Evaluation results show that semantic-based techniques achieve recall comparable to AnswerBot, but without the need for artifact-specific data, and that some of our proposed techniques perform equivalently well across multiple artifact types.

Chapter 6

Evaluating an Automated Approach to Task-Relevant Text Identification

In the last chapter, we showed that semantic-based approaches can help identify text in artifacts relevant to a task. In this chapter, we consider whether the identified best approach—*BERT with no filters*—can assist a software developer while they *work* on a task.

For that, we embed the approach into a web browser plug-in that automatically highlights text relevant to a particular software task in web pages inspected by a developer. We investigate benefits brought by using this tool, which we call TARTI, through a controlled experiment, thus addressing whether developers can effectively complete a software task when provided with task-relevant text automatically extracted from natural language artifacts. We report how 24 participants completed three Python programming tasks with and without TARTI ~~where~~ results show that participants considered the majority of the text automatically identified in the artifacts they perused useful and that the tool assisted them in producing a more correct solution for one of the experimental tasks.

We start by outlining the evaluation approach (Section 6.1). We then detail experimental procedures (Section 6.2) before reporting results from the experiment (Section 6.3). Section 6.4 concludes the chapter.

6.1 Motivation

Our goal is to examine how TARTI—a tool that automatically identifies task-relevant text in pertinent documents—can affect a developer's work. Building on work presented earlier in this thesis, the tool uses a semantic-based technique. By identifying information that is useful to the developer's task, a developer's burden to find task-relevant information [155] can be lowered, allowing them to focus their time on other activities such as judging how the information found applies to their task.

To be helpful, TARTI must direct a developer's attention to text that assists them in completing a task. If the tool is successful in identifying text useful to the task, we hypothesize that the developer will produce a correct solution more often than if they had not used the tool.

Even if a developer is more successful with the tool than without, there is a chance that the text shown by TARTI is not *useful*—either because it is not relevant for the task at hand or because it is unsurprising, i.e., a developer finds the text identified by the tool as "common-knowledge" [53, 155]. Our experimental design incorporates the gathering of qualitative data to assess the usefulness of the text identified.

6.2 Experiment

We seek to evaluate how a semantic-based tool, i.e., our web browser plug-in, might assists a software developer perform a task. We consider three questions:

*RQ*1 Does using of a semantic-based tool—TARTI—help developers produce more correct solutions?

*RQ*2 How useful is the text identified by TARTI, i.e., does automatically-identified text contain information that assists task completion?

*RQ*3 How does the text automatically identified by TARTI compare to text that humans perceive as relevant to a task?

To answer these questions, we designed an experiment where 24 participants with software development backgrounds each attempted a *control* and *tool-assisted*

task randomly drawn from a list of well-known Python programming tasks. Participants are asked to write a solution for both of their assigned tasks and we use the control task to collect what text a participant deems relevant to the task at hand. In the tool-assisted task, we gather input on the usefulness of the text automatically identified and shown by our tool. This design, summarized in Figure 6.1 and detailed in the following subsections, allow us to:

- assess the correctness of the solutions for each tasks performed *with* or *without* tool support, thus addressing *RQ1*;
- discuss the usefulness of the text automatically identified according to the feedback provided by the participants, which helps us answer *RQ2*, and;
- compare manual and tool identified task-relevant text, which addresses *RQ3*.

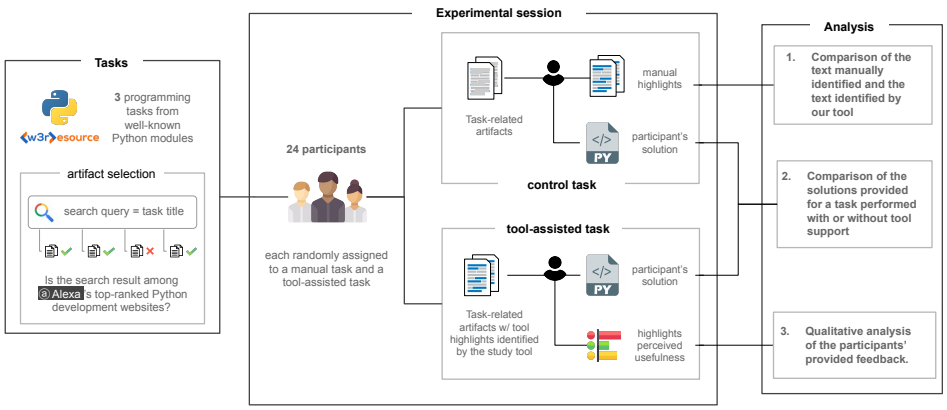

Figure 6.1: Summary of experimental procedures

The UBC ERB approved this experiment under the certificate *H19-04054*. The experiment's supplementary material is also publicly available [**?**].

6.2.1 Tasks

We opted for an experiment with tasks that could be completed by participants on their own time and computer. This decision was motivated by the COVID-19 pandemic and challenges related to recruiting participants and conducting an

Task	Description
Practice task	Given three dictionaries representing address books, you must write an algorithm using the Python core `dict` module to merge them.
Distances	Given a string representing a rendezvou point and a list of suggested picnic addresses you must write an algorithm using the `geopy` module to find the picnic address closest to the rendezvou point.
NYTimes	Given a string representing the url for NY Times Today's, write an algorithm using the `BeautifulSoup` and `requests` modules to srape all the headlines of that page.
Titanic	Given a string representing a url for the titanic dataset, you must write an algorithm using the `pandas` and `seaborn` modules to create a barchart of the data.

Table 6.1: Python tasks

in-person experiment [158, 159]. Since participants would follow instructions on their own, we decided to use tasks that are easy to understand and perform in a single experimental session, but still required a participant to seek information in artifacts associated with each task.

Table 6.1 details the tasks that we have selected based on task selection procedures from related work that meet these criteria [179]. These tasks were drawn from Python w3resource[1] tasks that require usage of at least one module external to the Python core library. By using external modules, we aim to reduce the likelihood that a participant can provide a solution for a task without consulting any of the artifacts (Section 6.2.2) that detail each of the modules associated with each task.

Figure 6.2 provides an excerpt of the information shown in a task[2]. For each task, participants had the task description and examples of input and output scenarios at their disposal. A task contained a list of resources that participants could consult so that they could write their solution. Each task also contained a link to an online coding environment (Section 6.2.3) where a participant could write and test their solution.

[1] https://www.w3resource.com/python-exercises/

[2] Full descriptions are available in the experiment's supplementary material [**?**].

6.2.2 Artifacts

Each task requires a set of artifacts that a participant could peruse for information that could assist them in writing their solution. Ideally, participants could find these artifacts on their own. However, our need to compare solutions between participants who perform a task assisted by our tool and without it as well as our need to compare the text that participants deem relevant to the text automatically identified by our tool means that all participants must have the exact same artifacts for a task.

Therefore, we follow procedures similar to the ones we used to create the $\text{DS}_{android}$ dataset to produce the list of artifacts for each of the tasks in Table 6.1. ~~That is, w~~e use the Google search engine to obtain up to ten artifacts that likely contain information that could help a participant correctly complete that task. Three pilot runs ensured that the artifacts collected using such procedures had sufficient information to complete a task without the need of additional resources. Based on these pilots, we simplified the description of the `distances` tasks by removing the need to sort the list of suggested addresses, which also led to the removal of two artifacts associated with sorting. Table 6.3 details the final list of artifact types per task.

Task	**Artifacts**	**#**	**Task**	**Artifacts**	**#**
Distances	API documents	2	NYTimes	API documents	3
	Stack Overflow posts	3		Stack Overflow posts	3
	Miscellaneous web pages	3		Miscellaneous web pages	4
Titanic	API documents	4	Practice*	API documents	1
	Stack Overflow posts	3		Stack Overflow posts	2
	Miscellaneous web pages	3			

* smaller number of artifacts due to it being a practice task;

Table 6.2: List of artifact types per task

6.2.3 Coding environment

To ensure that participants had the same conditions to perform each task and also to minimize setup instructions, we used Google Colab[3] as our coding environment. Colab is a product from Google Research that allows people to write and execute Python code through their browser [10].

Colab provided participants with a code editor with features commonly found in modern IDEs, e.g., code completion and syntax highlighting. It also ensured that all the participants performed the tasks in the same Python version and it lifted burdens that could arise from installing dependencies associated with the external modules used in each of our tasks.

Figure 6.3 shows an example of the Colab coding environment. First, it handled dependencies management and then, it presented a class containing a single method with a `TODO` block where participants should write their solution. The environment also provided a main function where participants could see the output of their code. Alternatively, a participant could use test cases to test their solution against the examples shown in each task description.

6.2.4 Participants

We advertised our study to professionals developers and to computer science students at several universities. Our target population comprised professionals and third, fourth-year or graduate students. We expected participants to have experience in object-oriented programming languages, and to consult API documentation when performing a programming task. We gathered this background information as part of our demographics (Figure 6.4) and no participants were excluded based on their background.

We obtained twenty four responses to our study advertisement (3 self-identified as female and 21 as male). At the time of the experiment, 10 participants were working as software developers and 14 were students (11 graduate and 3 undergraduate). The majority of the students (71%) also reported having some previous professional experience.

On average, participants self-reported 8 years of programming experience ($\pm$

[3] https://colab.research.google.com/

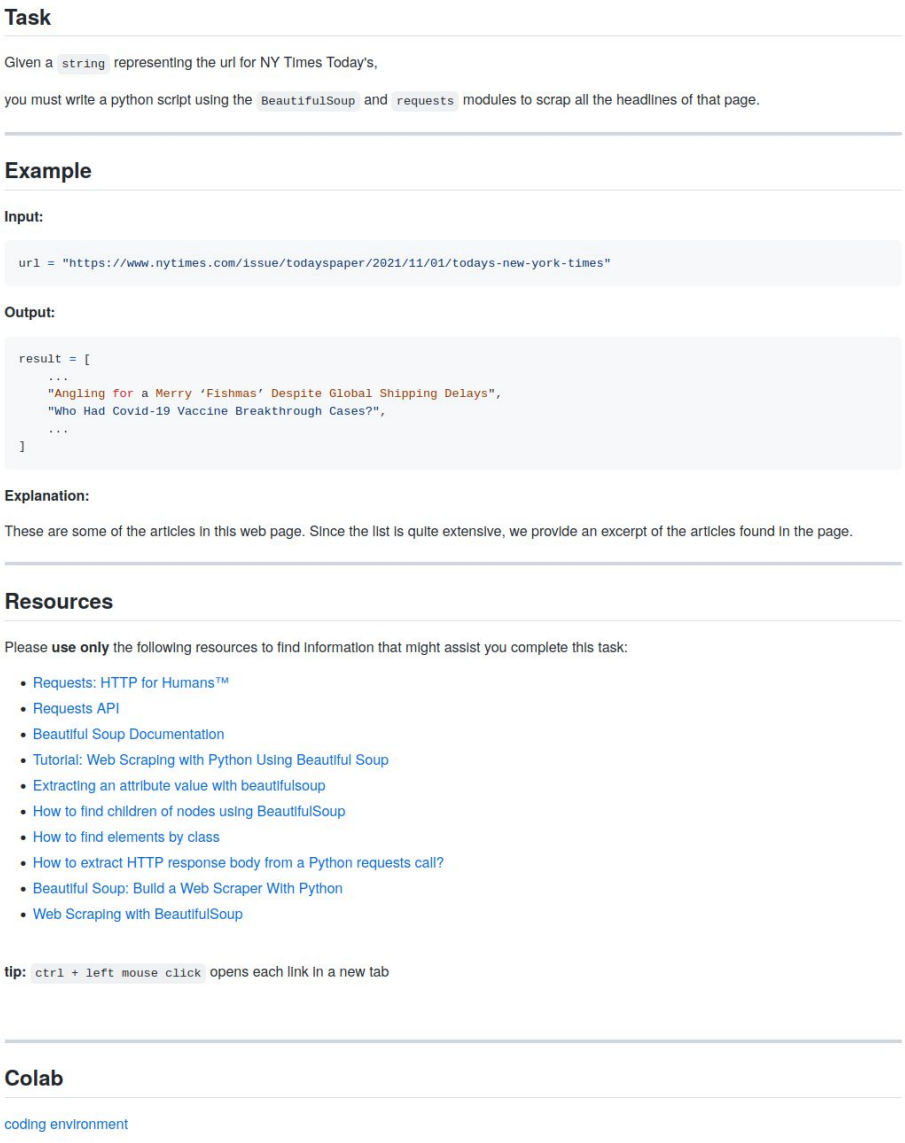

Figure 6.2: Information shown in a task

3.8, ranging from 3 to 17 years). The majority of the participants (54%) had between 5 to 10 years of experience in object-oriented programming languages, closely followed by participants with 3 to 4 years of experience (29%). Most of the participants also indicated that they did check API documents almost every time

Figure 6.3: Colab environment

they performed a programming task.

To which gender do you identify?

If you are a student, in which year of the course program are you at?

☐ 1*st* ☐ 2*nd* ☐ 3*rd* ☐ 4*th* ☐ 5*th*+ year ☐ *graduate student*

For how many years have you been developing software?

For how many years have you been developing software professionaly?

How many years of experience do you have in Object-Oriented programming languages? [a]

☐ *no experience* ☐ $(\infty, 1)$ ☐ $[1, 3)$ ☐ $[3, 5)$ ☐ $[5, 10)$ ☐ $[10, \infty)$

How often do you consult API documentation when performing a programming task?

(never) 1 - 2 - 3 - 4 - 5 *(always)*

[a] closed or open intervals notation

Figure 6.4: Background questions asked to a participant

6.2.5 Procedures

The entry point to our experiment was our advertisement email. The email disclosed the purpose of the experiment, eligibility criteria, an estimate of the time it would take to complete it as well as a link to a web survey containing the experiment's consent form and tasks.

Once a participant consented to participate, the survey gathered demographics and provided further instructions about how to perform each task, requesting them to install our tool, a web browser plug-in. Setup was followed by a short practice task—separate from the experimental tasks—that allowed participants to familiarize themselves with the content of a task, the tool, and the coding environment that we used (Colab).

Once a participant completed the practice task, the survey randomly assigned to them a *control* task, which was followed by a randomly assigned *tool-assisted* tasks—different from the control task. For each task, including the practice tasks, the survey provided to the participants a link to the task description (Figure 6.2) and asked them to submit a solution for the task, i.e., written Python code. While tasks were randomly assigned, we made sure that a similar number of participants attempted a task with and without tool support, as Table 6.3 shows.

Once a participant submitted their solutions, the survey asked them about any additional feedback that they wished to share and offered them the opportunity to enter a raffle for one of two iPads 64 GB to compensate them for their time, concluding the experiment.

Task	Configuration	Participants who attempted the task	#
Distances	*control group*	$P3, P4, P8, P12, P16, P17, P21, P22$	8
	with tool support	$P1, P5, P9, P13, P15, P18, P20, P23, P24$	9
NYTimes	*control group*	$P1, P2, P6, P10, P11, P14, P15, P20$	8
	with tool support	$P3, P7, P12, P16, P17, P19, P21, P22$	8
Titanic	*control group*	$P5, P7, P9, P13, P18, P19, P23, P24$	8
	with tool support	$P2, P4, P6, P8, P10, P11, P14$	7

Table 6.3: List of participants who performed each task

Control Task

In the *control* task, we use the web browser plug-in to gather text that a participant deems useful for the task at hand. In this task, the survey asked participants to use our tool to highlight sentences that they deemed useful and that provided information that assisted task completion—instructions similar to the ones used for the creation of the $\text{DS}_{android}$ corpus (Chapter 4).

Figure 6.5 gives insight into how participants highlighted sentences. Whenever a participant inspected one of the artifacts available for their task, they could click on the `highlight` button in the tool's context menu. This would then instrumented the HTML of the page identifying individual sentences. A participant could hover over identified sentences and select them as relevant by clicking on the hovered text. Once a participant had finished selecting sentences, they could submit their data also through the tool's context menu.

Participants could highlight text any time before submitting their solution. Based on observations from the pilots, the most common strategy we noticed was that of highlighting text on-the-fly, i.e., as a participant performed the task and consulted its artifacts, they would highlight text while reading each document. Nonetheless, some participants shared that they performed their highlights just before finishing a task.

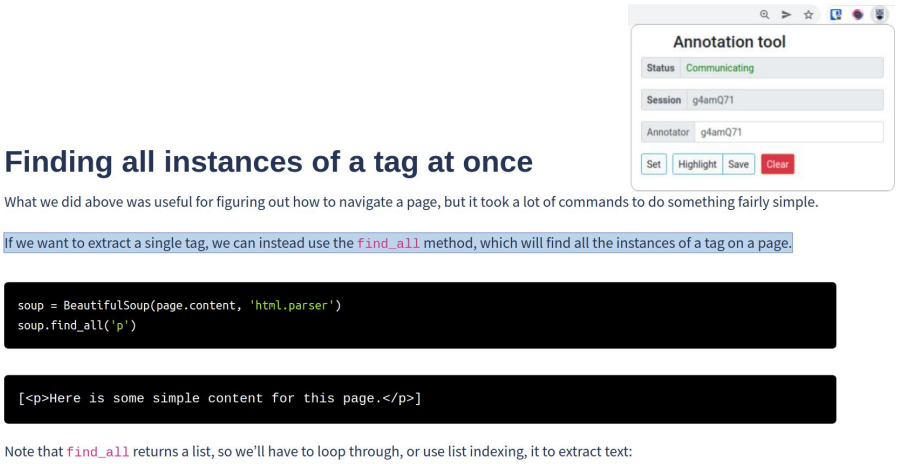

Figure 6.5: Study's tool context menu (top-right corner) and a sentence highlighted by a participant

Tool-assisted Task

In the tool-assisted task, our tool automatically highlighted text that its underlying semantic-based technique identified as relevant to the participant's task[4]. The highlights were shown in a format similar to the one in Figure 6.5, but without the need for any actions by a participant.

For this task, when a participant submitted their solution, we asked them to rate on a 5-point Likert scale [111] how helpful were the highlights shown by the tool. Figure 6.6 shows an example of how we gathered data about the usefulness of the text automatically identified. Participants rated highlights on a per artifact basis. We gather input at the artifact level because it would be too demanding for a participant to provide individual feedback on each of the highlights shown in the time that we estimated and that we advertised for the study. Section 6.3.5 discusses threats that arise from this decision.

[4]We configure the tool to identify no more than 10% of the sentences of an input artifact as relevant. This number was determined based on the average number of sentences indicated as relevant by human annotators in the artifacts of the $DS_{android}$ corpus.

1. Indicate whether you agree with the following statement:

The highlights in "How to extract HTTP response body from a Python requests call" were helpful to correctly accomplish the task in question.

(Strongly disagree) 1 - 2 - 3 - 4 - 5 *(Strongly agree)*

2. Indicate whether you agree with the following statement:

The highlights in "BeautifulSoup tutorial: Scraping web pages with Python" were helpful to correctly accomplish the task in question.

(Strongly disagree) 1 - 2 - 3 - 4 - 5 *(Strongly agree)*

...

Figure 6.6: Questions asking a participant to rate the usefulness of the highlights shown in two artifacts; by clicking on the name of an artifact, a participant could revisit the highlights of that artifact

6.2.6 Summary of experimental procedures

We have described experimental procedures where participants attempted two programming tasks each. These procedures allowed us to gather:

1. a participant's submitted solution (written Python code) for each task;
2. text that participants deemed relevant in the artifacts of the control task;
3. the usefulness of the highlights shown in a tool-assisted task; and
4. any additional feedback (written text) that a participant wished to provide.

We use this data to investigate whether a tool embedding a semantic-based technique helps developers complete a software task.

6.3 Results

We organize results by assessing the solutions submitted by the participants and their ratings on the usefulness of the highlights shown in each artifact of each task. We also compare manually and automatically identified text.

6.3.1 Tasks Correctness

When assisted by a tool that automatically highlights text identified as relevant to a task, we expect that a developer can produce a solution that is equally or more correct than the solution of a developer who attempted a task without tool support.

Metrics

To compute how correct a participant's solution is, we compile their submitted code and run it against a set of 10 test cases (not provided to participants) that check whether it produces the correct output for each given test input. Hence, *correctness* represents the number of passing test cases of a solution (Equation 6.1). For example, if the solution of a participant passes 7 out of 10 test cases, we would assign a correctness score of 7 to this solution. A solution with compilation errors has a correctness score of 0.

$$Correctness = \#\ of\ passing\ test\ cases \tag{6.1}$$

Data

From all submitted solutions (24 manual and 24 tool-assisted), two solutions from tool-assisted tasks had compilation errors or failed all test cases. One of these was from a participant who indicated that they decided to not finish their tool-assisted task due to time constraints; the other, from an exception thrown in the code of a participant who misused the `geopy` module. We do not ignore these two solutions when reporting results.

Results

Figure 6.7 aggregates all correctness scores for tasks done with and without tool support. We compare correctness scores first using a Shapiro-Wilk test [187] to check for normality and then, due to deviation from a normal distribution ($p\text{-}value < 0.05$), applying a Wilcoxon-Mann-Whitney test [122]. Results from this test indicate that we cannot draw statistically significant conclusions for the solution scores

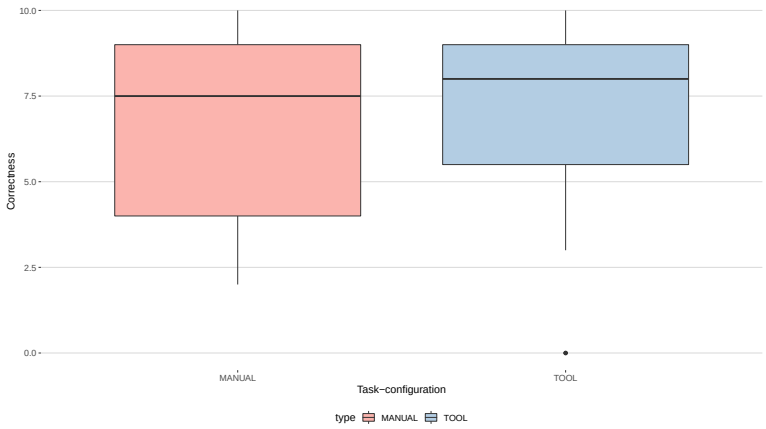

Figure 6.7: Boxplots showing aggregated correctness scores for task performed with and without tool support

of the two groups ($p\text{-}value \geq 0.10$)[5]. Hence, we evaluate scores on a per-task basis, as shown in Figure 6.8.

Comparing the correctness scores for each individual task, we observe that participants with tool support performed worse in the `distances` task. Among potential reasons for the lower score, we believe that the text automatically identified might not have been relevant for this task, what can explain the higher percentage of participants who indicated that the text identified by the tool was not helpful (Section 6.3.2).

The two groups have the same correctness scores in the `NYTimes` tasks. We found that the most notable differences in the solutions of this task arise from participants who did (or not) consider corner-case scenarios. To illustrate this, we quote one participant who stated that their solution could "*fetch all the articles, but it could also get some noise*", e.g., getting all the headlines of the web page but also an unrelated ad, which is indeed one of the test cases for this task.

The `titanic` task required using two data science modules and it is the one with the most differences. Participants who did the task with tool support were able to produce more correct solutions, with an average correctness score of 6. In

[5]As Section 6.3.5 details, the lack of statistical power is not surprising and is a common risk to between groups comparisons [104].

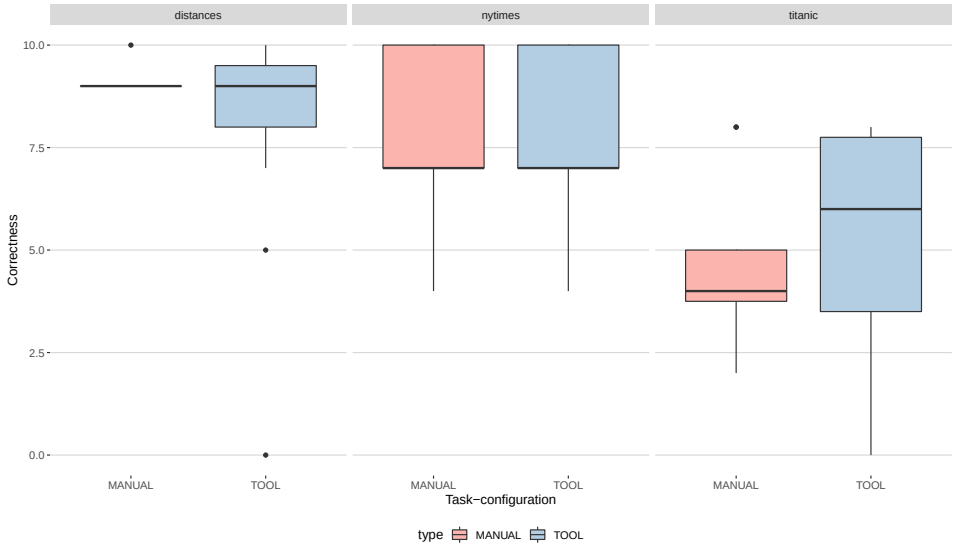

Figure 6.8: Boxplots showing correctness scores for each task performed with and without tool support

contrast, participants in the control group had an average correctness of 4. As we detail in the next sections, we found that the tool identified much of the text that the participants deemed relevant and that participants also indicated that the text automatically identified was indeed useful.

6.3.2 Usefulness Analysis

Having compared the correctness of manual and tool-assisted tasks, we turn to the question of whether the highlights shown by the tool were considered helpful. For that, we analyze participants' ratings and the feedback that they provided at the end of our experiment.

Metrics

To investigate the usefulness of the highlights shown by our tool, we asked participants to indicate on a 5-point Likert scale whether the highlights of each artifact were helpful to correctly accomplish their assigned task (Figure 6.6). We aggregate individual responses to measure how useful the tool was in assisting developers

complete each task in our experiment, plotting responses using a diverging stacked bar chart [172].

Data

Participants produced a total of 197 ratings representing the usefulness of the highlights in the artifacts that they inspected. On average, we collected 65 responses per task and 7 responses per artifact. These values do not match the exact number of participants and artifacts in our experiment since some participants did skip this part of the survey for their own reasons.

We also obtained written feedback from 19 out of the 24 participants, divided on the feedback of the tasks (24 data points) or of the experiment itself (15 data points). We use this feedback to quote scenarios that support our observations.

Results

From all the ratings collected on the usefulness of the text automatically identified and shown by our tool, 40% of them agreed that the highlights were useful, 25% neither agreed nor disagreed on their usefulness, and 35% indicated that they were *not* useful. Similar to how we presented results on correctness, we analyze usefulness ratings on a per-task basis.

Figure 6.9 shows participants' ratings aggregated for each task. Participants indicated that the highlights shown for the `titanic` task were the most useful. Ratings for this task support our observations on the correctness of the solutions produced. Notably, one participant indicated that highlights for this task assisted them in identifying essential function arguments needed to use the `pandas` group by function. In contrast, participants indicated that the highlights for the `distances` task were in its majority not useful. For example, one participant pointed out that, in the `geopy` API documentation, there were no highlights in a section where they would have expected otherwise.

The mixed results for the `NYTimes` task might explain the lack of differences in the correctness scores observed for this task. Although anecdotal, one interesting feedback for this task was from a participant who described that their experience with the `BeautifulSoup` module influenced their negative ratings—"*I have ex-*

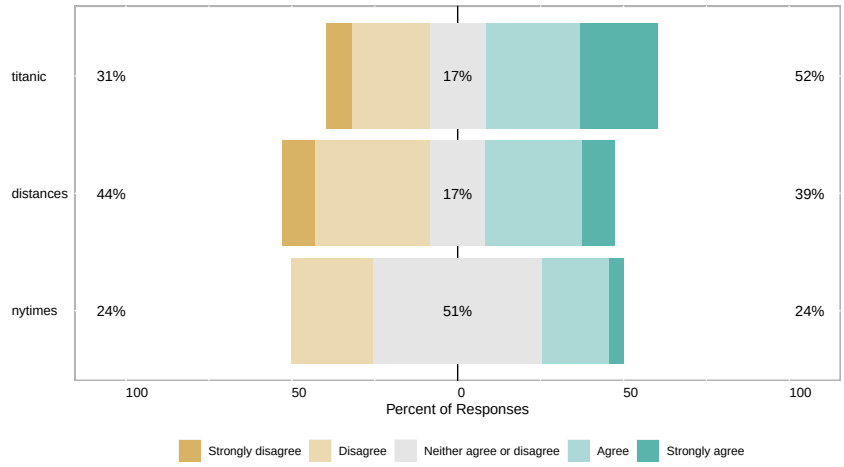

Figure 6.9: Diverging stacked plot of the usefulness of the text automatically identified for each task

perience with BeautifulSoup and webscraping, and so my [negative] ratings of the usefulness of the highlights may have been influenced by that".

Surprisingly, when we look at the ratings per type of artifact (Figure 6.10), API documents had the most useful highlights. For this type of artifact, we observe that positive ratings originate from artifacts that follow a *'how to'* format, which is not conventional for API documents [20, 156]. The highlights on Stack Overflow artifacts were also perceived as useful and participants expressed familiarity with this type of artifact, where the highlights helped them determine parts of the page to ignore—"*the highlights [on Stack Overflow] helped me quickly determine which parts of the page to ignore*".

Miscellaneous web pages were the artifact where participants disagreed the most on the usefulness of the text automatically identified. Due to their '*tutorial*' format, these artifacts are lengthy and some participants expressed that they would like more direct explanations about how to use the modules of each task—"*I have used these libraries before to some extent, so I don't need a narrative around purpose or procedure*".

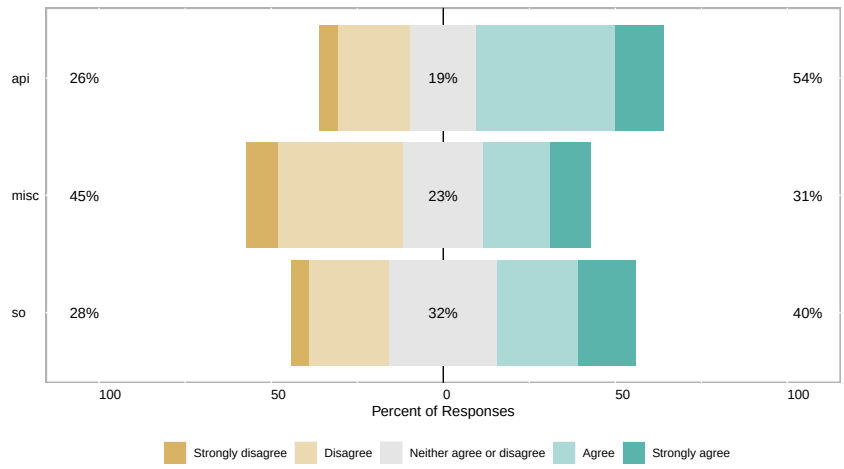

Figure 6.10: Diverging stacked plot of the usefulness of the text automatically identified for each type of artifact

6.3.3 Comparison of manual and automatically identified task-relevant text

To assist a developer to complete a task correctly, a tool that automatically identifies text pertinent to that task might identify text that humans have also considered as relevant. Procedures from our control task asked participants to identify text they deemed useful. We compare this manually provided data against the text automatically identified.

Metrics

To investigate the overlap between the participants' manual highlights and the the automatic highlights identified by TARTI, we use *precision*, *recall* [123], and *pyramid precision* [139]. We compute these metrics for each artifact of each task and report their average.

For this analysis, we follow Lotufo et al.'s procedures [119] and we consider any text marked by any participant as relevant. We investigate if our tool automatically identifies text that multiple participants deemed relevant via *pyramid precision*. Details for each metric are as follows.

Precision measures the fraction of the text automatically identified that partici-

pants deemed relevant (Equation 6.2).

$$Precision = \frac{automatic\ highlights\ \cap\ manual\ highlights}{automatic\ highlights} \tag{6.2}$$

Recall represents how many of all the manual highlights were identified by the semantic-based technique applied by our tool (Equation 6.3).

$$Recall = \frac{automatic\ highlights\ \cap\ manual\ highlights}{manual\ highlights} \tag{6.3}$$

Pyramid precision compares the text automatically identified to an optimal output, i.e., one where—for the same number of sentences—we identify sentences selected by the most number of participants (Equation 6.4). The more we identify text that more participants indicated as relevant, the higher pyramid precision is.

$$\triangle Precision = \frac{weight(automatic\ highlights\ )}{weight(optimal\ highlights)} \tag{6.4}$$

To illustrate these metrics, consider an artifact with 4 sentences $\{s_1, s_2, s_3, s_4\}$ that have been selected by $\{2, 0, 1, 1\}$ participants, respectively. That is, s_1 was selected by 2 participants, s_2 by 0, and so on. Table 6.4 shows precision, pyramid precision, and recall metrics in a scenario where we output sentences $\{s_2, s_3\}$ as relevant.

metric	*formula*	*result*
precision	$\frac{\{s_2,s_3\} \cap \{s_1,s_3\}}{\{s_2,s_3\}} = \frac{1}{2}$	0.5
recall	$\frac{\{s_2,s_3\} \cap \{s_1,s_3\}}{\{s_1,s_3,s_4\}} = \frac{1}{3}$	0.33
$\triangle$ *precision*	$\frac{weight(s_2)+weight(s_3)}{weight(optimal)} = \frac{0+1}{3}$	0.33

Table 6.4: Example showing how we compute precision, recall and pyramid precision metrics

Data

Participants who indicated what text was relevant to their assigned control task produced a total of 415 highlights with an average of 7 highlights (std ± 3) per artifact inspected. On average, this comprises 9% of the entire content of the artifacts in our experiment.

Some participants also selected code snippets as relevant to a task—a threat that we discuss in Section 6.3.5. Code snippets account for 30% of the highlights produced, but we remove them from our analysis since TARTI operates on text only. For the textual highlights, Krippendorf's alpha indicates good agreement of what text in an artifact participants deemed relevant ($\alpha = 0.68$) [96, 144]. We compare these manually produced highlights to the text automatically identified by our tool.

To help future research in the field, we bundle the three Python programming tasks in our experiment, its associated natural language artifacts, and the text that participants indicated as useful in a corpus named DS_{python} [**?**].

Results

Table 6.5 summarizes the average of precision, pyramid precision, and recall metrics for each of the tasks in the experiment. Precision scores range from 0.55 to 0.68, while pyramid precision scores range from 0.55 to 0.57, which suggests that our tool failed to identify some of the text that participants deemed the most relevant. These results also corroborate the correctness scores detailed in Figure 6.8. For example, in the `distances` task, participants who performed the task with tool support had solutions less correct than participants in the control group. This was the task with the lowest precision, recall and pyramid precision values. In contrast, the task where participants assisted by our tool obtained the best correctness scores, `titanic`, is the one with the best precision, recall and pyramid precision values.

Table 6.6 details evaluation metrics artifact-type wise. Stack Overflow posts and API documentation have the highest precision scores. For these types of artifacts, pyramid precision indicates that the text automatically identified on Stack Overflow was the text that several participants deemed relevant. The same does not apply to API documentation, i.e., our tool failed to detect a portion of the text

	precision	△ precision	recall
Distances	0.55	0.55	0.57
NYTimes	0.59	0.57	0.58
Titanic	0.68	0.57	0.60
overall	0.60	0.56	0.58

Table 6.5: Evaluation metrics per task

	precision	△ precision	recall
API documentation	0.65	0.55	0.59
Stack Overflow posts	0.66	0.62	0.63
Miscellaneous web pages	0.53	0.53	0.54

Table 6.6: Evaluation metrics per type of artifact

that many participants deemed relevant. Miscellaneous web pages were the artifact type with the lowest scores. This results support our findings on the usefulness of text per type of artifact.

We also compare evaluation results to the ones in Chapter 5. This comparison is interesting because it let us cross-examine our findings with new data [60, 165]. Figure 6.11 presents boxplots for precision and recall of the BERT technique with no filters for the tasks in the $\text{DS}_{android}$ corpus and the Python tasks (experiment). We observe that the technique's accuracy is comparable across the two evaluations.

6.3.4 Summary of results

The preliminary results we obtained suggest that an automatic approach to task-relevant text identification assists developers in completing a software tasks. Participants found the text automatically identified useful in two out of the three types of artifacts that they inspected, namely API documents and Stack Overflow posts, and; for the task that required using more modules and functions (`titanic`), participants who used our also produced more correct results.

6.3.5 Threats to Validity

Our experiment compares solutions submitted by participants who attempted each task with and without tool support. This represents a between groups design [104]

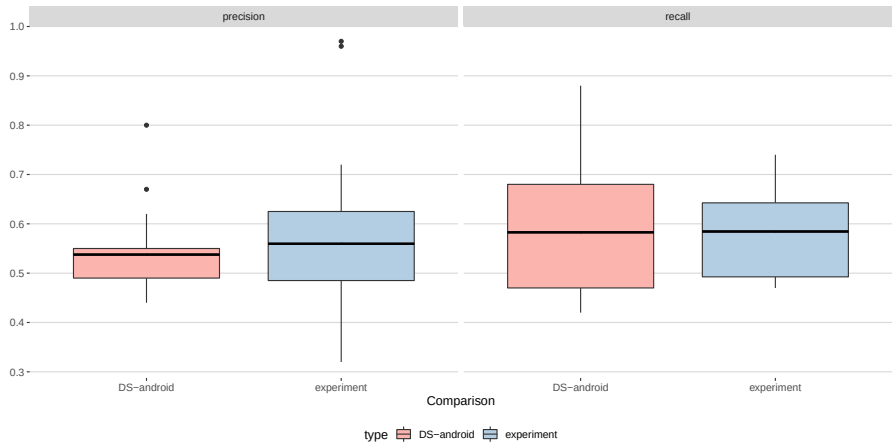

Figure 6.11: Boxplots comparing precision and recall values when applying our semantic-based technique to the DS_{python} and the $\text{DS}_{android}$ corpora

and we discuss threats inherent to it.

Since we compare results from different participants, our analysis might be subject to substantial impact from individual differences [104]. For example, participants who performed a task with tool support may have been more experienced than participants who did the same task without tool support what affects correctness scores. As another example, participants' skill and background influences the text that they indicate as relevant in the control task as well what text they perceive as useful in the tool-assisted task. We minimized these threats by recruiting participants of varied background and randomly assigning tasks to each participant.

The tasks in our experiment impact generalizability. Although we opted for simple tasks, we ensured they modules used in our tasks were representative. For example, we found open-source systems[6] using `BeautifulSoup` with function calls similar to the ones needed to complete the `NYTimes` task. Nonetheless, there are clear differences in the artifacts one can gather based on the domain or programming language of a task [26]. Hence, we consider other domains and a wider range of task and artifacts for future work.

The selection of tasks also affects our conclusions. We opted for Python programming tasks that required writing code, which we use to assess correctness.

[6] https://github.com/ArchiveBox/ArchiveBox/issues/18

As observed by other researchers [132, 164], developers work on many different tasks, some of which focus on code [131] while others on information seeking [77], e.g., finding duplicated bug reports or researching visualization libraries to identify the most suitable one [164]. Had we decided to use information-seeking tasks, participants could have produced a different set of highlights, perhaps selecting fewer code snippets. Given that our experiment was completely remote, instructing participants on how to perform information-seeking tasks would have been more difficult. Furthermore, objectively judging their correctness would also be more strenuous, which would lead to a different experiment with challenges and risks of its own.

The fact that we consider the text marked by any participant as relevant also affects our conclusions. We refrain from excluding text selected by a few participants from our analysis for reasons similar to the ones in our characterization of task-relevant information (Chapter 3). That is, the text marked by these participants might still contain valuable information. We minimize this threat by reporting both precision and pyramid precision, where we observe that our approach failed to detect the text that multiple participants deemed relevant for some tasks or types of artifacts.

Concerning the text automatically identified by our tool, we gather usefulness at the artifact level. Suppose we had gathered usefulness at the sentence level. In that case, we could have used this information to further refine our analysis, for example, reporting precision and recall at different usefulness levels or computing accuracy based on the participants' input, as done by Xu et al [190]. However, asking participants to provide feedback at the sentence level would have considerably increased the time we estimated that the experiment would take, which would impact recruitment. We weighed the benefits and drawbacks of a fine-grained or more coarse-grained analysis, and we opted for the latter so that this would not be a barrier to people deciding on whether to participate in our experiment.

Chapter 7 futher discusses limitations or improvements to the semantic-based techniques and to our tool.

6.4 Summary

In this chapter, we presented an experiment to evaluate whether TARTI, a tool that embeds a semantic-based technique, assists a developer working on a software task. The experiment examined how 24 participants with software development backgrounds attempted two programming tasks with or without such a tool. Results from this experiment indicate that, participants found the text automatically identified and shown by our tool useful in two out of the three types of artifacts that assisted them completing their assigned tasks, where our automatic approach identified on average 58% of the text that participants deemed relevant. These results encourage further exploration of semantic-based techniques, embedding them into tools that ultimately facilitate a developer's work.

Chapter 7

Discussion and Future Work

In conducting the studies reported on in this dissertation and in developing the techniques explored, we made many decisions. In this chapter, we discuss the limitations and trade-offs that resulted and discuss future avenues for exploration.

7.1 Relevance

A primary assumption in this dissertation is that there will be text in documents associated with a software development task that is commonly seen as *relevant* to the task. Chapter 3 demonstrates that sufficient commonality of relevant text does exist to support the development of techniques and Chapter 6 shows that the automatically identified text is seen as relevant. For instance, some participants in the TARTI experiment indicated:

> "*The highlights were super useful, without them I would definitely had not been able to rapidly do the task.*"

> "*With the highlighted references, I was able to move much quicker. I quickly glanced at each resource, reading just the highlights to determine how valuable that resource was. The highlights allowed me to focus on the most relevant resources, gathering the necessary information to complete the task.* "

Yet, others who participated in the TARTI experiment disagreed with the concept that text in these documents was important at all:

> "*I realized going through [the experiment] that I read very little free-form text when looking for solutions. Mainly code samples with clear, succinct examples or type/method definitions.*"

> "*I'm not going to bother reading the text if I don't have to, especially when the code snippets are easy to understand.*"

This feedback suggests that the kind of information considered useful in a document not only differs based on a developer's *explicit* versus *implicit* reasoning, but also that it varies with the developer. It suggests that techniques will need to be even more general to work across document types: techniques will also need to work across the different kinds of information in a document. Future studies should more deeply investigate the kinds of information developers deem as relevant and explore how developers gauge relevance.

How and what developers consider relevant in documents may also impact how techniques trade-off precision versus recall. In Chapter 5, we indicated the techniques we explored favored locating all relevant text within an artifact, in other words recall, rather than focusing on correctly determining relevant text, in other words precision. We made this decision because, as described in (Chapter 1), missing relevant text would mean that a developer might have an incomplete or partial view of the information needed, which may lead to sub-optimal decisions.

However, when favoring recall, we may obtain more false positives; non-relevant text indicated as relevant. Due to the limited time developers spend inspecting a natural language artifact [174], there is a chance that a developer could discard reading an artifact due to a false positive. Although abandoning reading an artifact and moving to another artifact could lead to non-efficient work ~~(i.e., a developer might have to perform further searches and perhaps revisit artifacts that they have already inspected)~~, we believe that the benefits of automatically identifying relevant text outweigh these risks. That is, inspecting an artifact without tool support is more time-consuming than inspecting each sentence retrieved to judge their rel-

evance to the task at hand. Future studies could explore the ramifications of this trade-off in detail.

7.2 Technique Deployment

In Chapter 5, we have shown that semantic-based approaches identify task-relevant textual information across different types of artifacts without relying on assumptions about the nature of an artifact or its meta-data. However, we observe that the techniques with the best accuracy require fine-tuning a deep learning model and this could be considered as an impediment to deploying such techniques.

Fine-tuning requires training data—tasks, natural language artifacts, and text annotated as relevant—and one potential limitation arises from how the training data might lead the model to identify text that is not relevant to certain types of tasks. For example, with the adoption of Agile practices [69] and continuous delivery [84], there has been a rise in automated approaches for organizing and facilitating continuous delivery, which are commonly referred to as *DevOps* [105, 166]. DevOps tasks and natural language artifacts documenting DevOps tools might significantly differ from tasks and artifacts associated with bug fixing or implementing new features and, despite the fact that we observed that a model fine-tuned with 50 Android tasks was able to identify text relevant to Python tasks (Figure 6.11), it is not clear if a model fine-tuned with certain data is able to identify text relevant to tasks in a different context. Future research should more deeply investigate the extent to which tasks and artifacts affect the task-relevant text identified by a model that requires fine-tuning.

A second limitation to deploying the techniques we explored relates to the current cost associated with deep learning models. The neural embeddings and the neural networks we used often requires dedicated servers with significant memory and high-throughput computational power. Due to the high demand for commercial servers with such properties, it might be difficult to deploy our tools for usage a large scale. We believe that hardware cost might be eased in the future. *AM: I simplified the cost paragraph, but it might still be better to remove it*

7.3 Future Work

In this section, we elaborate some of the future research mentioned in Sections 7.1 and 7.2. We also discuss other avenues for future work.

7.3.1 Relevance Datasets

In Section 7.1, we indicated that future studies should further investigate what kind of information developers deem relevant and how developers gauge relevance. Upon reflecting on the data-gathering procedures described throughout this work, a potential line of work is to consider gathering data from software professionals performing daily tasks in a non-obstructive way.

Conducting empirical experiments in a more realistic environment is challenging [94]. This effort is worthwhile as the richness of collected data can provide valuable insights to provide a foundation for tool development. One potential method for data collection considers how recent studies with eye-trackers [52, 147, 167] have shown that the technology is not as disruptive as other methods [104] such as think aloud protocols or manual input. Hence, we believe that eye-tracking could be used in a developer's working environment, leading to more realistic data on which text developers perceive as task-relevant. For example, one could extend the work done by Kevic and colleagues on tracing developers' eye for change tasks [94] to also consider tracing data outside a developer's IDE for such a purpose.

A second venue is to consider who perceived which text as relevant. As observed both in related work [40, 50] and in our formative study (Chapter 3), there is variability in what text may be relevant to a software task and providing more background data about the participants who considered some information useful could lead to benchmarks for evaluating techniques focused on a certain population (e.g., expert versus novice developers [40, 50]).

7.3.2 Sentence Semantics

In Chapter 3, we used frame semantics, a general linguistic approach [67], as a means for access to the meaning of the text in natural language software artifacts. However, questions remain about whether semantic frames can help in identifying

the semantics of software engineering text and the extent to which it applies to software engineering text.

We partially addressed these questions in a study, orthogonal to this dissertation [128]. assessing the applicability of generic semantic frame parsing to software engineering text aimed at supporting program comprehension activities. In this study, we first assessed how the tool we used in Chapter 3 to analyze the meaning of the text considered relevant (i.e., SEMAFOR [54]) applies to text sampled from 1,802 documents drawn from a set of datasets [19, 44, 121, 190]. Based on the results from this analysis, we proposed *SEFrame*, a tool that tailors frame parsing to natural language text in software engineering artifacts. Results from a second evaluation indicated that *SEFrame* was correct in between 73% and 74% of the cases and that it parsed text from a variety of software artifacts used to support program comprehension, which motivated our decision to apply *SEFrame* in the design of the techniques we explored in Chapter 5. Nonetheless, future research should consider a more in depth investigation of frame semantics or other general approaches that can infer the meaning of text at the sentence level.

For example, future studies could focus on answering which frames most accurately capture the information conveyed in the text and how frames extracted by state-of-the-art tools (e.g., [47, 177]) compare to manual labels provided by researchers about the information available in certain kinds of software artifacts [19, 58, 121].

7.3.3 Deep Learning for Software Engineering

Future research in the field of deep learning for software engineering could consider exploring other pre-trained DL models as well as other applications of deep-learning to software engineering tasks.

In Chapter 5, we have discussed threats associated with the BERT pre-trained model we used and how future-research should consider other general or software-specific pre-trained models. We did not explore other general pre-trained models because most of the models that apply to our domain problem are variations of BERT (e.g., Albert [101] or Roberta [118]) and by using the BERT base model, we sought to establish a baseline for future comparisons.

With regards to software-specific models (e.g., [64, 107]), we also emphasize that these models have been trained on either source code or source code and method level text documentation. These datasets might not align with how developers discuss text in natural language artifacts [20], such as Stack Overflow posts or web tutorials. Therefore, future research should consider how models pre-trained on text originating from various kinds of natural language software artifacts compare to the base model that we used.

In Chapter 5, we have shown that artifact-specific techniques, such as AnswerBot, have accuracy comparable to the semantic-based approaches we have explored. This provides a unique opportunity for the creation of synthetic data that we can use to further fine-tune the deep-learning models we explored. For that, future research should show that models trained on synthetic data extend beyond what an artifact-specific technique used for bootstrapping already provides. For example, showing that a model fine-tuned using synthetic data can identify task-relevant textual information in a different type of artifact or showing that the model's accuracy is significantly better than the technique used for bootstrapping.

AM: Think whether I need anything else in this section

7.3.4 Presenting Task-Relevant Text

In Chapter 6, we described how TARTI highlights the text in a natural language artifact that it identifies as relevant to an input task. Our idea to highlight text draws from related work which suggests that textual highlights are a simple approach to surface the most important information in an artifact [137, 155]. Although simple, participants shared limitations of this strategy:

> "*It was much easier to follow with previously highlighted text. However, it would have been nicer to have some sort of side bar/index of highlighted snippets where I could know and scroll directly through the highlighted parts of a page.*"

> "*There was a resource page which was super long, and I found it very difficult to locate which sentences were highlighted, therefore, making that resource useless to me because I didn't have the motivation*

to scroll through it to find all highlights. An interface that gathers highlight locations would make a difference."

This feedback made us question other potential ways to present the text identified by TARTI. For example, we could have followed design principles adopted by Unakite [116]—a tool that collects, organizes and keeps track of information—to make TARTI display the text identified on its context menu. Figure 7.1 shows a mock up of TARTI bundling the highlights identified for one of the artifacts in the `titanic` task (Section 6.2.1). Using this context menu, a user could click on the text identified and navigate to the part of the documentation originally containing the text.

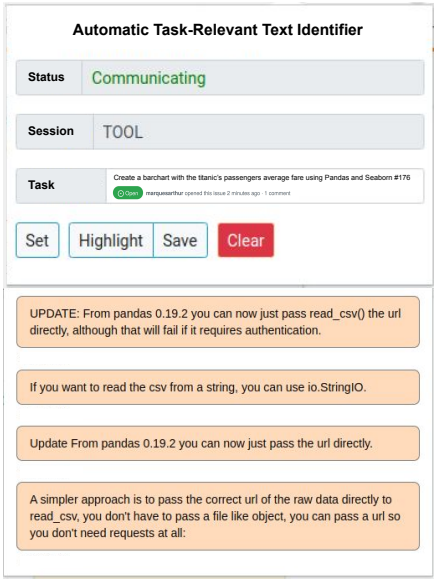

Figure 7.1: Mock up of the highlights identified by TARTI as navigational cues; by clicking on a highlight, a user could be directed to the part of a document containing that highlight

A second approach to organizing the text identified could consider the semantic meaning of each sentence, as extracted through frame semantics or other semantic-based approaches. Semantic frames could assist a user in comprehending the content of the text automatically identified without the need to read it. As done by

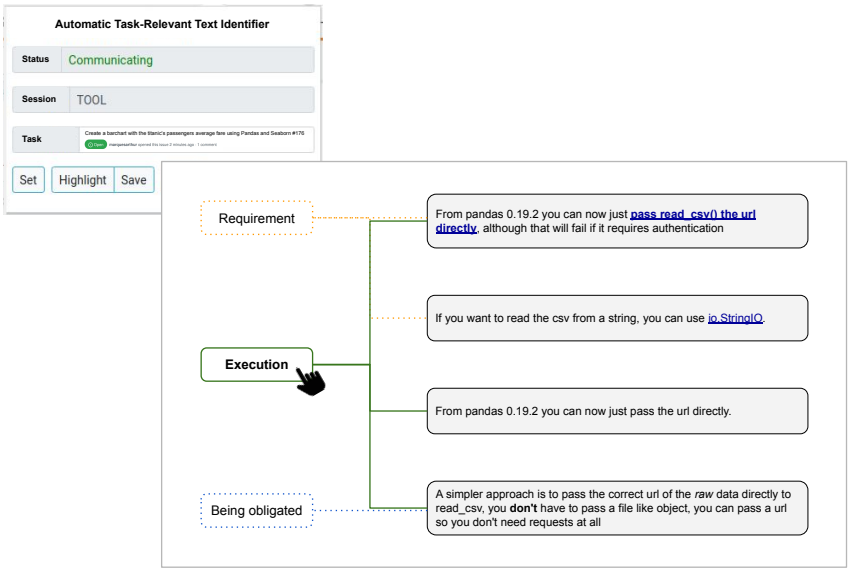

Figure 7.2: Mock up of the highlights grouped by semantic frames; by hovering over a semantic frame, a user could quickly identify which of the text identified is associated with a certain meaning

Libra [151], semantic frames could also be used to group the text identified in bubbles or a hierarchical representation helping a developer in deciding what portions of the text they would inspect first. Figure 7.2 shows a mock up with some of the semantic frames extracted for the sentences in Figure 7.1. Using the semantic frames identified, a developer could decide whether they would read sentences describing some coding procedure (*execution*), or sentences with warnings or requirements (*requirement* and *being obligated*) about the Pandas API. To be useful, future research must consider how to identify from all the frames available in a sentence, which frames better summarize the meaning of the text.

We leave the investigation of other ways to present task-relevant text to future research.

Chapter 8

Summary

The information that a developer seeks to aid in the completion of a task typically exists across different kinds of natural language software artifacts. In the artifacts that a developer consults, only some portions of the text will be useful to a developer's current task. Finding information that assists a developer in completing a task can be a time-consuming and cognitively frustrating process; to aid developers in this activity, researchers have proposed several artifact-centric techniques that automatically identify text likely relevant to a task.

However, existing techniques are constrained to working on one or only a few types of artifacts. As a result, these techniques might not follow the pace with which developers progress to using new kinds of technology. Furthermore, integrating them to allow developers to search for information seamlessly is challenging and costly.

To address these limitations, we propose a set of techniques that build upon approaches to interpret the meaning, or semantics, of the text for automating the identification of text relevant to a task. Our decision to use semantic-based approaches arises from the empirical analysis of text originating from different kinds of artifacts, that developers deem relevant to completing a set of software tasks where we have found consistency in the meaning of text considered relevant.

We evaluate the techniques proposed in this work by assessing whether they can identify text that developers deemed relevant in a series of artifacts associated with Android development tasks, embedding the most-promising technique identi-

fied in a tool, TARTI, which highlights the text identified as relevant in the artifacts that a developer inspects in their web browser. A second evaluation focuses on how this tool might assist developers while they work on a task, where we provide initial empirical evidence on how such a tool assists a software developer in completing a software task.

This thesis makes the following contributions to the field of software engineering and to research in mining unstructured text from natural language software artifacts:

- it demonstrates consistency in the semantic meaning of text relevant to six information-seeking tasks and their associated artifacts, which include API documents, bug reports, and question-and-answer web pages;
- it reports on the design of six semantic-based techniques that incorporate the semantics of words and sentences for identifying task-relevant text automatically;
- it identifies BERT and frame semantics as the most promising approaches for identifying text in a number of artifacts relevant to Android development tasks; where these techniques have accuracy comparable to a state-of-the-art approach tailored to one kind of artifact [190], i.e., Stack Overflow;
- it provides empirical data demonstrating the usefulness of a semantic-based tool in assisting a developer in completing a software development task by automatically providing them with text relevant to their task.

These contributions show that it is possible to design more generalizable techniques that can evolve to identify relevant text across the different kinds of natural language artifacts that developers use to record information, where we identify semantic-based techniques as a promising way for determining relevancy. Future research in this field could consider other means to semantically relate the text in a task and the text in the artifacts that developers might peruse. A second possible direction is to take into account the many information needs that individuals might have and further tailor the identification of task-relevant information based on who performs which task.

Bibliography

[1] Stanford CoreNLP. https://stanfordnlp.github.io/CoreNLP/. Verified: 2021-06-09. → page 54

[2] PyGithub. https://pypi.org/project/PyGithub/. Verified: 2021-06-09. → page 54

[3] StackAPI. https://stackapi.readthedocs.io/en/latest/. Verified: 2021-06-09. → page 54

[4] Alexa. https://alexa.com. Verified: 2021-04-08. → page 53

[5] Lock task mode. https://developer.android.com/work/dpc/dedicated-devices/lock-task-mode, . Verified: 2021-04-08. → page 50

[6] Managing WebView objects. https://developer.android.com/guide/webapps/managing-webview, . Verified: 2021-04-08. → page 51

[7] apriori-python. https://pypi.org/project/apriori-python/. Verified: 2021-10-11. → page 68

[8] beautifulsoup4. https://pypi.org/project/beautifulsoup4/. Verified: 2021-06-09. → page 54

[9] No lock screen controls ever #3578. https://github.com/AntennaPod/AntennaPod/issues/3578. Verified: 2021-04-08. → page 54

[10] Colaboratory. https://research.google.com/colaboratory/faq.html, . Verified: 2022-03-24. → page 84

[11] googlesearch. https://pypi.org/project/googlesearch-python/, . Verified: 2020-10-27. → page 53

[12] Create A React Native App Which works on Lock Screen (Android). https://bit.ly/3dNF8I5. Verified: 2021-04-08. → page 50

[13] *Evaluating the Use of Semantics for Identifying Task-relevant Textual Information — Supplementary Material*, Oct. 2021. Zenodo. doi:10.5281/zenodo.5591010. URL https://doi.org/10.5281/zenodo.5591010. → pages 49, 67

[14] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994. ISBN 1558601538. → page 68

[15] F. N. A. Al Omran and C. Treude. Choosing an NLP library for analyzing software documentation: A systematic literature review and a series of experiments. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 187–197, 2017. doi:10.1109/MSR.2017.42. → page 53

[16] M. Allamanis and C. Sutton. Why, when, and what: Analyzing stack overflow questions by topic, type, and code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 53–56. IEEE Press, 2013. ISBN 9781467329361. → pages 8, 20

[17] E. Alpaydin. *Introduction to machine learning*. MIT press, 2020. → page 19

[18] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 50–59, 2018. doi:10.1109/SEAA.2018.00018. → page 20

[19] D. Arya, W. Wang, J. L. Guo, and J. Cheng. Analysis and detection of information types of open source software issue discussions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 454–464, 2019. doi:10.1109/ICSE.2019.00058. → pages 8, 19, 20, 50, 54, 107

[20] D. M. Arya, J. L. Guo, and M. P. Robillard. Information correspondence between types of documentation for apis. *Empirical Software Engineering*, 25(5):4069–4096, 2020. → pages 3, 57, 58, 95, 108

[21] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. → page 21

[22] C. F. Baker, C. J. Fillmore, and J. B. Lowe. The berkeley framenet project. In *Proc. of the 17th Int'l Conf. on Computational Linguistics - Volume 1*, pages 86–90, Stroudsburg, PA, USA, 1998. → page 37

[23] S. Baltes and S. Diehl. Usage and attribution of stack overflow code snippets in github projects. *Empirical Software Engineering*, 24(3): 1259–1295, 2019. → pages 50, 78

[24] S. Baltes, C. Treude, and S. Diehl. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 191–194, 2019. → page 51

[25] S. Baltes, C. Treude, and M. P. Robillard. Contextual Documentation Referencing on Stack Overflow — Supplementary Material. Zenodo, Mar. 2019. doi:10.5281/zenodo.2585828. → page 51

[26] S. Baltes, C. Treude, and M. P. Robillard. Contextual documentation referencing on stack overflow. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi:10.1109/TSE.2020.2981898. → pages 51, 100

[27] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144. IEEE, 2015. → page 23

[28] C. L. Barry. User-defined relevance criteria: An exploratory study. *Journal of the American Society for Information Science*, 45(3):149–159, 1994. → page 60

[29] C. L. Barry. Document representations and clues to document relevance. *Journal of the American Society for Information Science*, 49(14): 1293–1303, 1998. ISSN 1097-4571. → page 60

[30] G. Bavota. Mining unstructured data in software repositories: Current and future trends. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 1–12, 2016. doi:10.1109/SANER.2016.47. → pages 1, 17, 19, 36, 64

[31] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015. doi:10.1109/TSE.2014.2367027. → page 49

[32] A. Begel and B. Simon. Novice software developers, all over again. In *Proc. of the Fourth Int'l Workshop on Computing Education Research*, pages 3–14, 2008. ISBN 978-1-60558-216-0. → page 1

[33] M. Bendersky, D. Metzler, and W. B. Croft. Effective query formulation with multiple information sources. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 443–452, 2012. → page 16

[34] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, 1994. → page 14

[35] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. → page 20

[36] H. Borges and M. T. Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018. → page 50

[37] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016. → page 50

[38] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming. *Proc. of the 27th Int'l Conf. on Human factors in computing systems - CHI 09*, page 1589, 2009. → pages 3, 16

[39] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proc. of the 2010 ACM Conf. on Computer Supported Cooperative Work*, pages 301–310, 2010. ISBN 978-1-60558-795-0. → pages 33, 52

[40] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the

linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265, 2015. → page 106

[41] K. Byström and K. Järvelin. Task complexity affects information seeking and use. *Information Processing and Management*, 31(2):191–213, 1995. ISSN 03064573. → page 8

[42] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 335–336, 1998. ISBN 1581130155. doi:10.1145/290941.291025. → page 16

[43] O. Chaparro, J. M. Florez, and A. Marcus. On the vocabulary agreement in software issue descriptions. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 448–452, 2016. doi:10.1109/ICSME.2016.44. → page 19

[44] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 396–407, 2017. ISBN 9781450351058. doi:10.1145/3106237.3106285. → pages 10, 19, 37, 107

[45] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 86–96, 2019. ISBN 9781450355728. doi:10.1145/3338906.3338947. → pages 10, 63

[46] P. Chatterjee, K. Damevski, N. A. Kraft, and L. Pollock. Software-related slack chats with disentangled conversations. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 588–592, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177. doi:10.1145/3379597.3387493. → page 8

[47] X. Chen, C. Zheng, and B. Chang. Joint multi-decoder framework with hierarchical pointer network for frame semantic parsing. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2570–2578, 2021. → page 107

[48] B. Clark. *Relevance theory*. Cambridge University Press, 2013. → pages 16, 28

[49] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 2126–2136, July 2018. doi:10.18653/v1/P18-1198. → page 65

[50] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):25–35, 1990. → page 106

[51] 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. doi:10.1109/TSE.2005.71. → pages 23, 24

[52] E. Cutrell and Z. Guan. What are you looking for? an eye-tracking study of information usage in web search. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'07, page 407–416, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935939. doi:10.1145/1240624.1240690. → pages 16, 106

[53] K. Cwalina and B. Abrams. *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Pearson Education, 2008. → page 80

[54] D. Das, D. Chen, A. F. Martins, N. Schneider, and N. A. Smith. Frame-semantic parsing. *Computational linguistics*, 40(1):9–56, 2014. → pages 25, 37, 107

[55] K. A. de Graaf, P. Liang, A. Tang, and H. van Vliet. The impact of prior knowledge on searching in software documentation. In *Proc. of the 2014 ACM Symposium on Document Engineering*, pages 189–198, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2949-1. → pages 16, 27

[56] L. Deng and Y. Liu. *Deep Learning in Natural Language Processing*. Springer Publishing Company, Incorporated, 1st edition, 2018. ISBN 9789811052088. → page 21

[57] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. → pages 9, 11, 22, 63, 65, 66, 77

[58] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall. Development emails content analyzer: Intention mining in developer discussions (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 12–23, 2015. doi:10.1109/ASE.2015.12. → page 107

[59] S. T. Dumais et al. Latent semantic indexing (LSI) and TREC-2. *Nist Special Publication Sp*, pages 105–105, 1994. → page 18

[60] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008. → page 99

[61] V. Efstathiou, C. Chatzilenas, and D. Spinellis. Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 38–41, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi:10.1145/3196398.3196448. → pages 65, 77

[62] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. How do API documentation and static typing affect API usability? In *Proc. of the 36th Int'l Conf. on SE*, pages 632–642, 2014. ISBN 978-1-4503-2756-5. → pages 1, 9

[63] D. Erhan, A. Courville, Y. Bengio, and P. Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010. → page 22

[64] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.findings-emnlp.139. → pages 77, 108

[65] F. Ferreira, L. L. Silva, and M. T. Valente. Software engineering meets deep learning: a mapping study. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1542–1549, 2021. → pages 20, 21

[66] I. Ferreira, E. Cirilo, V. Vieira, and F. Mourão. Bug report summarization: An evaluation of ranking techniques. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 101–110, Sep. 2016. doi:10.1109/SBCARS.2016.17. → page 50

[67] C. J. Fillmore. Frame semantics and the nature of language. *Annals of the New York Academy of Sciences*, 280(1):20–32, 1976. → pages 9, 11, 37, 63, 106

[68] J. L. Fleiss, B. Levin, M. C. Paik, et al. The measurement of interrater agreement. *Statistical methods for rates and proportions*, 2(212-236): 22–23, 1981. → page 29

[69] M. Fowler, J. Highsmith, et al. The agile manifesto. *Software development*, 9(8):28–35, 2001. → page 105

[70] L. Freund. A cross-domain analysis of task and genre effects on perceptions of usefulness. *Information Processing and Management*, 49(5): 1108–1121, 2013. ISSN 03064573. → page 46

[71] L. Freund. Contextualizing the information-seeking behavior of software engineers. *Journal of the Association for Information Science and Technology*, 66(8):1594–1605, aug 2015. ISSN 23301635. → pages 46, 73

[72] L. Fu, P. Liang, X. Li, and C. Yang. A machine learning based ensemble method for automatic multiclass classification of decisions. In *Evaluation and Assessment in Software Engineering*, pages 40–49. 2021. → page 20

[73] D. Fucci, A. Mollaalizadehbahnemiri, and W. Maalej. On using machine learning to identify knowledge in api reference documentation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 109–119, 2019. → pages 17, 22

[74] V. Garousi, D. Pfahl, J. M. Fernandes, M. Felderer, M. V. Mäntylä, D. Shepherd, A. Arcuri, A. Coşkunçay, and B. Tekinerdogan. Characterizing industry-academia collaborations in software engineering: evidence from 101 projects. *Empirical Software Engineering*, 24(4): 2540–2602, 2019. → page 8

[75] W. W. Gibbs. Software's chronic crisis. *Scientific american*, 271(3):86–95, 1994. → page 8

[76] J. Goldstein et.al. Summarizing Text Document: Sentence Selection and Evaluation Metrics. *In Proceeding of SIGIR'99*, pages 121–128, 1999. → pages 20, 22

[77] M. K. Gonçalves, C. R. de Souza, and V. M. González. Collaboration, information seeking and communication: An observational study of software developers' work practices. *J. Univers. Comput. Sci.*, 17(14): 1913–1930, 2011. → pages 8, 101

[78] T. Gross and A. G. Taylor. What have we got to lose? the effect of controlled vocabulary on keyword searching results. *College & research libraries*, 2005. → page 16

[79] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proc. of the 2013 Int'l Conf. on SE*, pages 842–851, 2013. ISBN 978-1-4673-3076-3. → pages 16, 53

[80] Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954. → pages 21, 64

[81] C. R. Hildreth. The use and understanding of keyword searching in a university online catalog. *Information technology and libraries*, 16(2):52, 1997. → page 16

[82] R. Holmes and A. Begel. Deep intellisense: A tool for rehydrating evaporated information. In *Proc. of the 2008 Int'l Working Conf. on Mining Software Repositories*, pages 23–26, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. → page 23

[83] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang. Api method recommendation without worrying about the task-api knowledge gap. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 293–304, 2018. doi:10.1145/3238147.3238191. → page 21

[84] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010. → page 105

[85] H. Jiang, J. Zhang, X. Li, Z. Ren, and D. Lo. A more accurate model for finding tutorial segments explaining apis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*,

volume 1, pages 157–167, 2016. doi:10.1109/SANER.2016.59. → pages 3, 36, 46, 57

[86] H. Jiang, J. Zhang, Z. Ren, and T. Zhang. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 38–48, 2017. doi:10.1109/ICSE.2017.12. → pages 3, 21, 34, 58

[87] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 2004. → page 18

[88] J. Josyula, S. Panamgipalli, M. Usman, R. Britto, and N. B. Ali. Software practitioners' information needs and sources: A survey study. In *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 1–6. IEEE, 2018. → page 53

[89] D. Jurafsky and J. H. Martin. *Speech and language processing*, volume 3. Pearson London, 2014. → pages 18, 25, 36, 37

[90] M. A. Just and P. A. Carpenter. A theory of reading: From eye fixations ot comprehension. *Psychological Review*, 87(4):329–354, 1980. doi:10.1037/0033-295X.87.4.329. → pages 5, 29

[91] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101, 2014. ISBN 9781450328630. doi:10.1145/2597073.2597074. → page 50

[92] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of SE*, pages 1–11, 2006. ISBN 1-59593-468-5. → page 23

[93] K. Kevic and T. Fritz. Automatic search term identification for change tasks. In *Companion Proc. of the 36th Int'l Conf. on SE*, pages 468–471, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. → pages 16, 53

[94] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz. Tracing software developers' eyes and interactions for change tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 202–213, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi:10.1145/2786805.2786864. → page 106

[95] W. Kintsch and T. A. van Dijk. Toward a model of text comprehension and production. *Psychological Review*, 85(5):363–394, 1978. ISSN 0033295X. → pages 10, 26

[96] K. Klaus. Content analysis: An introduction to its methodology, 1980. → page 98

[97] A. J. Ko and B. A. M. and. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 127–134, Sep. 2006. → pages 8, 10, 19, 36

[98] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12): 971–987, Dec. 2006. ISSN 0098-5589. doi:10.1109/TSE.2006.116. → pages 8, 42, 73

[99] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353. IEEE, 2007. → page 3

[100] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018. → page 58

[101] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019. → page 107

[102] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977. → page 29

[103] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006. → page 53

[104] J. Lazar, J. H. Feng, and H. Hochheiser. *Research Methods in Human Computer Interaction*. Morgan Kaufmann, Boston, second edition edition, 2017. ISBN 978-0-12-805390-4. → pages 92, 99, 100, 106

[105] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6): 1–35, 2019. → page 105

[106] H. Li, Z. Xing, X. Peng, and W. Zhao. What help do developers seek, when and how? In *2013 20th Working Conf. on Reverse Engineering (WCRE)*, pages 142–151, Oct 2013. → pages 3, 5, 8, 28, 45, 52, 53

[107] H. Li, S. Kim, and S. Chandra. Neural code search evaluation dataset. *arXiv preprint arXiv:1908.09804*, 2019. → page 108

[108] L. Li, J. Gao, T. Bissyandé, L. Ma, X. Xia, and J. Klein. Characterising deprecated android apis. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 254–264, 2018. → page 49

[109] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, pages 144–155, 2018. ISBN 978-1-4503-5714-2. doi:10.1145/3196321.3196326. → page 20

[110] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang. Deep learning in software engineering. *arXiv preprint arXiv:1805.04825*, 2018. → pages 9, 20, 21, 22

[111] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932. → page 89

[112] B. Lin, A. Zagalsky, M. Storey, and A. Serebrenik. Why developers are slacking off: Understanding how software teams use slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, CSCW '16 Companion, pages 333–336, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3950-6. doi:10.1145/2818052.2869117. → page 8

[113] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 83–94, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi:10.1145/2597008.2597155. → page 49

[114] C.-Y. Liou, W.-C. Cheng, J.-W. Liou, and D.-R. Liou. Autoencoder for words. *Neurocomputing*, 139:84–96, 2014. → page 22

[115] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu. Generating query-specific class api summaries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, pages 120–130, 2019. → pages 22, 23

[116] M. X. Liu, N. Hahn, A. Zhou, S. Burley, E. Deng, J. Hsieh, B. A. Myers, and A. Kittur. UNAKITE: Support developers for capturing and persisting design rationales when solving problems using web resources. 2018. → pages 23, 109

[117] M. X. Liu, A. Kittur, and B. A. Myers. To reuse or not to reuse? a framework and system for evaluating summarized knowledge. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW1):1–35, 2021. → pages 23, 24

[118] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. → page 107

[119] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 430–439, 2012. doi:10.1109/ICSM.2012.6405303. → pages 9, 18, 20, 27, 55, 70, 96

[120] H. P. Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4): 309–317, 1957. → page 18

[121] W. Maalej and M. P. Robillard. Patterns of knowledge in api reference documentation. In *IEEE Transactions on Software Engineering*, volume 39, pages 1264–1282, 2013. doi:10.1109/TSE.2013.12. → pages 8, 17, 19, 107

[122] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50 – 60, 1947. doi:10.1214/aoms/1177730491. → pages 60, 72, 91

[123] C. Manning, P. Raghavan, and H. Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010. → pages 17, 69, 96

[124] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2009. → pages 64, 67

[125] M.-C. Marcos, F. Gavin, and I. Arapakis. Effect of snippets on user experience in web search. In *Proceedings of the XVI International Conference on Human Computer Interaction*, pages 1–8, 2015. → page 16

[126] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 125–135, 2003. ISBN 076951877X. → page 18

[127] A. Marques, N. C. Bradley, and G. C. Murphy. Characterizing task-relevant information in natural language software artifacts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 476–487, 2020. doi:10.1109/ICSME46990.2020.00052. → pages 54, 55, 67, 73

[128] A. Marques, G. Viviani, and G. C. Murphy. Assessing semantic frames to support program comprehension activities. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 13–24, 2021. doi:10.1109/ICPC52881.2021.00011. → pages 11, 67, 107

[129] B. G. Mateus and M. Martinez. On the adoption, usage and evolution of kotlin features in android development. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020. ISBN 9781450375801. doi:10.1145/3382494.3410676. → page 49

[130] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79, 2013. doi:10.1109/ICSM.2013.18. → page 49

[131] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12): 1178–1193, 2017. doi:10.1109/TSE.2017.2656886. → pages 1, 23, 50, 101

[132] A. N. Meyer, C. Satterfield, M. Züger, K. Kevic, G. C. Murphy, T. Zimmermann, and T. Fritz. Detecting developers' task switches and types. *IEEE Transactions on Software Engineering*, 2020. → page 101

[133] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. 2013. → page 21

[134] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS)*, pages 3111–3119, 2013. → pages 9, 11, 21, 63, 64

[135] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012. → page 5

[136] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 33–48, 2005. ISBN 978-3-540-31725-8. → pages 1, 44, 69

[137] S. Nadi and C. Treude. Essential sentences for navigating stack overflow answers. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 229–239, 2020. doi:10.1109/SANER48275.2020.9054828. → pages 6, 8, 17, 50, 54, 55, 57, 108

[138] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. In *Proc. of Tenth International World Wide Web Conference, Hong Kong, China*, 2001. → page 16

[139] A. Nenkova and R. Passonneau. Evaluating content selection in summarization: The pyramid method. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 145–152, 2004. → pages 33, 46, 58, 60, 96

[140] E. Novotny. I don't think i click: A protocol analysis study of use of a library online catalog in the internet age. *College & research libraries*, 65 (6):525–537, 2004. → page 16

[141] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120. → pages 14, 20

[142] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *2012 20th*

IEEE International Conference on Program Comprehension (ICPC), pages 63–72. IEEE, 2012. → pages 8, 17

[143] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep*, 11, 2012. → pages 1, 8, 49

[144] R. Passonneau. Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy, May 2006. European Language Resources Association (ELRA). → page 98

[145] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi:10.18653/v1/N18-1202. → page 77

[146] G. Petrosyan, M. P. Robillard, and R. De Mori. Discovering information explaining API types using text classification. In *Proc. of the 37th Int'l Conf. on SE - Volume 1*, pages 869–879, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. → pages 19, 27, 34, 46

[147] R. Petrusel and J. Mendling. Eye-Tracking the Factors of Process Model Comprehension Tasks. In C. Salinesi, M. C. Norrie, and Ó. Pastor, editors, *Advanced Information Systems Engineering*, pages 224–239, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38709-8. → page 106

[148] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett. Foraging and navigations, fundamentally: Developers' predictions of value and cost. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 97–108, 2016. ISBN 9781450342186. doi:10.1145/2950290.2950302. → page 53

[149] P. Pirolli and S. Card. Information foraging. *Psychological review*, 106(4): 643, 1999. → pages 14, 15, 44, 46

[150] L. Ponzanelli, A. Mocci, and M. Lanza. Summarizing complex development artifacts by mining heterogeneous data. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 401–405, 2015. doi:10.1109/MSR.2015.49. → page 20

[151] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza. Supporting software developers with a holistic recommender system. In *Proc. of the 39th Int'l Conf. on SE*, pages 94–105, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. → pages 23, 28, 110

[152] D. Rafiei, K. Bharat, and A. Shukla. Diversifying web search results. In *Proceedings of the 19th international conference on World wide web*, pages 781–790, 2010. → page 16

[153] N. Rao, C. Bansal, T. Zimmermann, A. H. Awadallah, and N. Nagappan. Analyzing web search behavior for software engineering tasks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 768–777, 2020. doi:10.1109/BigData50022.2020.9378083. → pages 3, 51

[154] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: A case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 505–514, 2010. ISBN 9781605587196. doi:10.1145/1806799.1806872. → pages 10, 33, 55, 57

[155] M. P. Robillard and Y. B. Chhetri. Recommending reference api documentation. *Empirical Software Engineering*, 20(6):1558–1586, Dec. 2015. ISSN 1382-3256. doi:10.1007/s10664-014-9323-y. → pages 10, 19, 36, 37, 54, 55, 80, 108

[156] M. P. Robillard and R. Deline. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011. → pages 1, 5, 9, 27, 33, 44, 52, 57, 95

[157] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. → page 21

[158] D. Russo, P. H. Hanel, S. Altnickel, and N. Van Berkel. The daily life of software engineers during the covid-19 pandemic. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 364–373. IEEE, 2021. → page 82

[159] D. Russo, P. H. Hanel, S. Altnickel, and N. van Berkel. Predictors of well-being and productivity among software professionals during the covid-19 pandemic–a longitudinal study. *Empirical Software Engineering*, 26(4):1–63, 2021. → page 82

[160] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975. → page 18

[161] T. Saracevic. Relevance: A review of and a framework for the thinking on the notion in information science. *Journal of the American Society for information science*, 26(6):321–343, 1975. → pages 14, 15, 46

[162] T. Saracevic. Relevance: A review of the literature and a framework for thinking on the notion in information science. part iii: Behavior and effects of relevance. In *Journal of the American Society for Information Science and Technology*, volume 58, pages 2126–2144, 2007. doi:https://doi.org/10.1002/asi.20681. → page 77

[163] T. Saracevic. Relevance: A review of the literature and a framework for thinking on the notion in information science. part ii: nature and manifestations of relevance. In *Journal of the American Society for Information Science and Technology*, volume 58, pages 1915–1933, 2007. doi:https://doi.org/10.1002/asi.20682. → page 77

[164] C. Satterfield, T. Fritz, and G. C. Murphy. Identifying and describing information seeking tasks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 797–808, 2020. → pages 23, 101

[165] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999. ISSN 0098-5589. doi:10.1109/32.799955. → page 99

[166] M. Senapathi, J. Buchan, and H. Osman. Devops capabilities, practices, and challenges: Insights from a case study. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 57–67, 2018. → page 105

[167] Z. Sharafi, B. Sharif, Y.-G. Guéhéneuc, A. Begel, R. Bednarik, and M. Crosby. A practical guide on conducting eye tracking studies in software engineering. *Empirical Software Engineering*, 25(5):3128–3174, 2020. → page 106

[168] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, 2006. → pages 3, 23

[169] R. F. Silva, C. K. Roy, M. M. Rahman, K. A. Schneider, K. Paixao, and M. de Almeida Maia. Recommending comprehensive solutions for programming tasks by mining crowd knowledge. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 358–368, 2019. doi:10.1109/ICPC.2019.00054. → pages 1, 9, 10, 22, 53, 63

[170] G. Singer, U. Norbisrath, E. Vainikko, H. Kikkas, and D. Lewandowski. Search-logger analyzing exploratory search tasks. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 751–756, 2011. → page 3

[171] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. In *Pro. 6th Int'l Workshop on Program Comprehension. IWPC'98*, pages 173–179, June 1998. → page 27

[172] R. Spence. *Information visualization*, volume 1. Springer, 2001. → page 94

[173] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009. → page 41

[174] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 157–166, 2009. doi:10.1109/ICSM.2009.5306335. → pages 1, 8, 16, 42, 53, 104

[175] J. Sun, Z. Xing, X. Peng, X. Xu, and L. Zhu. Task-oriented api usage examples prompting powered by programming task knowledge graph. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 448–459. IEEE, 2021. → page 23

[176] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014. → page 21

[177] S. Swayamdipta, S. Thomson, C. Dyer, and N. A. Smith. Frame-Semantic Parsing with Softmax-Margin Segmental RNNs and a Syntactic Scaffold. *arXiv preprint arXiv:1706.09528*, 2017. → page 107

[178] A. Taylor, M. Marcus, and B. Santorini. The penn treebank: an overview. *Treebanks*, pages 5–22, 2003. → page 18

[179] E. Thiselton and C. Treude. Enhancing python compiler error messages via stack overflow. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. doi:10.1109/ESEM.2019.8870155. → page 82

[180] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? *Proc. of the 33rd Int'l Conf. on SE*, page 804, 2011. ISSN 0270-5257. → page 6

[181] M. Umarji, S. E. Sim, and C. Lopes. Archetypal internet-scale source code searching. In B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, pages 257–263, 2008. ISBN 978-0-387-09684-1. → pages 1, 14, 27, 45

[182] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 6000–6010, 2017. ISBN 9781510860964. → pages 21, 66

[183] M. R. Vieira, H. L. Razente, M. C. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras. On query result diversification. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1163–1174. IEEE, 2011. → page 16

[184] G. Viviani, M. Famelis, X. Xia, C. Janik-Jones, and G. C. Murphy. Locating latent design information in developer discussions: A study on pull requests. 2019. doi:10.1109/TSE.2019.2924006. → page 58

[185] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022. → page 21

[186] B. M. Wildemuth and L. Freund. Assigning search tasks designed to elicit exploratory search behaviors. In *Proc. of the Symposium on Human-Computer Interaction and Information Retrieval*, pages 4:1–4:10, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1796-2. → page 27

[187] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. → pages 30, 34, 41, 91

[188] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 137–146, 2020. ISBN 9781450375177. doi:10.1145/3379597.3387459. → page 50

[189] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing. What do developers search for on the web? *Empirical Softw. Engg.*, 22(6): 3149–3185, Dec. 2017. ISSN 1382-3256. → pages 3, 45, 51

[190] B. Xu, Z. Xing, X. Xia, and D. Lo. AnswerBot: Automated generation of answer summary to developers' technical questions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 706–716, 2017. doi:10.1109/ASE.2017.8115681. → pages 9, 11, 12, 22, 23, 50, 53, 70, 101, 107, 112

[191] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 127–137, 2016. doi:10.1109/ISSRE.2016.33. → pages 10, 63

[192] M. Yazdaninia, D. Lo, and A. Sami. Characterization and prediction of questions without accepted answers on stack overflow. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 59–70, 2021. doi:10.1109/ICPC52881.2021.00015. → page 50

[193] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 404–415, 2016. ISBN 9781450339001. doi:10.1145/2884781.2884862. → pages 9, 21, 64

[194] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021. → page 21

[195] H. Zhang, S. Wang, T.-H. Chen, and A. E. Hassan. Reading answers on stack overflow: Not enough! *IEEE Transactions on Software Engineering*, 2019. → page 57

[196] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010. → page 57