Supporting a Developer's Discovery of Task-Relevant Information

by

Arthur de Sousa Marques

B. Computer Science, Federal University of Campina Grande, 2013

M. Computer Science, Federal University of 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, artifacts structured as tutorials might be easier to locate information given their structured headings 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 such 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 the portion of the 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 perform 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 effectively complete a software development task.

Table of Contents

List of Tables

List of Figures

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

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

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 [76, 106]. 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 [96, 111] or question-and-answer developer forums [87, 102].

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 [9]. However, the sheer amount of information in these natural language artifacts may prevent a developer from comprehensively assessing what is useful to their task [80]. 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 [30, 76]. 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 [10, 96]. 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.

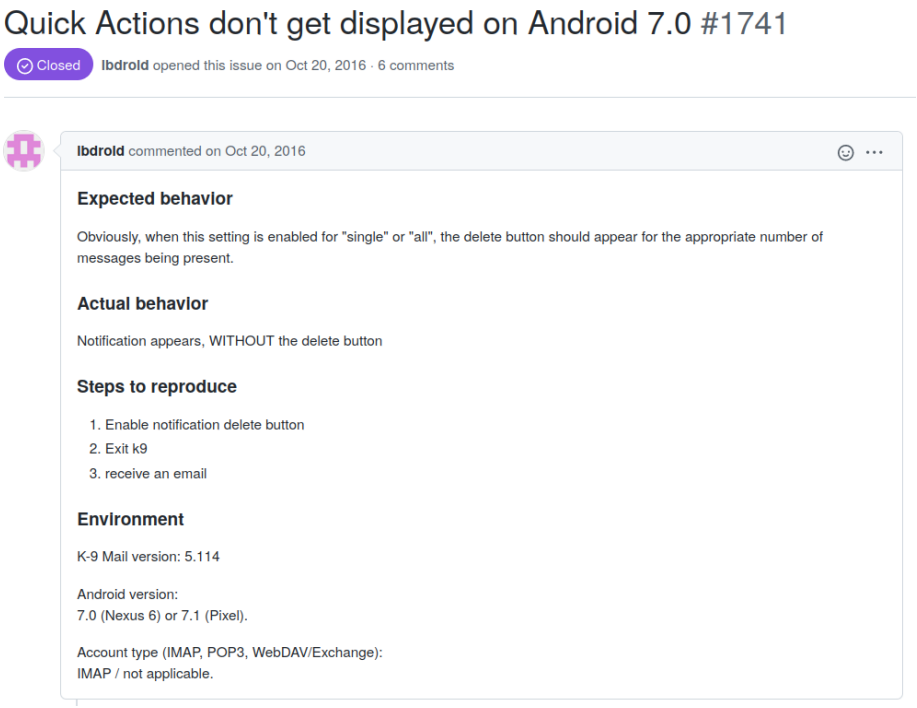

Figure 1.1: k-9 mail GitHub issue #1741 indicating that quick actions don't get displayed on Android 7.0

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

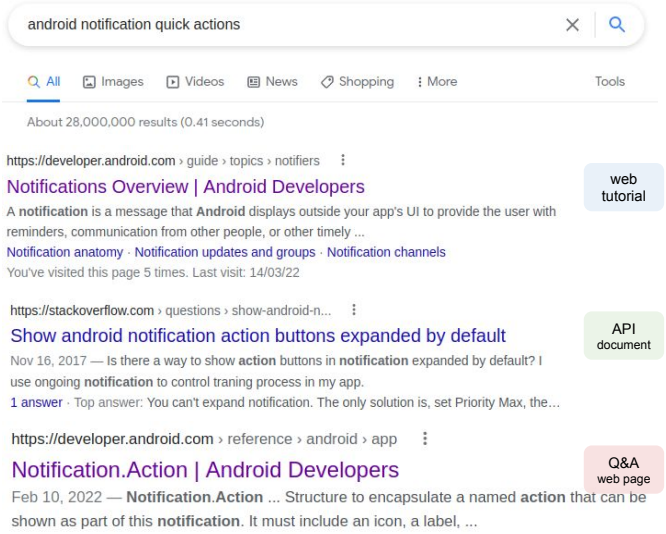

Figure 1.2: Search results showing artifacts of potential interest to the Android quick actions issue

stand and resolve this task [59, 61, 101]. Often, this knowledge can be acquired from a developer's peers [103]. However, the fragmented and distributed nature of software development may prevent the developer from accessing their peers [59], instead they seek online web resources for information that may assist them in completing the task-at-hand [93, 117].

A common way to find software artifacts pertinent to the developer's task is through a web search engine [14, 61]. 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 [5] through a series of structured topics that progressively explain concepts about the technology [49, 50]. 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.

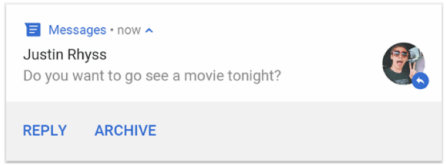

Figure 9. A notification with action buttons

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 [61]. 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 [79]. 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 [96] 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 [53].

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

[5] https://stackoverflow.com/

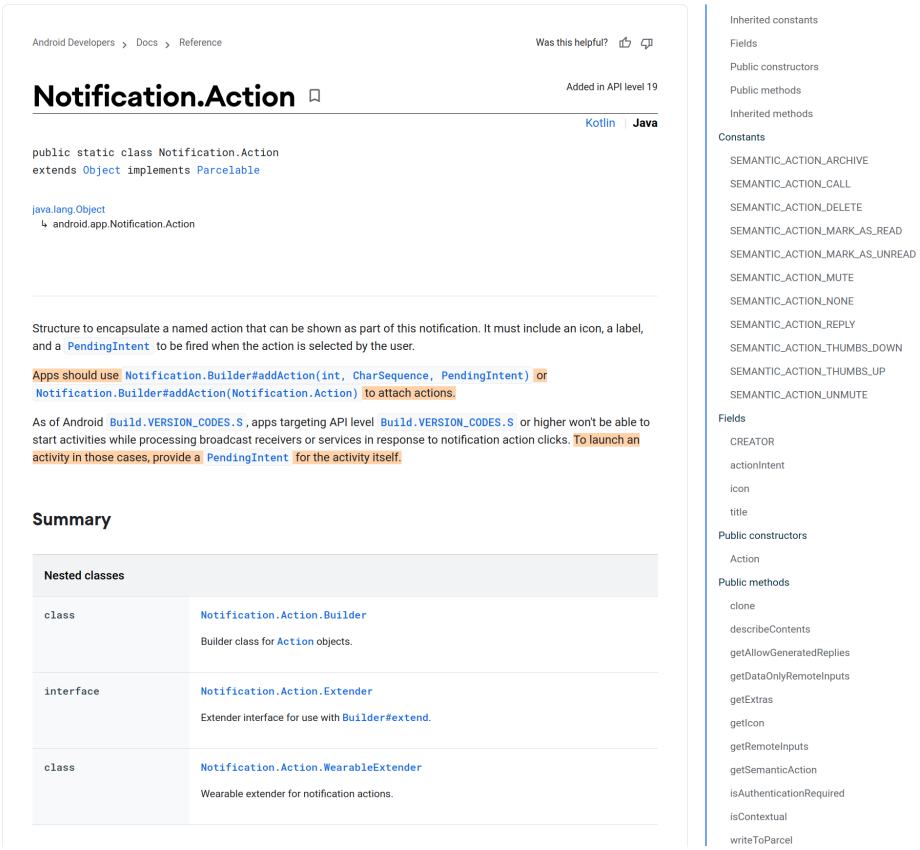

Figure 1.4: Snapshot of the Android Notification.Action API reference documentation with highlights relevant to the GitHub issue #1741

question usually contains both text and code snippets and it includes a set of tags about the problem's programming language or technology [110]. 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 [81], 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

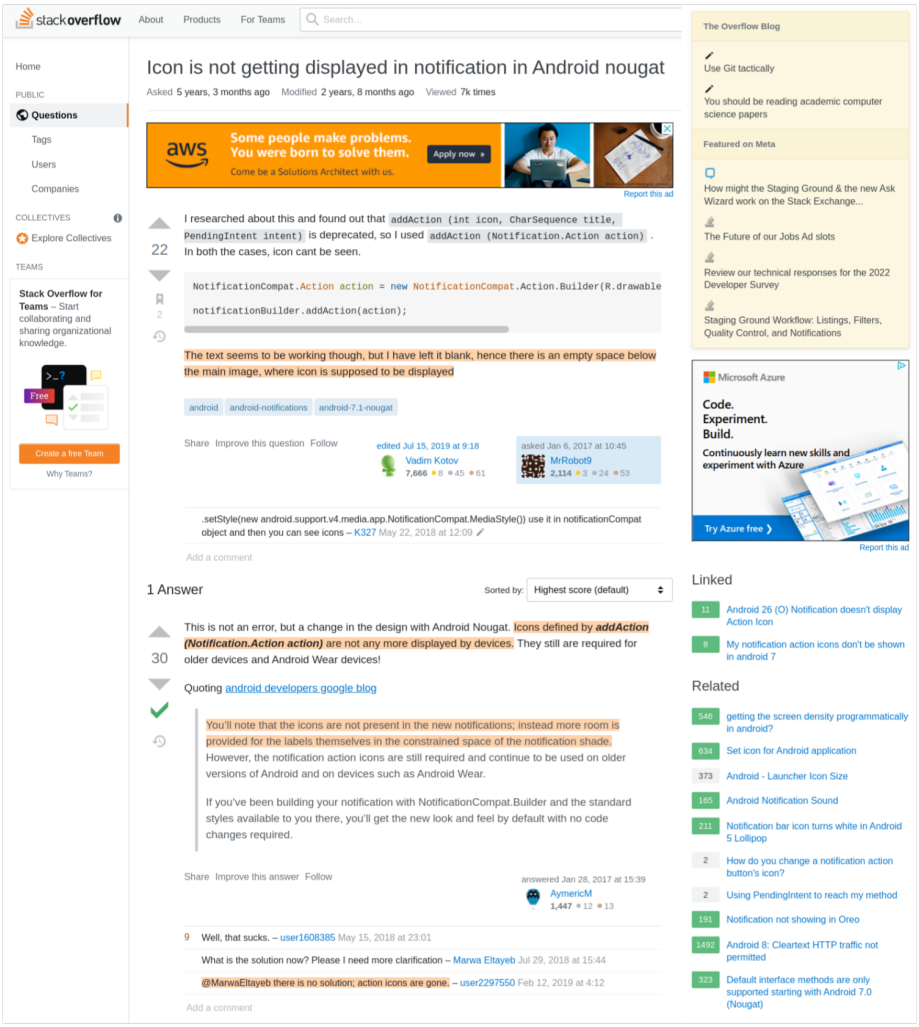

Figure 1.5: Snapshot of a Stack Overflow question about Android notifications with highlights relevant to the GitHub issue #1741

make finding useful information challenging. For instance, only half of the Android questions on Stack Overflow have an accepted answer [87] and millions of questions have more than one answer [81]. Technologies also evolve and answers become obsolete [1]. 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 [16, 42, 58]. 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 [61, 106].

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 [4, 57, 71, 86].

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 [39]. By the time an artifact-specific tool might be available [40], 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 [21, 64].

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 [63, 69], 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 [30] 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 [96].

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., [102, 118]) 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 [120] and do not investigate the range of semantic techniques [28, 33, 78] 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 [56]. 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 [19, 57, 94, 95]. 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 [119], searching for comprehensive code examples [102], or assessing the quality of information available in bug reports [20]. 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 [28, 78] and sentences [33, 75] 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 [118] 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 [118], 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 **??** describes the groundwork for producing the corpus ($\text{DS}_{android}$) 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 **??** 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 $\text{DS}_{android}$ 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 **??** 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 **??**, we discuss challenges and decisions made throughout our work, implications from our findings, and potential future work.

Chapter **??** 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 [89]. At times, an information need would be directed to a library clerk who would suggest books for the person's search [99]. Then, when the World Wide Web (WWW) [12] became popular, individuals started to rely on search engines to seek information now digitally stored in web pages [85]. 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 [111], 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 [89] 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 [89].

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 [99]. 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) [46] while other studies observed impatience and near-random search habits in novice librarians [84]. In contemporary web search engines, researchers have also investigated how individuals formulate queries [11, 43], how prior knowledge helps them to more efficiently perform searches [26], and what search results they inspect, going to the lengths of using eye-tracking technology for this purpose [24, 73].

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 [17]. 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 [22]. This has led researchers to investigated algorithms that balance the relevance and the novelty of the results retrieved by a web search [82, 92, 113] 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 [14, 26, 106], a common trend is that the identification of good search terms is as, or even more important than the search algorithm itself [55]. 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 [44, 84]. 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 [9]. 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 [9]. 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.

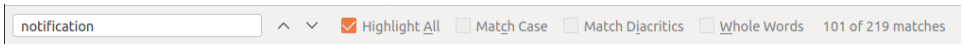

Figure 2.2: Find in page feature with results for the '*notification*' term

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 [86]. Although the heuristics and regular expressions used in this and other studies [71, 81] are lightweight and effective [9], pattern-matching approaches are often specific to certain kinds of domains and types of artifact [38], 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 [72]. 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*) [51, 70], how to represent sentences (*VSM*) [98], or how to account for different words that appear in a similar context (*LSI*) [29].

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 [69], or in approaches that help cluster software components to aid program comprehension of a software system [74]. In this dissertation, we combine information retrieval and word embeddings to identify relevant text across different kinds of artifacts, as Chapter **??** further details.

2.2.3 Natural Language Processing

Natural Language Processing (NLP) relies on the lexical or syntactical analysis of the text [52] 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 [109] while the latter identifies relationships between them, such as how '*io.StringIO*' is the direct object (*obj*) associated with the verb '*use*'.

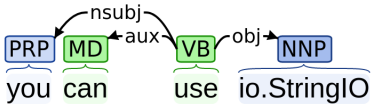

Figure 2.3: Syntactic elements for a sentence giving instructions about how to perform some file system operation

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 [95]. As another example, DeMIBuD is an automatic approach that uses dependency parsing to automatically detect sentences discussing a bug's expected behaviour [19]. 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 [4, 57, 71]. 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 [9].

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 [2], 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 [18] or to classify paragraphs which contain explanations about an API element in web tutorials [88]. 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 [4]. 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 [37].

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 [3, 32].

Unsupervised Learning

Common applications of unsupervised learning to natural language software artifacts comprise text summarization [41] 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 [69].

Unsupervised summarization approaches are often based on variations of the PageRank algorithm [85], 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 [90], coding tutorials [62], or bug reports, as when Li et al. summarized key elements required to understand a bug and its resolution [63].

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) [13], 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 [1] 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 [50].

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 [27, 121]. 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 [78] 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 [45, 77]. 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 [120] or Huang and colleagues' study on the usage of word embeddings for API recommendation [48]. Guided by these findings, Chapter **??** describes how we use word embeddings in the identification of task-relevant text.

Many other DL studies in software engineering [32, 63] use neural network architectures for the same range of problems discussed in Section 2.2.4. As explained by Watson et al. [114], 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 [97, 108], encoder-decoders [6], Transformers [112], etc.) suitable for different problems.

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

used an auto-encoder [65] to produce more accurate and diverse summaries for bug reports [63] while Fucci et al. used a recurrent neural network (RNN) to identify the types of information available in API documentation [38].

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 [31]. In Chapter **??**, we investigate how we can use a state-of-the-art architecture, namely BERT [28], 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 [66, 102, 118] 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 [41]. 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 [118] 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 [66], 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 [118] 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 [66]. 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 [54, 76, 100].

Software developers engage in many sensemaking and decision-making activities [101] and several studies have investigated how to provide means to better assist developers in performing such activities [8, 67, 68]. 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 [91].

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 [47] 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 [107] 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 [23] 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 [23]. 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 [68].

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 [68], which assisted in the design of our experiment for evaluating an automated approach to task-relevant text identification (Chapter **??**).

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 [25, 52] 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 [56]. 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 [96, 104, 111]; evaluates a possible reusable library to be incorporated into their implementation [111]; or confirms a system's behaviour referring to past discussions in community forums or in the system's documentation [69, 104, 111]. To design tasks for these activities, we applied criteria described by Petrosyan and colleagues [88], 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 [26, 115].

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

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

common information sources sought in development tasks [61, 91], 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 [22]. As examples of these criteria in action, the `Lucene` artifacts chosen contain complementary sources where sen-

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

tences in different artifacts pertain to the same topic while the artifacts chosen for 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 ($\kappa = 0.45$) [34, 60] 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 [53] 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 (± 4, ranging from 3 to 20 years) and 16 out of the 20 participants reported professional experience with an average of 4.7 years (± 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 [116], 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 correct-

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

ness of every task, and also conducted a semi-structured interview that discussed 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

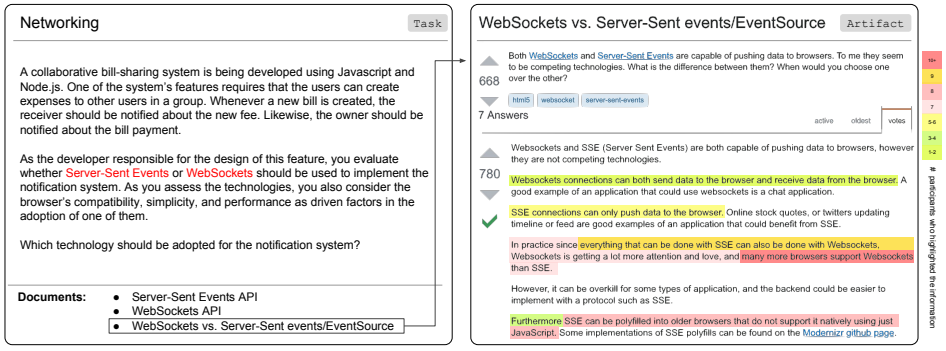

Figure 3.2: An example of a task description and a depiction of collected data, shown as a heatmap of text that participants highlighted as relevant to the task

it provides an overview of highlights (right-side) selected by participants using our 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 [96]. 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 [15, 94]. 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 [83] 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

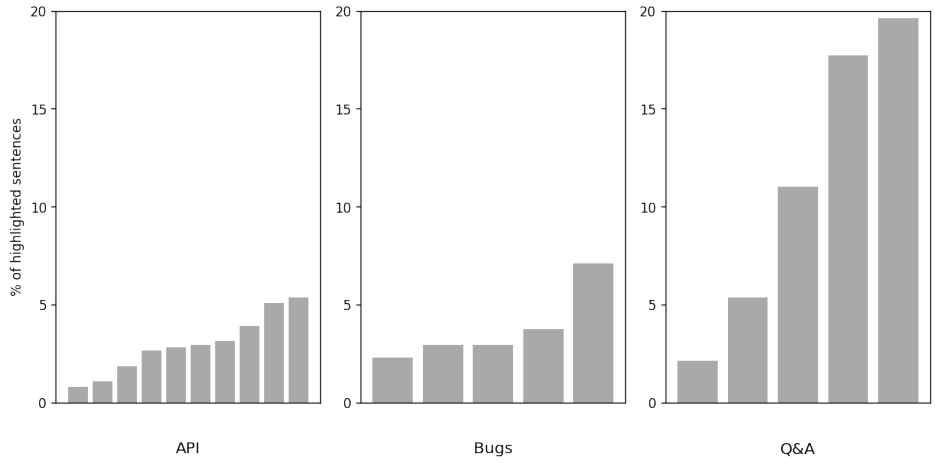

Figure 3.3: Percentage of highlighted sentences per artifact grouped by artifact's type

the same content in their summaries, the more consistent the view of information 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., [88] and [50]) 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 [116] 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

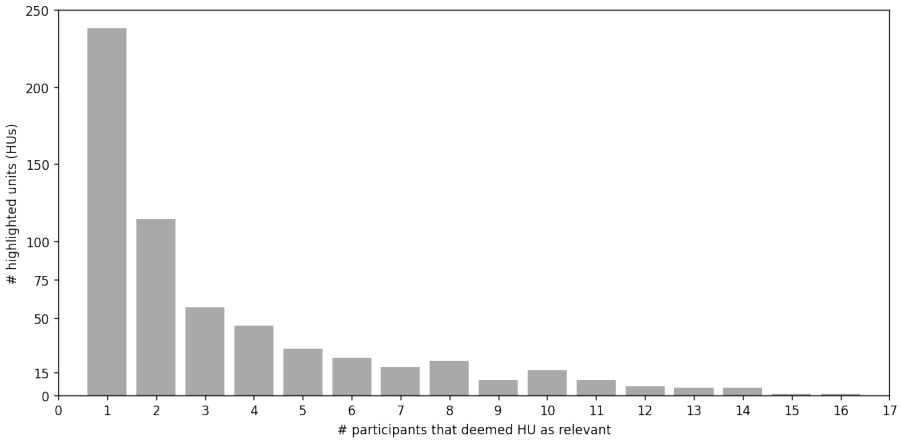

Figure 3.4: Distribution of the number of participants who deem an HU as relevant

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

completing a task. As expected, HUs at the bottom tier vary more per participant. 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 [95], 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., [95] or [49]) 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 [57]. 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 [52]. As a result, many NLP techniques would discard them as part of their pre-processing steps [9].

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., [19, 95]) 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 [33, 52]. Semantic frames are centered around events, labelled frames, which abstract both the event as well as relationships, entities or participants related to that event [7, 33].

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 [25], where we extract the frames of every HU, resulting in 3,719 frames

Databases: *I have yet to see a situation where hibernate was the reason for poor performance in production.*

[I]$_{\texttt{fe:perceiver}}$ have yet to [see]$_{\texttt{perception of experience}}$

[a situation where hibernate was the reason for ...]$_{\texttt{fe:phenomenon}}$

Lucence: *As far as I understand that sums up totalTermFreqs for all terms of a field*

As far as [I]$_{\texttt{fe:cognizer}}$ [understand]$_{\texttt{grasp}}$

[that sums up totalTermFreqs for all terms of a field]$_{\texttt{fe:phenomenon}}$

Figure 3.5: Example of frames and frame elements

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 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

	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

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 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, 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 [116] 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.

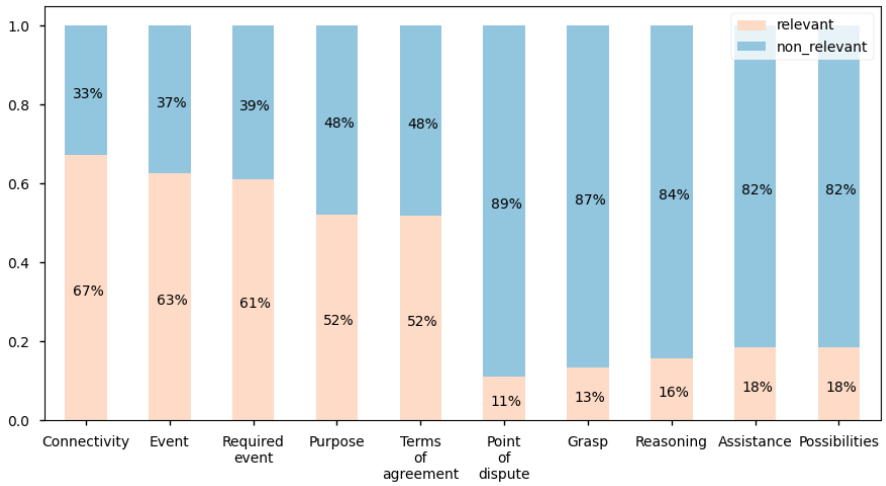

Figure 3.6: Distribuion of semantic frames over the text; the figure depicts the top five frames most commonly observed in relevant and non-relevant sentences, respectively

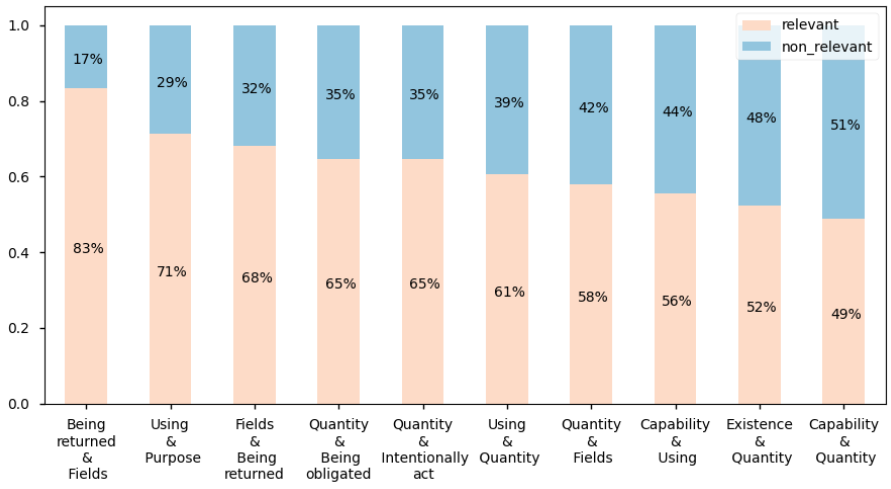

Figure 3.7: Distribution of co-occurring frames over the text

> *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 [105]. 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* [58, 106] 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

	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

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 foraging theory and how an individual follows some *information scent* for judging the value, or cost of investigating some text according to available cues [89], 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 [80]; 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 [96] 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 infor-

mation 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 [61, 111, 117]. 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 [83, 89, 99] and usefulness [35, 36]. 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 [88] and Android [49] 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.

Bibliography

[1] 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

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

[3] 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

[4] 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

[5] 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. → page 3

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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. → page 33

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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. → page 10

[21] 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

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

[23] 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

[24] 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 407416, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935939. doi:10.1145/1240624.1240690. → page 16

[25] 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

[26] 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

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

[28] 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

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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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. → page 46

[37] 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

[38] 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

[39] 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

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

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

[42] 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. → page 8

[43] 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

[44] 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. → page 16

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

[46] 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

[47] 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

[48] 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

[49] 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, 47

[50] 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

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

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

[53] 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

[54] 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

[55] 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. → page 16

[56] 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

[57] 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

[58] 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

[59] 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

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

[61] 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, 46

[62] 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

[63] 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

[64] 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

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

[66] 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

[67] 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. → page 23

[68] 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

[69] 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

[70] 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

[71] 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

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

[73] 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

[74] 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

[75] 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. → page 11

[76] 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

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

[78] 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

[79] 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

[80] 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

[81] 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

[82] 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

[83] 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

[84] 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

[85] 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

[86] 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

[87] 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

[88] 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, 47

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

[90] 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

[91] 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

[92] 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

[93] 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. → page 3

[94] 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

[95] 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

[96] 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

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

[98] 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

[99] 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

[100] 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. → page 23

[101] 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

[102] 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

[103] 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

[104] 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

[105] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009. → page 42

[106] 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

[107] 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

[108] 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

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

[110] 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

[111] 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, 46

[112] 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. → page 21

[113] 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

[114] 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

[115] 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

[116] 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, 40

[117] 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, 46

[118] 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

[119] 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. → page 10

[120] 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

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