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 natural language associated artifacts

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$_{Python}$ TODO

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

STDV Standard deviation

SDK software development kit

NLP Natural Language Processing

IR Information Retrieval

ML Machine Learning

DL Deep Learning

CNN Convolutional Neural Network

VSM Vector Space Model

BERT Bidirectional Encoder Representations from Transformers

Chapter 1

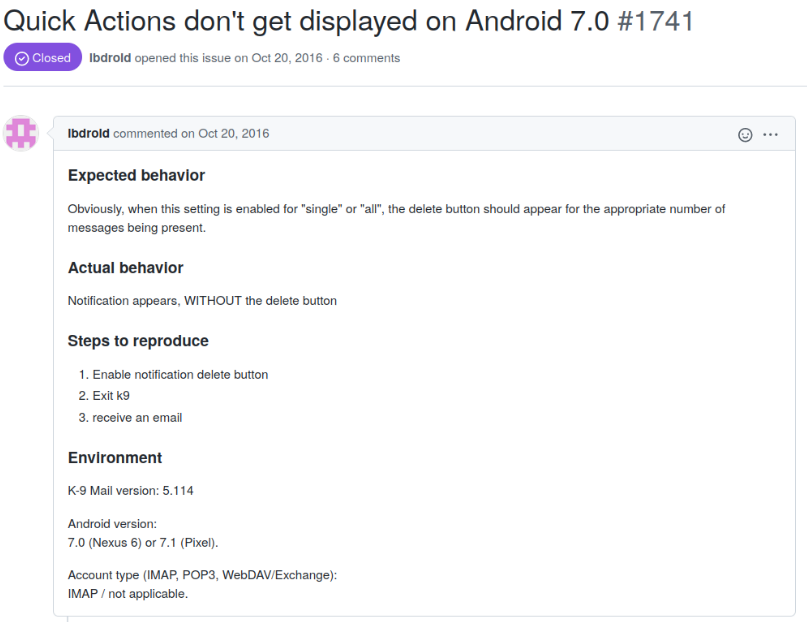
Introduction

When performing software tasks in large and complex software systems, software developers typically consult several different kinds of artifacts that assist them in their work [110, 150]. For example, when incorporating a new API library needed for a new feature, a developer might consult official API documents and guidelines [136, 154] or question-and-answer forums for functionality, security and performance-related topics [123, 146].

Much critical information in this and other non-source code types of artifacts contain data in the form of unstructured text [27] and a developer must read the text to find the information that is *relevant* to the task being performed. However, the sheer amount of information *within* these natural language artifacts may prevent a developer from comprehensively assessing what is useful to their task [117]. Just within one kind of document, 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 a particular software task [55, 110] and 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 [117].

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 the app notifications are not working as expected in the Android 7.0 version.



Quick Actions don't get displayed on Android 7.0 #1741

lbdroid opened this issue on Oct 20, 2016 · 6 comments

lbdroid commented on Oct 20, 2016

Expected behavior

Obviously, when this setting is enabled for "single" or "all", the delete button should appear for the appropriate number of messages being present.

Actual behavior

Notification appears, WITHOUT the delete button

Steps to reproduce

1. Enable notification delete button
2. Exit k9
3. receive an email

Environment

K-9 Mail version: 5.114

Android version:
7.0 (Nexus 6) or 7.1 (Pixel).

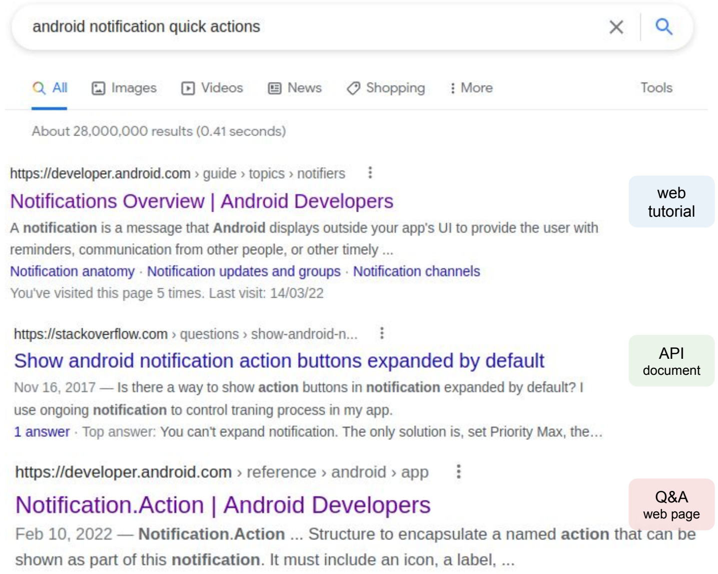
Account type (IMAP, POP3, WebDAV/Exchange):
IMAP / not applicable.

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 need additional knowledge to understand and resolve the bug [84, 89, 145]. More than often, this knowledge can be acquired from a developer's peers [147]. However, the fragmented and distributed nature of software development may prevent the developer from accessing their peers [84], what often makes them seek online web resources for information that may assist them in completing the task-at-hand [131, 161].

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

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



About 28,000,000 results (0.41 seconds)

https://developer.android.com › guide › topics › notifiers

Notifications Overview | Android Developers

A **notification** is a message that **Android** displays outside your app's UI to provide the user with reminders, communication from other people, or other timely ...

Notification anatomy · Notification updates and groups · Notification channels

You've visited this page 5 times. Last visit: 14/03/22

https://stackoverflow.com › questions › show-android-n...

Show android notification action buttons expanded by default

Nov 16, 2017 — Is there a way to show **action** buttons in **notification** expanded by default? I use ongoing **notification** to control traning process in my app.

1 answer · Top answer: You can't expand notification. The only solution is, set Priority Max, the...

https://developer.android.com › reference › android › app

Notification.Action | Android Developers

Feb 10, 2022 — **Notification.Action** ... Structure to encapsulate a named **action** that can be shown as part of this **notification**. It must include an icon, a label, ...

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

A common way to find software artifacts pertinent to the developer's task is through the usage of a web search engine [35, 89], what will eventually provide to the developer a set of artifacts, such as the ones shown in Figure 1.2, which the developer will inspect in search for information that will eventually explain how the Android notifications work and why they are not being displayed properly.

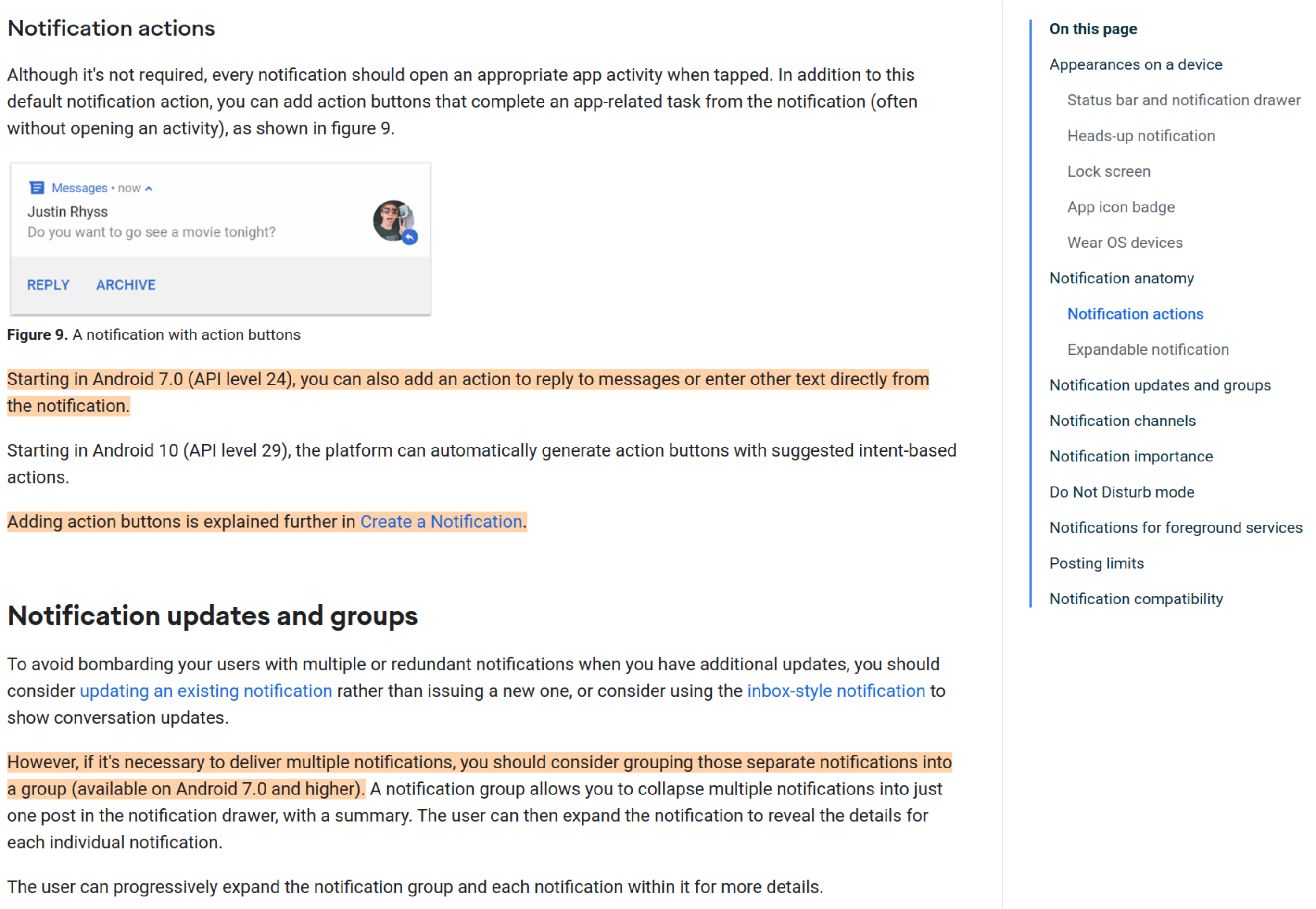
However, finding information useful to the developer's task in these and other pertinent artifacts can be a time-consuming and cognitively frustrating process [29, 136]. First, an artifact such as the web tutorial retrieved in the developer's search and detailed in Figure 1.3 might contain legacy information, boilerplate text, or sections intended for a target audience other than the developer reading it (e.g., novice programmers) [135]. This artifact contains nine distinct sections and a total of approximately 200 sentences. Reading all of its content would take approximately 10 minutes or more[3] of a developer's time [55, 110] when only a portion of the text might be relevant to the quick actions issue. For example, Figure 1.3

[3] Using a standard reading metric of 200 words per minute [77]

shows that the Android notifications tutorial has information about both the Android versions 7.0 and 12.0 and, given that the developer's task is related to the former version, only the text highlighted (in orange) might be of relevance.

As tasks become more complex [37, 128], a developer also has to combine multiple textual fragments—from the same artifact or from different ones—to understand what is needed for the task-at-hand [127]. For example, the information within the web tutorial might have only partially assisted the developer in understanding how Android notifications work and thus, they would consult other artifacts from their search for more information, namely the API document and Q&A web page. Figure 1.4 gives further insight into how task-relevant text is scattered across these other artifacts. From the figure, we find that some of the text potentially relevant to this task (in orange) is attached to elements that a reader would not intuitively access [135], i.e., a sentence at the end of a paragraph or a comment under the original question, and if no tool support is provided, much of the process of finding this relevant text falls on the developer's shoulders [37, 65, 83].

Researchers have long recognized the value of assisting developers in locating information in the natural language artifacts sought as part of a software task, proposing many tools and approaches that combine Information Retrieval (IR), Natural Language Processing (NLP) and Machine Learning (ML) techniques to identify potentially useful text in certain kinds of artifacts. For example, Nadi and Treude consider that relevant information in Q&A web pages are often found in text with conditional clauses (i.e., sentences with the word '*if*') [120] while Robillard and Chhetri assume that relevant text in API documents mention a code element such as a class name or method signature [135]; they use such assumptions in techniques that identify this text automatically. As other examples, both Xu et al. [163] and Silva et al. [146] use meta-data available on each of the answers in a Stack Overflow post as a proxy for relevant information. That is, they use the number of votes an answer has or the checkmark indicating if an answer is the correct one, both shown in Figure 1.4, to automatically identify text that might be relevant to a given task in these answers.



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.

Messages • now

Justin Rhyss

Do you want to go see a movie tonight?

REPLY ARCHIVE

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

Appearances on a device

Status bar and notification drawer

Heads-up notification

Lock screen

App icon badge

Wear OS devices

Notification anatomy

Notification actions

Expandable notification

Notification updates and groups

Notification channels

Notification importance

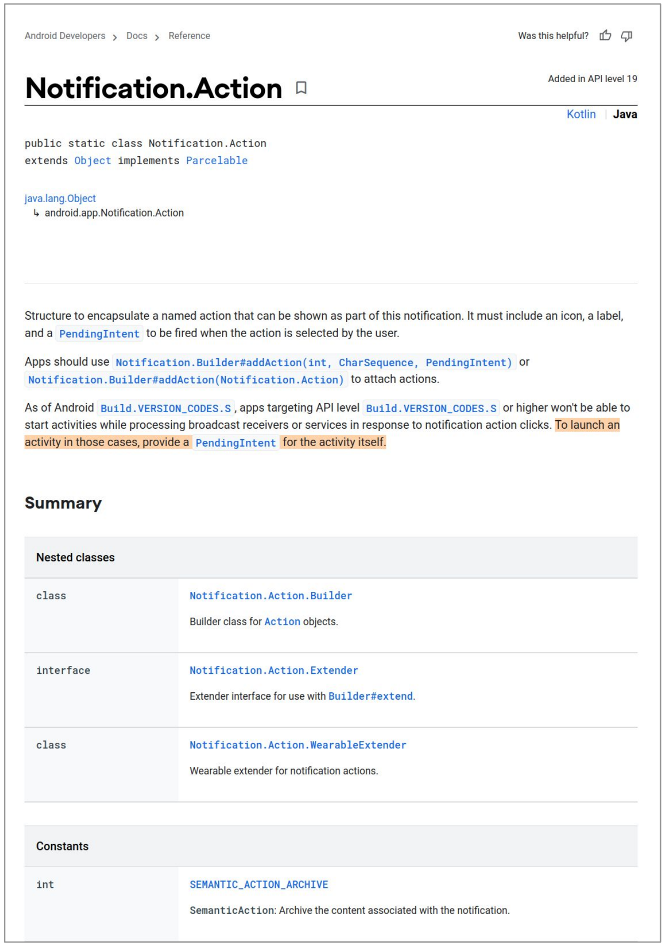
Do Not Disturb mode

Notifications for foreground services

Posting limits

Notification compatibility

Figure 1.3: Snapshot of the official Android notifications API overview



Android Developers › Docs › Reference

Notification.Action

Added in API level 19

```
public static class Notification.Action
extends Object implements Parcelable
```

java.lang.Object
↳ android.app.Notification.Action

Structure to encapsulate a named action that can be shown as part of this notification. It must include an icon, a label, and a `PendingIntent` to be fired when the action is selected by the user.

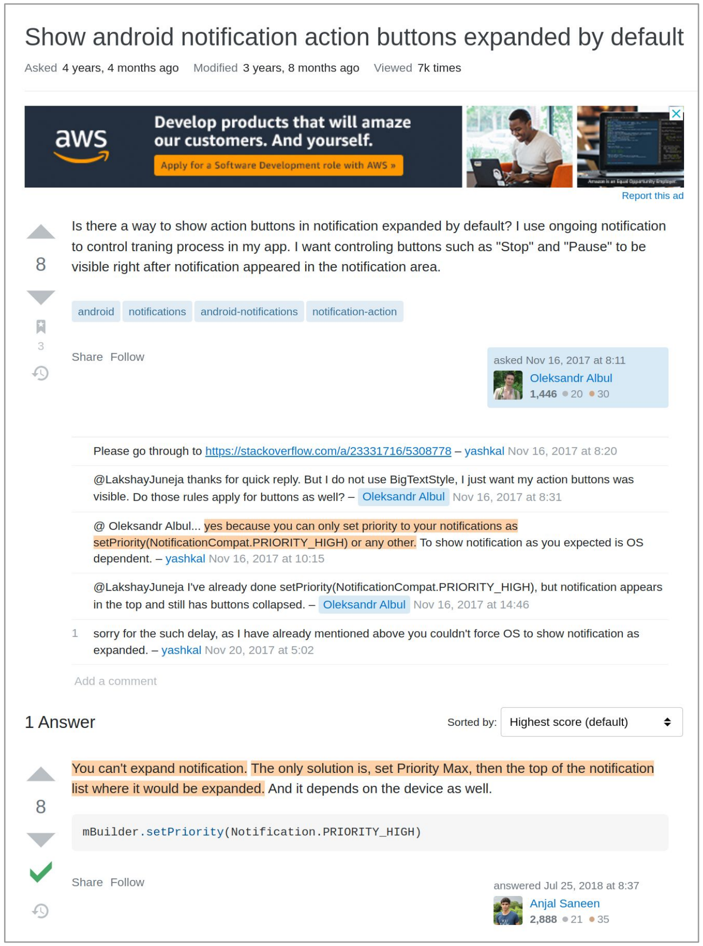
Apps should use `Notification.Builder#addAction(int, CharSequence, PendingIntent)` or `Notification.Builder#addAction(Notification.Action)` to attach actions.

As of Android `Build.VERSION_CODES.S`, apps targeting API level `Build.VERSION_CODES.S` or higher won't be able to start activities while processing broadcast receivers or services in response to notification action clicks. To launch an activity in those cases, provide a `PendingIntent` for the activity itself.

Summary

Nested classes	
class	`Notification.Action.Builder` Builder class for `Action` objects.
interface	`Notification.Action.Extender` Extender interface for use with `Builder#extend`.
class	`Notification.Action.WearableExtender` Wearable extender for notification actions.

Constants	
int	`SEMANTIC_ACTION_ARCHIVE` `SemanticAction`: Archive the content associated with the notification.



Show android notification action buttons expanded by default

Asked 4 years, 4 months ago Modified 3 years, 8 months ago Viewed 7k times

8

Is there a way to show action buttons in notification expanded by default? I use ongoing notification to control traning process in my app. I want controling buttons such as "Stop" and "Pause" to be visible right after notification appeared in the notification area.

android notifications android-notifications notification-action

3

asked Nov 16, 2017 at 8:11
Oleksandr Albul
1,446 ● 20 ● 30

Please go through to https://stackoverflow.com/a/23331716/5308778 – yashkal Nov 16, 2017 at 8:20

@LakshayJuneja thanks for quick reply. But I do not use BigTextStyle, I just want my action buttons was visible. Do those rules apply for buttons as well? – Oleksandr Albul Nov 16, 2017 at 8:31

@ Oleksandr Albul... yes because you can only set priority to your notifications as setPriority(NotificationCompat.PRIORITY_HIGH) or any other. To show notification as you expected is OS dependent. – yashkal Nov 16, 2017 at 10:15

@LakshayJuneja I've already done setPriority(NotificationCompat.PRIORITY_HIGH), but notification appears in the top and still has buttons collapsed. – Oleksandr Albul Nov 16, 2017 at 14:46

1 sorry for the such delay, as I have already mentioned above you couldn't force OS to show notification as expanded. – yashkal Nov 20, 2017 at 5:02

1 Answer

Sorted by: Highest score (default)

8

You can't expand notification. The only solution is, set Priority Max, then the top of the notification list where it would be expanded. And it depends on the device as well.

```
mBuilder.setPriority(Notification.PRIORITY_HIGH)
```

answered Jul 25, 2018 at 8:37
Anjal Saneen
2,888 ● 21 ● 35

Figure 1.4: Relevant text for the Android notifications task found across different kinds of artifacts, i.e., an API document (on the left) and a question-and-answer web page (on the right)

Although effective in specific contexts, it is reasonable to assume that these techniques might not apply to the different kinds of artifacts. This is either because these techniques have assumptions on the nature of relevant text that do not extend to the text found in other types of artifacts, or because these techniques rely on meta-data, which is only available in specific kids of artifacts and given how quickly developers progress to using new kinds of technology to record pertinent information (e.g., slack [94, 151]), it may be difficult to scale such artifact-centric approaches to cover the range of artifacts that a developer may encounter daily in their work [89].

This scenario illustrates thus some of the challenges in locating task-relevant textual information and motivating the need for more generalizable techniques.

1.2 Thesis

As there is an underlying structure to many software development tasks [117], we hypothesize that one can use information about a task to assist in the design of *artifact-agnostic* techniques able to automatically identify text relevant to a developer's task [27, 150]. We posit that:

> *A technique that applies to different kinds of natural language artifacts associated with software development assists a developer in correctly completing a software task by automatically providing them with text relevant to their task.*

To investigate this statement and to design a more generalizable technique, we ask what are common properties, if any, in the text deemed relevant to a software task? This question has been the focus of many software engineering studies [43, 83, 126, 127] that have explained qualitative aspects that guide a developer's decision on the relevance of natural language text [30, 49, 60, 150]. Researchers have also contributed with valuable corpora containing text annotated as relevant to particular tasks and artifacts [119, 134] as well as automatic approaches for extracting such text [39, 135, 163], but investigating if the properties of the text in certain artifacts apply to different kinds of artifacts is beyond their scope [31, 72].

To address gaps left by these studies on the relevance of natural language text and to provide a foundation for this work of dissertation, we present a formative study that *characterizes* task-relevant text found in different kinds of artifacts. We examine the text that twenty developers deemed relevant in the artifacts associated with six software tasks to investigate rules that can guide us to relevant text [81]. Analysis of the task-relevant text in bug reports, API documents and question-and-answer (Q&A) websites inspected as part of this study show consistency in the meaning, or *semantics*, of the text, suggesting that semantics might assist in the automatic identification of task-relevant text for the different types of artifacts a developer might use when performing a task [89, 109].

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 [164], searching for comprehensive code examples [146], or assessing the quality of information available in bug reports [40]. Nonetheless, few studies [97, 130] have investigated if and how accurately such approaches identify text likely useful to a developer's task across the different types of natural language artifacts available online.

To determine whether techniques that build upon semantic approaches apply to our domain problem, the second part of this thesis describes the investigation of a design space of six possible techniques that incorporate the semantics of words [51, 115] and sentences [59, 106] to automatically identify text likely relevant to a developer's task. Assessment of these techniques reveals that semantic-based techniques achieve recall comparable to a state-of-the-art technique [163], but without the need for artifact-specific data, and that some of our techniques also perform equivalently well across multiple artifact types, what strengthens the claim that semantic-based techniques are more generalizable.

Provided that we have found consistency in the text considered relevant within the natural language text of artifacts pertinent to a task and that semantic-based techniques can automatically identify such text; in the last part of this thesis, we examine the impact of tools that use semantic-based techniques to assist developers in completing a task.

We present a controlled experiment where participants had to perform two Python programming tasks when assisted (or not) by a tool that automatically identifies task-relevant text in a curated set of artifacts associated with the tasks considered. With this experiment, we compare the correctness of the solutions of each task performed by participants with and without tool assistance as well as the perceived usefulness of the text automatically identified and shown by the tool. Results indicate that participants found the text automatically identified useful in two out of the three tasks of the experiment and that tool support also led to more correct solutions in one of the tasks in the experiment. These results provide initial evidence on the role of semantic-based tools for supporting a developer's discovery of task-relevant information across different natural language artifacts.

1.3 Contributions

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

- It details a formative study that characterizes how task-relevant text appears 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 buil upon approaches that interpret the meaning, or semantics, of text to automatically identify task-relevant text across different kinds of software artifacts, where:
  - we report how accurately these techniques identify text that human annotators considered relevant to natural language artifacts associated with Android development tasks; and
  - we show that semantic approaches have accuracy comparable to a state-of-the-art approach tailored to one kind of artifact [163], i.e., Stack Overflow.
- It presents an empirical experiment that provides initial evidence on how an automated approach to task-relevant text identification assists a software developer while they work on a task.

This work of 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 associated with natural language artifacts that include annotations from 20 participants of text deemed relevant to the tasks in our study on characterizing task-relevant text;

- $\text{DS}_{android}$ is a dataset with 50 Android tasks and associated text artifacts that we produced to evaluate the semantic-based techniques that we assess for the automatic identification of task-relevant text;

- DS_{python} is a by-product of our empirical evaluation on the impact of tools that automatically identify task-relevant text; it contains three Python tasks and it includes annotations from 24 participants that indicated what text assisted them in writing each task's solution.

1.4 Structure of the Thesis

In Chapter 2, we present an overview of state of the art. The chapter details text analysis 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 and the qualitative factors related to locating task-relevant text.

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

Chapter 5 details the semantic-based techniques we investigate for automatically identifying task-relvant text. The first two techniques that the chapter presents use word embeddings to identify likely relevant text via semantic similarity and via a deep 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.

Chapter 6 details our empirical experiment investigating whether 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 as well as implications from our findings.

Chapter **??** concludes this work by reflecting on the contributions in the thesis and by outlining potential future work. *AM: Will review where I will have future work*

Chapter 2

Related Work

Performing a task on a software system, like fixing a bug or adding a feature typically requires a developer to consult a number of different kinds of artifacts, such as API documentation, bug reports, community forums and web tutorials. Developers produce such natural language artifacts on a continuous basis [132] and there has been a steep growth in the number of natural language artifacts available [27, 154].

Natural language artifacts have become intrinsically tied to software development [98], what led software engineering researchers to both investigate properties of the text appearing in these artifacts [52, 101] and design approaches that can be embedded in tools that mine the textual data available to assist software developer performing some task [27, 46, 69]. In Sections 2.1 and 2.2 we respectively provide background information on the significant body of work that investigates properties of the text in software engineering artifacts and the techniques that automatically identify text of interest to a particular task.

Most of the techniques that extract information from natural language artifacts, including the work presented in this thesis, are examples of applications that assist help developers in performing several tasks during their work day [110]. Other applications have exploited textual data to provide guidance to developers reading software documents [137, 153] or to explain solutions for programming tasks found in community forums [146]. We discuss these and other applications in Section 2.3.

2.1 Natural Language Text in Software Artifacts

Software engineering researchers have investigated several questions on the nature of the text in software artifacts. These studies have focused on different kinds of artifacts and have deepen researchers' knowledge on the richness of information in natural language artifacts, serving as a first step towards the design of automatic tools for identifying and extracting text from these artifacts [18, 101].

A significant body of work has proposed taxonomies for the text available in development mailing lists [52, 70], API documents [101], bug reports [18], and others. In written development emails, Di Sorbo et al. have found that text can be classified according to its purpose (e.g., feature request, solution proposal, etc.) [52], suggesting that text under certain categories might be useful for developers performing specific tasks. They have sampled 100 emails and identified that text for feature requests often contain expressions in the form suggestions (e.g., '*we should add a new button*') while solution proposals are often expressed in the form of attempts (e.g., '*let's try a new method to compute cost*').

Maalej and Robillard have inspected the Java SDK 6 and .NET 4.0. API documentation [101] and proposed a set of categories (e.g., directives, purpose, rationale, functionality, etc.) that define how knowledge is structured in API documentation. This taxonomy led Robillard and Chhetri to find that directives—or pieces of information that the programmers cannot afford to ignore, such as instructions on how to efficiently perform some I/O operation—often follow certain linguistic patterns [135], what made they produce a catalog of patterns that capture directives found in the Java SDK 6 documentation.

Rastkar et al. observe that much of the text in a bug report resembles a conversation between different subjects [133] while Arya et al. argue that due to this conversational nature, a single bug report contains several threads each pertaining to some topic, e.g., reproducing a bug, discussing potential solutions, testing, or reporting progress [18]. Although the topicality of the content in bug reports might encourage the design of tools that automatically identify such text, researchers have also found high variance in the text of different issues within a software project [38], what poses significant challenges to approaches that gather information from multiple bug reports.

Although the studies described above shed light on many aspects of natural language text in software artifacts, they do not discuss common properties of the text that is relevant to some software task found in different kinds of artifacts. In Chapter 3 we study text in API documentation, bug reports, and community forums that is considered relevant to a software task.

2.2 Automatic Text Identification Approaches

Information useful to a software task can be buried in irrelevant text or attached to non-intuitive blocks of text, making it difficult to discover [135]. In this section, we detail tools and approaches from related work that seek to assist developers in identifying information within the natural language text of software artifacts.

2.2.1 Pattern Matching Approaches

Pattern matching approaches rely on regular expressions describing a sequence of tokens that represent a relevant text fragment [27]. Tokens can either represent words or linguistic elements extracted using Natural Language Processing (NLP).

As examples of pattern matching approaches, DeMIBuD [39] and Knowledge Recommender (Krec) [101, 135] are tools that detect relevant sentences in bug reports and API documentation, respectively. These tools use a set of patterns derived from annotated data to identify relevant text. Krec uses 361 unique patterns to detect relevant sentences mentioning a code element (e.g., a class name or method signature) in API documentation [135]. Likewise, DeMIBuD uses a set of 154 discourse patterns to detect sentences relevant to understanding a bugs expected behaviour and steps to reproduce it, which are essential to bug triaging tasks.

In Stack Overflow posts, Nadi and Treude [120] have both applied the original set of patterns from Krec [135] and proposed heuristics that use conditional clauses (i.e., sentences with the word '*if*') to identify text that help a developer decide whether they want to carefully inspect a Stack Overflow posts or skip it.

Although the heuristics and regular expressions used in the aforementioned studies are often light-weight and effective [27, 101], pattern matching approaches are specific to certain kinds of domain and types of artifact [63], what limits using them in a general manner.

2.2.2 Summarization Approaches

Extractive text summarization techniques are used in natural language artifacts in software engineering to produce a summary of the artifact's content. A summary represents key information that may help a developer complete their task [27]. There are summarization techniques based on both supervised and unsupervised learning [116] and one can summarize the entire content of an artifact or content specific to a input query, as in *query-based* summarization [64, 71].

A number of summarization approaches target bug reports and GitHub issues, largely focusing on identifying key information within these artifacts. Rastkar and colleagues [134] use a supervised learning approach to summarize the content of bug reports showing that conversational features used to summarize emails [118] can also be applied to bug reports while Lotufo et al. [100] proposed an unsupervised summarization approach that automates the identification of sentences that a developer would first read when a inspecting bug report.

While many summarization approaches largely rely on lexical aspects in text, researchers have also made use of structured data available in software artifacts [41, 129, 153]. For example, Ponzanelli et al. proposed a summarization approach that uses the number of votes an answer has on Stack Overflow as a filter to the text in the summary for a Stack Overflow post [129]. As another example, DeepSum [91] pre-processes a bug report dividing sentences containing software elements, the reporter of the bug, and any other sentences in the bug report to produce summaries containing more diverse information.

A smaller number of summarization approaches have focused on producing task specific summaries [146, 163]. These approaches pose the problem of finding task-relevant text as a query-based extractive summarization problem and tools such as AnswerBot [163] identify relevant text in Stack Overflow posts based on the content of the text, how similar that content is with regards to a input query (i.e., task) and the structured data available on each of the answers in a Stack Overflow post (i.e., number of votes or whether an answer is the accepted answer).

Although we refrain from using structural data or assumptions on the content of an artifact because this might hinder the design of an automatic approach that identifies text relevant to some task in a more general manner, in Chapter 5 we compare the techniques that we explore in this thesis to the state-of-the-art.

2.2.3 Machine Learning Approaches

Machine Learning (ML) approaches take the text of a natural language software artifact and identify the sentences likely relevant to a particular software task using *supervised* or *unsupervised learning* methods [168].

Supervised learning approaches use a set of features and labeled data to train classifiers with the goal of identifying sentences relevant to certain software task. We have already presented supervised approaches that use text summarization (*i.e.,* [134]) and there are also approaches that identify relevant parts of software tutorials [73] or API documents [63, 101]. Despite their value, the cost and effort of hiring skilled workers to produce labeled data in software engineering artifacts has been a major limitation to the usage of supervised learning methods [17].

Unsupervised learning approaches do not require labelled data and determine relevant sentences according to properties inferred from the data. DeepSum [91] and Lotufo et al.'s [100] techniques are examples of unsupervised approaches in the scope of text summarization.

Other unsupervised approaches (*e.g.,* [74, 129, 130]) are mostly based around variations of the PageRank [122] or LexRank [56] algorithms. These algorithms represent all the text in an artifact as a graph. Then, they establish relationships (*i.e.,* weighted edges in the graph) between different sentences (*i.e.,* nodes in the graph) and select the nodes with highest weights as the most relevant ones. A crucial step in building the graph is in establishing relationships between nodes. Early approaches [74, 100] use Vector Space Model (VSM) [140] for this purpose while more modern ones [71, 146] use different word embeddings [32, 115], which we detail in Section 2.2.4.

2.2.4 Deep Learning Approaches

One substantial challenge of standard Machine Learning (ML) approaches is that researchers must engineer which features or properties of the text to use [57]. For example, Rastkar et al. uses conversational features in the text of a bug report to assist in determining which sentences to include in the bug report's summary [134] while Petrosyan and colleagues use linguistic and structural properties in the text of API documents to identify key text explaining API elements [125]. Given the specificity of such features, researchers have questioned the generalizability of standard ML approaches [63, 162].

In contrast to the human-engineered features, Deep Learning (DL) approaches allow the automatic extraction of features from training data through a series of mathematical transformations [50, 167]. Deep learning has led to groundbreaking advancements in many research areas (e.g., machine translation [99]) and, given its wide range of applications, we focus on its usage in natural language text appearing in software engineering artifacts [57, 92, 158].

Software engineering researchers have identified that the text in software task often differs from the text in the artifacts that are related to that tasks [71]. These so-called *lexical mismatches* [166] make it difficult to identify information of interest to a task and a number of studies have used DL neural embeddings [115] to bridge lexical gaps between the text of different software artifacts.

Neural, or word, embeddings produce vector representations in a continuous space, where words with similar meanings are typically close in the vector space model [68, 114]. Their usage has allowed researchers to improve the identification API elements pertinent to a programming task [166] or to more accurately assess the quality of the content in bug reports [40]. Word embeddings have become a common way to compare the semantic similarity of the text [113], being applied in query-based summarization techniques such as AnswerBot [163] or PageRank-based approaches such as HoliRank [130].

Many other DL studies in software engineering [57, 92, 158] use neural network architectures in a variety of software engineering tasks, including code comprehension [16, 112], community forum analysis [95, 157], or requirements traceability [42, 66]. DeepSum [91], which we described earlier, is an example of a summarization approach that uses an encoder-decoder architecture [44] to identify sentences of interest in bug reports. As another example, Xi et al. [161] use a Convolutional Neural Network (CNN) [86] to identify sentences pertaining to Di Sorbo et al.'s topics [52].

Few studies in software engineering have considered more modern neural networks [158] able to leverage hidden features not only between words but also between sentence pairs (e.g., BERT [51]). These models might assist in determining implicit relationships between the text in a task and the text in a relevant sentence within a software artifact. As such, Chapter 5 describes how we use BERT for automatically identifying text relevant to a software task.

2.3 Improving Developers' Productivity

The tools and approaches presented in Section 2.2 are examples of studies that help developers in locating information that assists them to complete a software task. These studies fit in the bigger context of software engineering research that facilitates or improves the quality of a developer's work [79, 110, 143].

Researchers have had a long interest in making knowledge bases that developers working on a task can benefit from. Hipikat [46] is a seminal tool that exemplifies this concept in action. It automatically tracks the artifacts used by a developer as part of a development tasks so that it can recommend these artifacts to any future developers who work on similar tasks [46]. Other tools like Strata summarize knowledge produced by developers while they navigate on the web so that other developers have a a head start when performing similar tasks [98].

As part of their work, developers engage in many sensemaking and decision making activities [145] and several studies have investigated how to provide means to better assist developers in performing such activities [24, 97, 98]. For example, tools such as Deep Intellisense display code changes, filed bugs, and forum discussions mined from an organization's database to better assist a developer understand the rationale behind the code they currently work on [69], and researchers have extended this idea to publicly available data, e.g., GitHub issues [156] or pull requests [61].

Although the studies mentioned above do not directly relate to this work of dissertation, we build 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 [98], what assisted in the design of our experiment for evaluating an automated approach to task-relevant text identification (Chapter 6).

Chapter 3

Characterizing Task-relevant Text

In the last chapter, we described several studies that analyze text in software engineering artifacts and many approaches that attempt to automatically extract relevant information from these natural language artifacts.

Although certainly valuable, we have established that most of these studies focus on specific kinds of artifacts. If we seek to design techniques that automatically determine relevance regardless of an artifact's type, we must determine whether there is consistency in the text that software developers identify as relevant to a task across different kinds of artifacts, and if there is consistency, we also ask what are common relevance cues in the portions of the text identified as task-relevant?

To answer these questions, this chapter presents an *empirical study* in which 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 [48, 76]. We also report the participants' reasoning process for determining relevance, which we gathered through interviews that we use to identify common themes about their approaches [149].

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

Chapter 4

Android Task Corpus

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

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

Figure 4.1 summarizes the process we use to create the corpus, which we call $\text{DS}_{android}$. We randomly sample a set of 50 Android development tasks from two

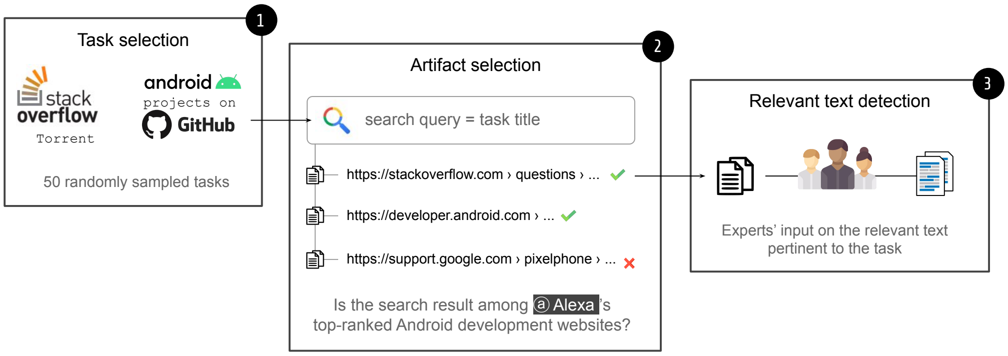

1
Task selection
stack overflow
Torrent
android
projects on
GitHub
50 randomly sampled tasks
2
Artifact selection
search query = task title
https://stackoverflow.com › questions › ... ✓
https://developer.android.com › ... ✓
https://support.google.com › pixelphone › ... ✗
Is the search result among Alexa's
top-ranked Android development websites?
3
Relevant text detection
Experts' input on the relevant text
pertinent to the task

Figure 4.1: Summary of procedures for corpus creation

common task sources, GitHub[1] and Stack Overflow[2]. For each of these tasks, we use the Google search engine to find potential artifacts that likely contain relevant information for that task in top-ranked Android development websites. We then gather task-relevant sentences from these artifacts from experienced developers. We detail each of these steps in turn in Sections 4.1, 4.2 and 4.3. We provide a summary of the final created corpus in Section 4.4.

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

4.1 Software Tasks

We start corpus creation by identifying software tasks for which a developer will likely benefit from the use of additional information to complete. We scope task selection to *Android development* because the Android software development kit (SDK) evolves constantly due to functionality, security and performance-related improvements [90, 107]. These improvements impact its development community, requiring them to often seek information regarding changes in the SDK [28, 96, 108]. For example, over 35,000 developers have used Q&A forums to discuss tasks covering 87% of the classes in the Android API [123].

Two common places where Android task can be found are:

- the description of an issue (e.g., a bug or feature request) reported in an issue

[1] https://github.com/
[2] https://stackoverflow.com/

tracking system; or in

- a post in a community forum, development mailing lists, and others.

Several studies have used such sources for software tasks [18, 20, 120, 163] and, following the lead of these studies, we select GitHub issues and Stack Overflow (SO) posts on Android development as the two sources for the tasks in our corpus.

GitHub tasks

To select tasks from GitHub, we are guided by studies that use stars [33, 34] as a proxy for a project's popularity [58, 160]. We selected 14 projects, ranging from mail clients[3] to development frameworks[4], by filtering the list of top-starred projects in GitHub to those with the *Java* and *Android* tags. We then randomly selected 25 distinct issues originating from these starred projects as the GitHub tasks of our corpus (average of 1.78 issues per project). While selecting issues, we took care to check that they had at least one follow-up comment and that the issue title did not contain certain words, e.g., `test` or `ignore`, as these words indicate issues created automatically by scripts or bots—a common pitfall that researchers must be aware of when mining GitHub [78].

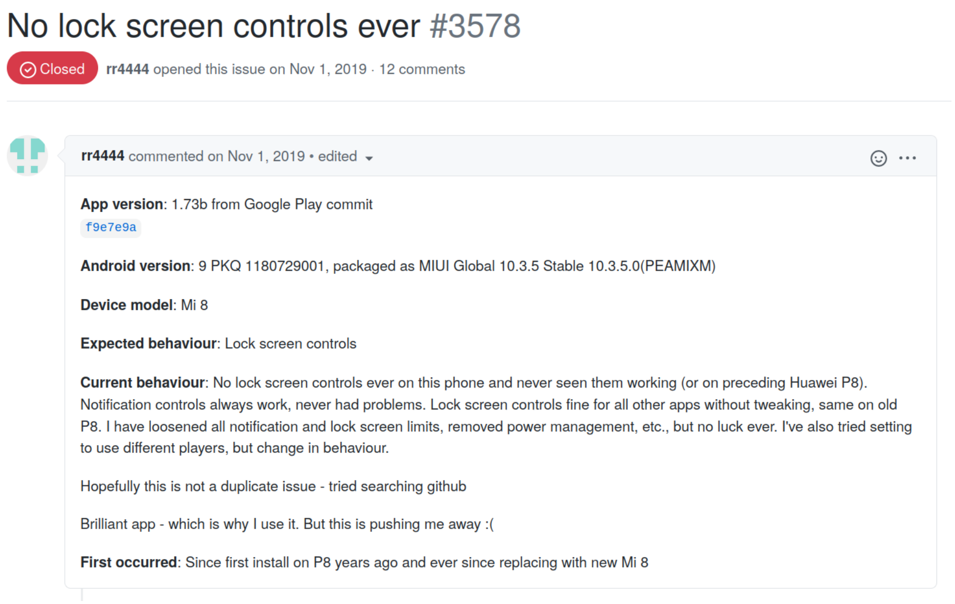
Figure 4.2 shows an example of a GitHub task in our corpus. Although the expected behaviour is that the app controls should be visible even with the screen locked, a user reports that the app screen is missing. A developer addressing this issue might need to review the Android lock task documentation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

Stack Overflow tasks

We consider Stack Overflow posts as software tasks because to answer a post, a developer often needs to provide references supporting their answer [165]. Finding these references in a timely manner and writing the key information that helps a

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

[4] https://github.com/libgdx/libgdx



No lock screen controls ever #3578

rr4444 opened this issue on Nov 1, 2019 · 12 comments

rr4444 commented on Nov 1, 2019 • edited

App version: 1.73b from Google Play commit `f9e7e9a`

Android version: 9 PKQ 1180729001, packaged as MIUI Global 10.3.5 Stable 10.3.5.0(PEAMIXM)

Device model: Mi 8

Expected behaviour: Lock screen controls

Current behaviour: No lock screen controls ever on this phone and never seen them working (or on preceding Huawei P8). Notification controls always work, never had problems. Lock screen controls fine for all other apps without tweaking, same on old P8. I have loosened all notification and lock screen limits, removed power management, etc., but no luck ever. I've also tried setting to use different players, but change in behaviour.

Hopefully this is not a duplicate issue - tried searching github

Brilliant app - which is why I use it. But this is pushing me away :(

First occurred: Since first install on P8 years ago and ever since replacing with new Mi 8

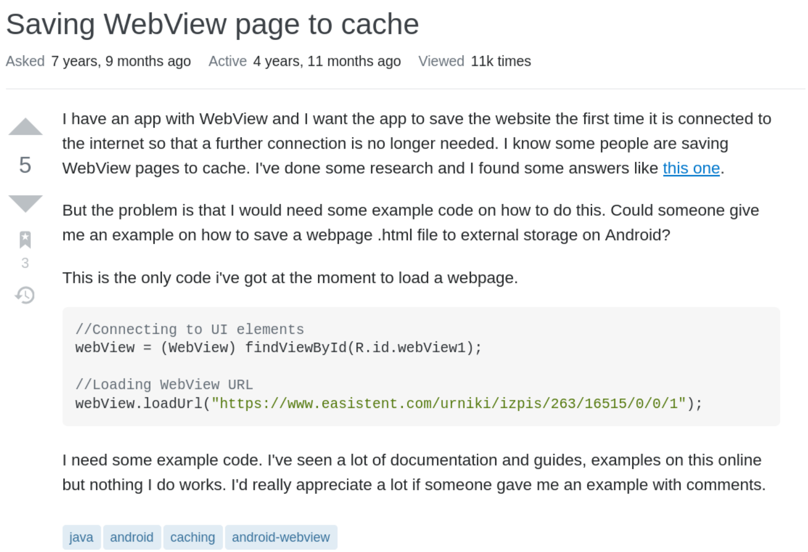
Figure 4.2: Sample GitHub task from our corpus

user understand the provided solution encompasses many of the activities found in a developer's daily work, e.g., work-related browsing, coding, debugging, and reading/writing documentation [110]. For example, Figure 4.3 depicts a task where a developer describes her struggles using the Android WebView component. To answer this question, a developer will not only provide a code snippet, but also explain key points of the Android WebView API [6] and how they were used in the solution provided to the task, as presented in Figure 4.4.

We randomly select 25 Stack Overflow posts from a curated list about Android development [23]. This list was built by Baltes et al. using the Stack Overflow dump published on June 5, 2018 [21, 22] and it contains 209,536 unique posts with the *Java* and *Android* tags.

4.2 Artifact Selection

When selecting artifacts pertinent to a task in our corpus, we seek to simulate everyday practices on how developers search the Web [131, 161]. We formulate a query for each task and use a Web search engine to retrieve artifacts that are



Saving WebView page to cache

Asked 7 years, 9 months ago Active 4 years, 11 months ago Viewed 11k times

I have an app with WebView and I want the app to save the website the first time it is connected to the internet so that a further connection is no longer needed. I know some people are saving WebView pages to cache. I've done some research and I found some answers like this one.

But the problem is that I would need some example code on how to do this. Could someone give me an example on how to save a webpage .html file to external storage on Android?

This is the only code i've got at the moment to load a webpage.

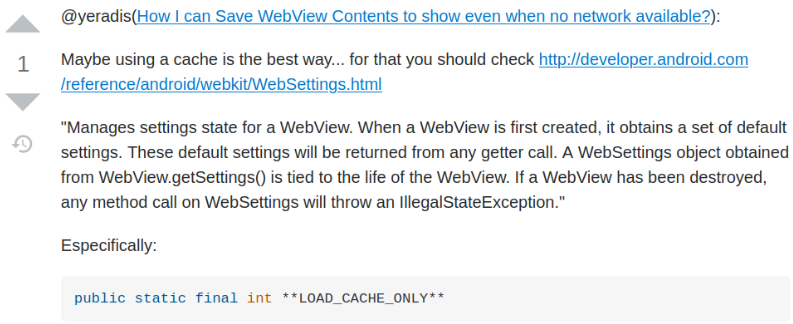
```
//Connecting to UI elements
webView = (WebView) findViewById(R.id.webView1);

//Loading WebView URL
webView.loadUrl("https://www.easistent.com/urniki/izpis/263/16515/0/0/1");
```

I need some example code. I've seen a lot of documentation and guides, examples on this online but nothing I do works. I'd really appreciate a lot if someone gave me an example with comments.

java android caching android-webview

Figure 4.3: Sample Stack Overflow question



@yeradis(How I can Save WebView Contents to show even when no network available?):

Maybe using a cache is the best way... for that you should check http://developer.android.com/reference/android/webkit/WebSettings.html

"Manages settings state for a WebView. When a WebView is first created, it obtains a set of default settings. These default settings will be returned from any getter call. A WebSettings object obtained from WebView.getSettings() is tied to the life of the WebView. If a WebView has been destroyed, any method call on WebSettings will throw an IllegalStateException."

Especifically:

```
public static final int **LOAD_CACHE_ONLY**
```

Figure 4.4: Excerpt of a Stack Overflow answer

pertinent to that task, as described below.

Artifact sources

The artifacts sought to find useful information or knowledge for completing a task depend on the type of task a developer performs. For example, when using a new framework or library, a developer refers to official API documentation [89, 136] while, for debugging or error diagnostic tasks, community-based sources are preferred [36, 89]. Despite such variability, researchers have observed that Web blogs, API documentation, and community forums are sources commonly used by developer to forage information that assists task completion [75, 89].

We use this knowledge to restrict artifact selection to well-known and studied artifact types within these sources [80, 89, 150], namely Android and Java SE API documentation, GitHub issues, Stack Overflow answers, and Web tutorials or blog posts on Java and Android development.

Query formulation

Coming up with proper search terms is a critical step of any search [67] and, ideally, we should be able to formulate a query with terms able to retrieve the most pertinent artifacts for a software task. However, studies have shown that developers perform poorly in identifying good search terms [80, 87, 150] and thus, using a task's title as an educated approximation to terms that a developer might use is a common procedure adopted by other studies in the field (e.g., [163] or [146]). Hence, we use a task's title (i.e., SO question or GitHub issue title) as the seed to search for pertinent artifacts.

Search results

We use `googlesearch` API [11] to request up to 5 resources per query adding `site:domain` to search for artifacts only in (or outside) a given web domain[5]—procedures similar to [163].

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

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

Limiting the number of artifacts up to a maximum of 5 artifacts per task relates to the time-consuming [15] and cognitively demanding [127] nature the final step in the dataset creation, i.e., asking annotators to carefully read, understand and identify the text within the fetched artifacts that relevant to a given task, as Section 4.3 further details.

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

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

Artifact's content

Last, we need to extract the natural language text within an artifact so that techniques that automate the identification of text relevant to a task can be built using our corpus. This step requires processing an artifact's content into a sequence of individual sentences, what prompted us to follow common procedures for processing the artifact types found in our corpus [18, 120]. That is, given a search result `URL`, we use use a series of python APIs[6] to fetch the artifacts' content and then, we use the Stanford CoreNLP toolkit [1] to identify individual sentences in the artifacts' content.

4.3 Relevant text detection

Next, we need to determine which of the text in the gathered artifacts could provide information that assists a developer in solving her task. In our corpus, this text represents *golden data* that one can use to design and evaluate automatic tools that assist developers in the identification of information useful to their tasks. To produce it, we ask experienced developers to mark the text that they deem useful and that provide information for tasks assigned to them [105, 120, 135].

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

4.3.1 Annotation process

Our intention is that golden data reflect text that instructs developers to perform important actions to accomplish their task [100, 135]. To this end, we describe the annotators' background, annotation procedures, and the the text inspected by the annotators.

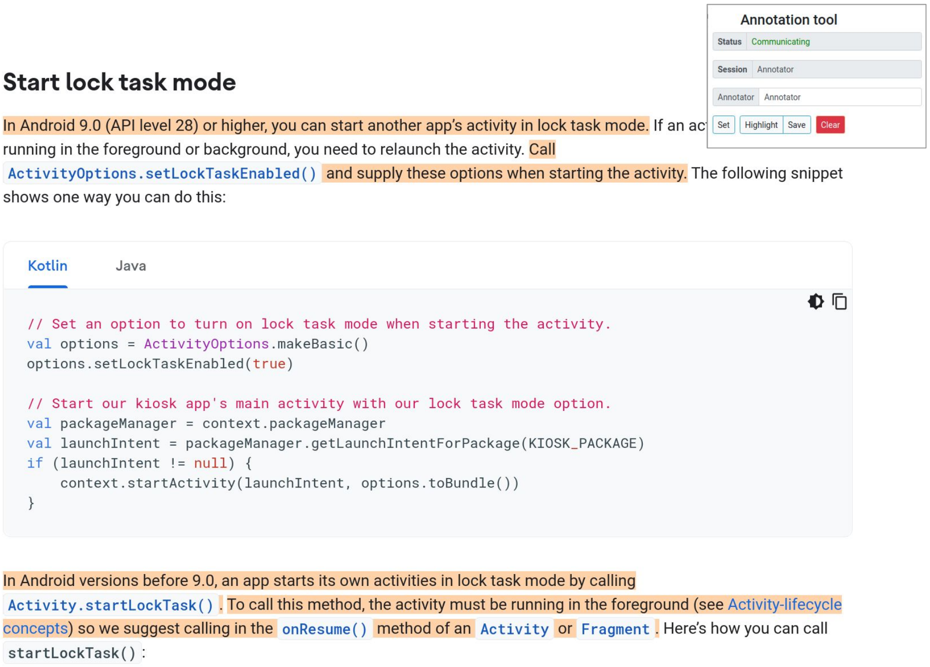
Annotators

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

Annotation procedures

The annotation process started with two tasks, other than the ones in our dataset, so that annotators could familiarize themselves with annotation procedures. For each task, annotators had the task description and links to artifacts pertinent to the task at their disposal. We asked annotators to write a short (250 words max [134]) with instructions that a developer could follow to complete the task successfully. The purpose of the plan was to ensure that annotators built enough context about the task. While perusing artifacts, annotators also had to manually highlight sentences that they deemed useful and that provided information that assisted task completion—instructions similar to the ones used for the creation of the data in Nadi and Treude's study [120] or in the $\text{DS}_{synthetic}$ corpus [105].

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



Annotation tool

Status Communicating

Session Annotator

Annotator Annotator

Start lock task mode

In Android 9.0 (API level 28) or higher, you can start another app's activity in lock task mode. If an ac running in the foreground or background, you need to relaunch the activity. Call `ActivityOptions.setLockTaskEnabled()` and supply these options when starting the activity. The following snippet shows one way you can do this:

```
// Set an option to turn on lock task mode when starting the activity.
val options = ActivityOptions.makeBasic()
options.setLockTaskEnabled(true)

// Start our kiosk app's main activity with our lock task mode option.
val packageManager = context.packageManager
val launchIntent = packageManager.getLaunchIntentForPackage(KIOSK_PACKAGE)
if (launchIntent != null) {
    context.startActivity(launchIntent, options.toBundle())
}
```

In Android versions before 9.0, an app starts its own activities in lock task mode by calling `Activity.startLockTask()`. To call this method, the activity must be running in the foreground (see Activity-lifecycle concepts) so we suggest calling in the `onResume()` method of an `Activity` or `Fragment`. Here's how you can call `startLockTask()`:

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

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

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

Text inspected

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

Table 4.2 gives insight into the number of sentences inspected per artifact type. We observe that API documents and miscellaneous Web pages contain the highest number of sentences in our corpus. This is not surprising because API documents often contain boilerplate text and all the information needed for the usage of an API element is usually found in a single document [136]. Miscellaneous Web pages comprise blogs or tutorials, which often provide step-by-step instructions and accompanying examples, what likely explains the large number of sentences for this type of artifact [19, 73].

The content of GitHub issues mostly resembles conversations [134] and, beyond code snippets and minimal structured fields, the 1,890 sentences inspected in this type of artifact comprise the description of a reported bug or a feature request as well as questions, answers, and discussion from community members who are interested in resolving the issue at hand [170].

For SO answers, the fewer number of inspected sentences (1,420) potentially relates to the fact that Stack Overflow offers little incentive for further discussion once a post has been answered. Nonetheless, it is worthy noting that sentences with crucial information are not limited to the ones in an accepted answer [120] and that later comments can be equally or more informative, often providing notes about updates in an API component or framework [169].

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

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

4.3.2 Corpus Description

Overall, the resultant $\text{DS}_{android}$ corpus has 50 tasks and 133 artifacts, divided as 33 API documents, 45 Stack Overflow answers, 20 GitHub issues, and 35 other web pages. Tasks in the dataset have an average of 3 associated artifacts each and we discriminate the text marked by each annotator per artifact. Overlap between the text marked by the annotators represents 30% of the entire data and *Krippendorff's alpha* [85] indicates good reliability[121] over the text marked as relevant (or not) by the annotators ($\alpha = 0.69$).

We structure tasks and annotation results in a format similar to the one shown in Table 4.3. Each task has a title, link, description, and a set of pertinent artifacts. In turn, each artifact has a type, title, link, and content, where each sentence within an artifact's content is preceded by the set of annotators (i.e., `none`, `A1`, `A2`, or `A3`) who marked the sentence as useful to the task. As an example, in a Stack Overflow answer (artifact$_2$), all three annotators marked the sentence "*Have you checked RemoteControlClient?*" as relevant to the lock mode task.

Overall, annotation required a total of $\approx$ 45 hours of manual work and it was done throughout the course of 5 weeks. Table 4.4 provides summary statistics for the text marked by the annotators over all the artifacts inspected as well as on an artifact type basis according to the text marked by any annotator. On average the text deemed useful to a software task in the artifacts inspected comprises 8.93 sentences per artifact per annotator. We observe that the highest number of sentences marked originate from miscellaneous artifacts while the lowest come from GitHub issue discussions. The high number of sentences marked in the former might relate to the more didactic nature of web tutorials [19, 74]. For the latter, the content on GitHub may be too project-specific what might prevent a developer performing a task in a different context from benefiting from most of the content found in this type of artifact. For example, discussions about whether a certain design is the most appropriate [156] might not extend to other projects, but instructions on how to use an API element might.

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

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

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

$^{\text{annot}}$ annotator(s);

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

Table 4.2 also reports the percentage of sentences marked by one, two or the three annotators who inspect the artifacts in the corpus. Our rationale to provide this data is that individuals might use different criteria to assess the usefulness of a sentence to a given task [25, 26]. Out of 1393 unique marked sentences, 10% were marked by all annotators, 19% by two of them and the remainder—996 sentences—by a single annotator. These ratios follow Nenkova and Passonneau's empirical findings on content selection [121] and are a similar to the ratios in the DS$_{synthetic}$ corpus, where sentences marked by a single annotator comprise the majority of the data. Although we leave the in-depth analysis of how an individual's background plays a role in what they perceive as relevant for future studies, evaluation metrics should outline how the differences between the text marked are taken into account when using the golden data provided in our corpus for evaluation purposes.

Based on the text marked by the annotators, we also ask if the amount of information required to solve a task was similar in GitHub or Stack Overflow tasks. Answering this question will allow us to know whether the corpus can be evenly used for evaluation purposes or whether we must distinguish tasks according to their origin. To answer this question, Table 4.5 shows the number of sentences marked by any annotator according to a task's origin. We use a Wilcoxon-Mann-Whitney test [102] to check if there are any differences between the average number of sentences marked per type of artifact in GitHub and Stack Overflow task. Results show there is no statistically significant difference between the number of sentences sentences marked. Hence, the corpus can be used for evaluation purposes without the need to distinguish tasks based on their origin.

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

Table 4.5: Number of sentences marked per type of task

4.4 Summary

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

Chapter 5

Identifying Task-Relevant Text

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

In this chapter, we explore a design space of possible techniques building on approaches to interpret the meaning, or semantics, of text to identify task-relevant text across different kinds of software artifacts. We introduce six possible techniques that incorporate the semantics of words and sentences. We show that some of the proposed semantic-based techniques compare to existing artifact-specific techniques and that they apply to a broader set of artifacts.

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

5.1 Problem Statement

Performing a task on a software system, like fixing a bug or adding a feature, typically requires a developer to consult a number of different kinds of artifacts, such as API documentation, bug reports, community forums and web tutorials. When consulting these artifacts, a developer must identify, from the large amount of text in these documents, just the fraction of text relevant to the task-at-hand.

Unfortunately, developers tend to struggle with identifying the relevant text and when they are unable to locate all, or most, of it, they may produce incomplete or incorrect solutions [117]. To aid developers in this situation, researchers have proposed various techniques and tools to identify likely relevant text. For example, *Krec* [135] identifies text that a developer cannot afford to ignore when reading an API document. As another example, *AnswerBot* [163] generates answers for a developer's task by identifying text pertinent to that particular task on Stack Overflow. Although effective, these techniques target specific types of artifacts, limiting their use across the many different kinds of artifacts developers encounter daily in their work.

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. Guided by recent success using techniques that interpret the meaning, or *semantics*, of text for a variety of development activities, such as for finding who should fix a bug [164], searching for comprehensive code examples [146], or assessing the quality of information available in bug reports [40], we ask:

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

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

5.2 Techniques

We introduce six novel semantic techniques that might aid developers in identifying task-relevant text in software artifacts. Three of the techniques use word2vec [115] as a base for determining a ranked list of similar sentences between an artifact and a task description and three use BERT [51]. Each of these technique takes as input a task description and an artifact. The technique then identifies sentences in the artifact pertinent to the task. The output from the technique is a ranked set of sentences from the artifact from most likely to be helpful for solving the task to the least helpful.

Table 5.1 provides an overview of the six techniques. Two techniques simply returns the results of *word2vec* or *BERT*. The four others apply *frame semantics* [59] to filter each of the sentences returned by the base techniques. A first filter—*frame-elements*—retains only sentences that include certain frames. The second filter—*frame-associations*—keeps sentences with certain co-occurrences of frame semantics between the task description sentences and the artifact sentence. We describe word2vec, BERT and the filters in turn to explain the six techniques.

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

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

5.2.1 Identifying Task-relevant Text with word2vec

To form the base of the first three techniques, we use word semantics via word2vec in conjunction with information retrieval approaches. Information retrieval techniques are usually applied to look for documents relevant to a query prompted by a user [27], where the relevance of a document to a query is based on some function that indicates how similar the document's content is to that query [104]. Given the right *similarity function*, information retrieval techniques can be also applied to identify sentences within a document, artifact in our problem formulation, that are relevant to a task.

We use the *Skip-gram* model [115], also referred to as *word2vec*, to define such a similarity function. The *Skip-gram* model exploits Harris' distributional hypothesis [68]—which states that words that appear in a similar context tend to have similar meanings—and builds vector representations, namely *word embeddings*, for each of the words in a text corpus. With a significantly large text corpus, the model associates similar vector embeddings to words that are similar in meaning [166].

Since *word2vec* represents words through a vector space, the similarity of two words i and j can be obtained computing the cosine similarity [104] between the corresponding vectors of each word, i.e., w_i and w_j:

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

In turn, the similarity of two sentences can be obtained by first computing a vector representation for the entire sentence and then, by measuring the cosine angle between the obtained sentence vectors. To obtain sentence vectors, we follow Conneau et al.'s guidelines [45] and we average the sum of the embeddings for each word in a sentence.

Following these procedures, we use a Skip-gram model with word embeddings trained for the software engineering domain [54] to compute the semantic similarity, or relevance, of the sentences $\{a_1, a_2, \ldots, a_n\}$ within a pertinent artifact A and the sentences $\{t_1, t_2, \ldots, t_m\}$ in a task T. As both entities have multiple sentences, the semantic similarity of a sentence $a_i \in A$ is the maximum value obtained for this sentence and each of the sentences in a task.

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

5.2.2 Identifying Task-relevant Text with BERT

In the Skip-gram model, context refers to the sentences used to train the model. This approach to context does not allow the model to disambiguate words based on their surrounding text. In other words, a Skip-gram model will have a single vector representation for a word such as '*company*' even when it can have different meanings, i.e., a business organization or being with someone. In contrast, the Bidirectional Encoder Representations from Transformers (BERT) [51] provides different representations for the same word based on the sentence in which a word appears. This additional layer allows the model to perform more complex operations, leading to state-of-the-art results in several tasks [51].

Typically, BERT is trained on a massive amount of data. During training, a percentage of the tokens in a sentence—usually 15%—are replaced with a special token and the model is optimized to predict these replaced, or *masked*, tokens. The model relies on a mechanism, called *attention* [155] that correlates and weights all non-replaced tokens in the input so that the model maximizes predictions. To fully train a BERT model, one first creates a base model using a large amount of text and the general token prediction task. Then, this base model is fine-tuned to specific tasks, such as text classification, using a dataset specific to the fine-tuning steps. The procedures of *training* and *fine-tuning* allow using the model even when training data is scarce and the model transfers what it learned during training steps to the fine-tuning steps [51].

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

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

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

5.2.3 Filtering Task-relevant Text with Frame Semantics

Although word2vec and BERT have provided significant improvements to a diverse set of natural language tasks, they still capture semantics at the word level. If we seek to infer a sentence's meaning, an alternative would be to consider *frame semantics* (Chapter 3).

We define two filters using frame semantics that can be applied with either the word2vec or BERT base techniques as a post-processing step to identify sentences potentially relevant to a task based on the frame elements that the *SEFrame* tool identifies [105]. For that, we implement a set of filters that can be applied to the previous techniques as a post-processing step [104]. A filter takes the output of a technique O and produces output O' where a sentence must match the filter's relevance criteria. For both filters, we use SEFrame [106] to assign frames relevant to software engineering text.

Frame-elements

This filter considers sentences identified with frames giving instructions, describing required events, explaining system procedures, and others as likely to contain information of interest. The rationale for the set of frames identified is based on findings from related work that have shown that the meaning of the text is an indicator of its relevance to a software task [105]. This filter takes a sentence $a_i \in A$ and checks whether any of the frame elements of this sentence match one meaningful frame $f \in F$ from a pre-established set[1].

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

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

Frame-associations

This filter considers that the relevance of a sentence depends on the intentionality of a task. For example, for a task that requires diagnosing an error, sentences in a bug report that describe success or failure of an action are likely relevant. This rationale is represented by a set of associated frame pairs, where each element represents a frame originating from a task and another frame originating from a sentence. This filter checks if a task-artifact frame pair obtained from an input artifact and input task appear in a pre-established set of pairs[1].

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

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

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

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

5.3 Evaluation

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

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

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

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

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

We use *precision* and *recall* metrics [103] to measure what portion of the text identified by human annotators in the $\text{DS}_{android}$ corpus the techniques automatically identify. In this context, we believe recall to be the most important metric since failure to identify text that is relevant to a task means that a developer will have an incomplete or partial view of the information needed, what can lead to faults [117].

5.3.1 Experimental Setup

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

Techniques' Output

We configure each technique to identify a target number of 10 relevant sentences for each task-artifact pair. This decision is based on the fact that no more than 15% of the content of any artifact in our dataset is deemed relevant to a task, which, on average, accounts for 8.93 sentences (Chapter 4). Researchers have also used the same target number of 10 sentences when evaluating techniques (e.g., [163] or [100]) able to identify relevant text for certain kinds of artifacts, what will also facilitate comparing our results to related work.

Training & Testing Data

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

Metrics

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

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

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

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

Table 5.2: Evaluation outcomes

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

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

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

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

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

5.3.2 Comparison to AnswerBot

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

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

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

Table 5.3: Comparison to AnswerBot

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

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

Results

Table 5.3 shows values of precision and recall for AnsBot and the semantic techniques that we explore. In the table, rows provide details about a specific technique while columns discriminate precision and recall values and which post-processing filters were applied. We also mark results from paired Wilcoxon-Mann-Whitney tests [102] that check if the results between each technique and AnsBot are statistically different.

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

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

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

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

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

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

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

5.3.3 Evaluation Across all Artifact Types

Provided that we can identify task-relevant textual information with accuracy comparable to a state-of-the-art approach, we measure how much of the text that is relevant to a task (within an artifact) can our semantic-based techniques identify. To this end, we compute precision and recall metrics over all the artifacts in the test data. To assist comparisons, we also report results for the text identified by a standard VSM lexical similarity approach. Several studies [62, 83, 105] have shown that developers often use keyword-matching as a first search strategy to locate text that might contain information relevant to their tasks. Thus, this baseline might assist us in interpreting how much of the task-relevant text a developer would identify by themselves when inspecting an artifact at first glance.

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

Results

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

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

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

When we compare techniques without any filters, differences between `word2vec` and `BERT` may be explained by how each of these two approaches compute relevance. That is, all the words in a sentence contribute equally to `word2vec`'s identification of task-relevant text via its semantic similarity function. If an artifact lacks much of the text that appears in a task description, this technique may not be able to identify the text deemed relevant. BERT shortens this gap because relevance is computed via its attention mechanism, which might assign more weight to words key to determining that the text is relevant to a task thus, explaining why the technique has better evaluation metrics.

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

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

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

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

5.3.4 Threats to Validity

We rely on the golden data for $\text{DS}_{android}$ to evaluate techniques, which impacts the conclusions we draw. This golden data in turn relies on the human annotations of text in artifacts that is related to a task-at-hand (Chapter 4). There is a possibility that this text marked by the annotators might not contain information associated with the solution for the task, or that an annotator missed highlighting text that they deemed relevant. We minimize this threat by considering only the sentences marked by at least two annotators as the golden data used in the evaluation of techniques. We also asked annotators to write a short set of instructions for the tasks they annotated and provided them an in-house tool that streamlined annotation procedures. These procedures help mitigate risks to the validity of our conclusions. We note that there are no objective means to quantify which text assists task-completion [141, 142].

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

The selection of tasks in the Android development domain could affect the generalizability of our work. Most notably, aspects such as programming languages, frameworks, associated technologies, and others [20] influence the information sought by a developer as well as what they find relevant. The three annotators that we recruited to construct the $\text{DS}_{android}$ dataset are also not representative of the entire developer population. We leave the investigation of software tasks in other programming languages or domains—and their associated natural language software artifacts—to future work.

5.4 Summary

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

Chapter 6

Evaluating an Automated Approach to Task-Relevant Text Identification

In the last chapter, we showed that semantic-based approaches can help identify text in artifacts relevant to a task. In this chapter, we consider whether these approaches can assist a software developer while they *work* on a task.

To investigate how semantic-based approaches might assist developers perfoming a task, we have designed a tool, in the form of a web browser plug-in, that automatically highlights text relevant to a particular software task. This plug-in takes a developer's task and a software artifact as inputs and then, it applies its underlying semantic-based technique—*BERT with no filters*—to identify sentences that likely contain information useful to the input task.

We investigate benefits brought by using this tool, if any, through a controlled experiment where 24 participants completed three Python programming tasks using (or not) this tool. With this experiment, we show that participants considered the majority of the text automatically identified in the artifacts they perused useful and that our tool assisted them in producing a more correct solution for one of the experimental tasks.

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

(Section 6.3). Section 6.4 concludes the chapter.

6.1 Motivation

Our goal is to examine how a tool that automatically identifies task-relevant text in pertinent documents can affect a developer's work. Building on work presented earlier in this thesis, the tool uses a semantic-based technique. By identifying information that is useful to the developer's task, a developer's burden to find task-relevant information [135] can be lowered, allowing them to focus their time on other activities such as judging how the information found applies to their task.

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

Even if a developer is more successful with the tool than without, there is a chance that the text shown by the tool is not *useful*—either because it is not relevant for the task at hand or because it is unsurprising, that is a developer finds the text identified by the tool as "common-knowledge" [47, 135]. Our experimental design incorporates the gathering of qualitative data to assess the usefulness of the text identified.

6.2 Experiment

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

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

*RQ*2 How useful is the text identified by a semantic-based tool, i.e., does the text automatically identified contains information that assists task completion?

*RQ*3 How does the text automatically identified by our semantic-based tool compare to text that humans perceive as relevant to a task?

To answer these questions, we designed an experiment where 24 participants with software development background each attempted a *control* and *tool-assisted* task randomly drawn from a list of well-known Python programming tasks. Participants are asked to write a solution for both of their assigned tasks and we use the control task to collect what text a participant deems relevant to the task at hand. In the tool-assisted task, we gather input on the usefulness of the text automatically identified and shown by our tool. This design, summarized in Figure 6.1 and detailed in the following subsections, allow us to:

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

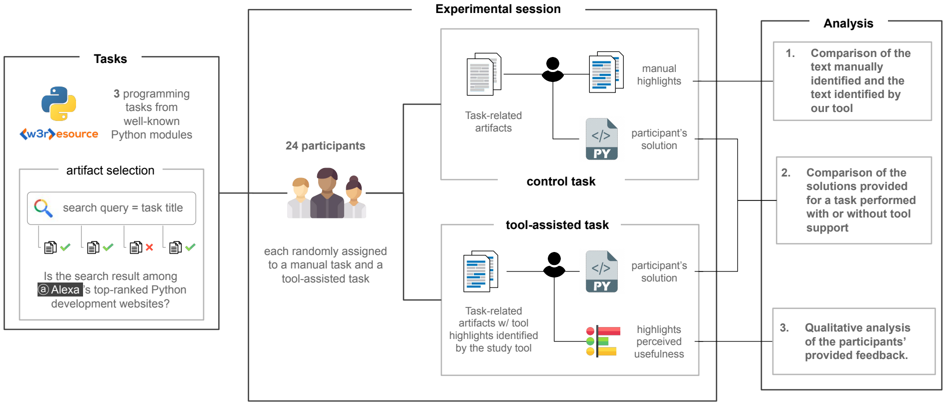

Tasks
3 programming tasks from well-known Python modules
w3resource
artifact selection
search query = task title
Is the search result among @Alexa's top-ranked Python development websites?
Experimental session
24 participants
each randomly assigned to a manual task and a tool-assisted task
Task-related artifacts
manual highlights
participant's solution
control task
tool-assisted task
participant's solution
Task-related artifacts w/ tool highlights identified by the study tool
highlights perceived usefulness
Analysis
1. Comparison of the text manually identified and the text identified by our tool
2. Comparison of the solutions provided for a task performed with or without tool support
3. Qualitative analysis of the participants' provided feedback.

Figure 6.1: Summary of experimental procedures

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

6.2.1 Tasks

We opted for an experiment with tasks that could be completed by participants on their own time and computer. This decision was motived by the COVID-19 pandemic and challenges related to recruiting participants and conducting an in-person experiment [138, 139]. Since participants would follow instructions on their own, we decided to use tasks that are easy to understand and perform in a single experimental session, but that still required a participant to seek information in artifacts associated with each task.

Table 6.1 details the tasks that we have selected based on task selection procedures from related work that meet these criteria [152]. These tasks were drawn from Python w3resource[1] tasks that require usage of at least one module external to the Python core library. By using external modules, we aim to reduce the likelihood that a participant can provide a solution for a task without consulting any of the artifacts (Section 6.2.2) that detail each of the modules associated with each task.

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

6.2.2 Artifacts

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

Therefore, we follow procedures similar to the ones we used to create the DS$_{android}$ dataset to produce the list of artifacts for each of the tasks in Table 6.1. That is, we use the Google search engine to obtain up to ten artifacts that likely contain information that could help a participant correctly complete that task. Three

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

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

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

Table 6.1: Python tasks

pilot runs ensured that the artifacts collected using such procedures had sufficient information to complete a task without the need of additional resources. Based on these pilots, we simplified the description of the `distances` tasks removing the need to sort the list of suggested addresses, what also led to the removal of two artifacts associated with sorting. Table 6.3 details the final list of artifact types per task.

6.2.3 Coding environment

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

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

Figure 6.3 shows an example of the Colab coding environment. First it handled dependencies management and then, it presented a class containing a single method with a `TODO` block where participants should write their solution. The environment

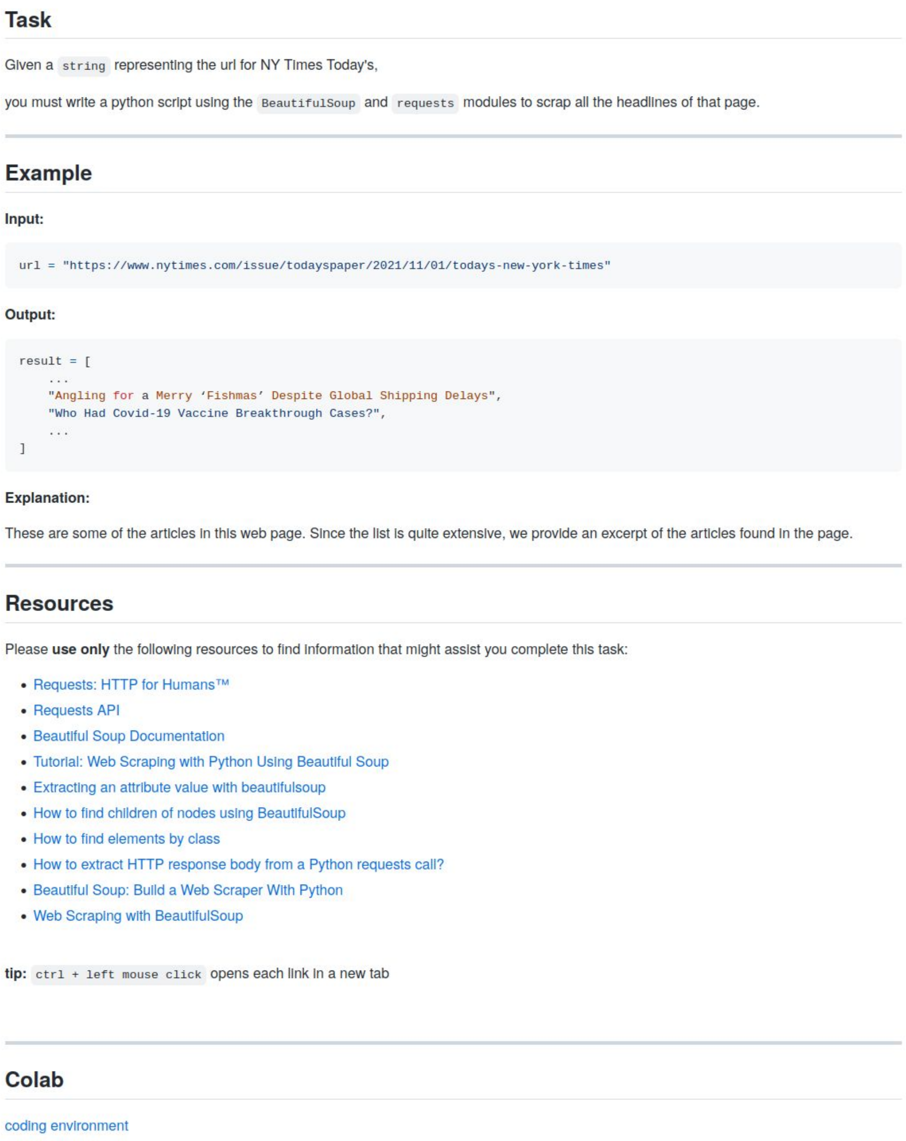
[3] https://colab.research.google.com/

also provided a main function where participants could see the output of their code. Alternatively, a participant could use test cases to test their solution against the examples shown in each task description.

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

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

Table 6.2: List of artifact types per task



Task

Given a `string` representing the url for NY Times Today's,

you must write a python script using the `BeautifulSoup` and `requests` modules to scrap all the headlines of that page.

Example

Input:

```
url = "https://www.nytimes.com/issue/todayspaper/2021/11/01/todays-new-york-times"
```

Output:

```
result = [
    ...
    "Angling for a Merry ‘Fishmas’ Despite Global Shipping Delays",
    "Who Had Covid-19 Vaccine Breakthrough Cases?",
    ...
]
```

Explanation:

These are some of the articles in this web page. Since the list is quite extensive, we provide an excerpt of the articles found in the page.

Resources

Please **use only** the following resources to find information that might assist you complete this task:

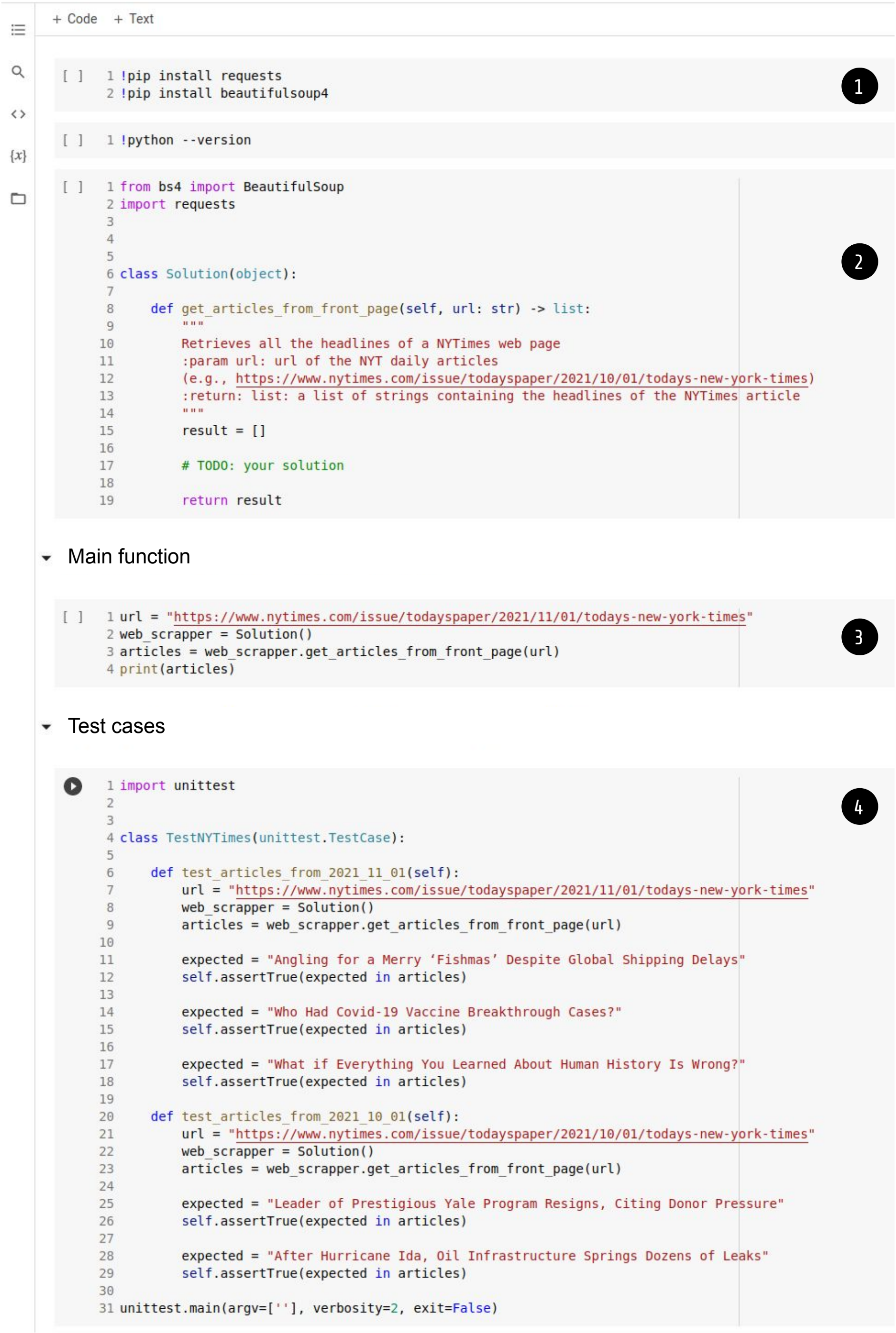
- Requests: HTTP for Humans™
- Requests API
- Beautiful Soup Documentation
- Tutorial: Web Scraping with Python Using Beautiful Soup
- Extracting an attribute value with beautifulsoup
- How to find children of nodes using BeautifulSoup
- How to find elements by class
- How to extract HTTP response body from a Python requests call?
- Beautiful Soup: Build a Web Scraper With Python
- Web Scraping with BeautifulSoup

tip: `ctrl + left mouse click` opens each link in a new tab

Colab

coding environment

Figure 6.2: Information shown in a task



```
!pip install requests
!pip install beautifulsoup4
```

```
!python --version
```

```
from bs4 import BeautifulSoup
import requests



class Solution(object):

    def get_articles_from_front_page(self, url: str) -> list:
        """
        Retrieves all the headlines of a NYTimes web page
        :param url: url of the NYT daily articles
        (e.g., https://www.nytimes.com/issue/todayspaper/2021/10/01/todays-new-york-times)
        :return: list: a list of strings containing the headlines of the NYTimes article
        """
        result = []

        # TODO: your solution

        return result
```

Main function

```
url = "https://www.nytimes.com/issue/todayspaper/2021/11/01/todays-new-york-times"
web_scrapper = Solution()
articles = web_scrapper.get_articles_from_front_page(url)
print(articles)
```

Test cases

```
import unittest


class TestNYTimes(unittest.TestCase):

    def test_articles_from_2021_11_01(self):
        url = "https://www.nytimes.com/issue/todayspaper/2021/11/01/todays-new-york-times"
        web_scrapper = Solution()
        articles = web_scrapper.get_articles_from_front_page(url)

        expected = "Angling for a Merry ‘Fishmas’ Despite Global Shipping Delays"
        self.assertTrue(expected in articles)

        expected = "Who Had Covid-19 Vaccine Breakthrough Cases?"
        self.assertTrue(expected in articles)

        expected = "What if Everything You Learned About Human History Is Wrong?"
        self.assertTrue(expected in articles)

    def test_articles_from_2021_10_01(self):
        url = "https://www.nytimes.com/issue/todayspaper/2021/10/01/todays-new-york-times"
        web_scrapper = Solution()
        articles = web_scrapper.get_articles_from_front_page(url)

        expected = "Leader of Prestigious Yale Program Resigns, Citing Donor Pressure"
        self.assertTrue(expected in articles)

        expected = "After Hurricane Ida, Oil Infrastructure Springs Dozens of Leaks"
        self.assertTrue(expected in articles)

unittest.main(argv=[''], verbosity=2, exit=False)
```

Figure 6.3: Colab environment

6.2.4 Participants

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

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

On average, participants self-reported 8 years of programming experience (± 3.8, ranging from 3 to 17 years). The majority of the participants (54%) had between 5 to 10 years of experience in Object-Oriented programming languages, closely followed by participants with 3 to 4 years of experience (29%). Most of the participants also indicated that they did check API documents almost every time they performed a programming task.

To witch gender do you identify?

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

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

For how many years have you been developing software?

For how many years have you been developing software professionaly?

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

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

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

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

[a] closed or open intervals notation

Figure 6.4: Background questions asked to a participant

6.2.5 Procedures

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

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

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

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

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

Table 6.3: List of participants who performed each task

Control Task

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

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

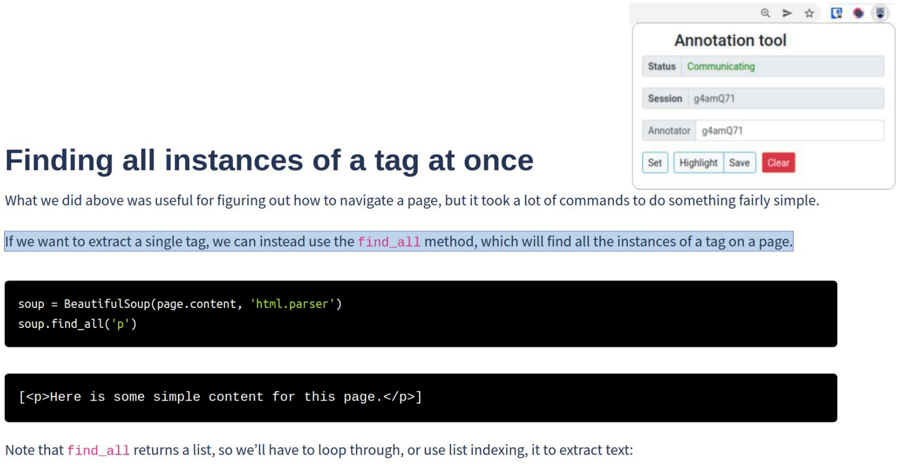
Participants could highlight text any time before submitting their solution. Based on observations from the pilots, the most common strategy we noticed was that of highlighting text on-the-fly, i.e., as a participant performed the task and consulted its artifacts, they would highlight text while reading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

Tool-assisted Task

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

For this task, when a participant submitted their solution, we asked them to rate on a 5 points Likert scale [93] how helpful were the highlights shown by the tool. Figure 6.6 shows an example of how we gathered data about the usefulness of the text automatically identified. Participants rated highlights on a per artifact basis. We gather input at the artifact level because it would be too demanding for

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



Status: Communicating

Session: g4amQ71

Annotator: g4amQ71

Finding all instances of a tag at once

What we did above was useful for figuring out how to navigate a page, but it took a lot of commands to do something fairly simple.

If we want to extract a single tag, we can instead use the `find_all` method, which will find all the instances of a tag on a page.

```
soup = BeautifulSoup(page.content, 'html.parser')
soup.find_all('p')
```

```
[<p>Here is some simple content for this page.</p>]
```

Note that `find_all` returns a list, so we'll have to loop through, or use list indexing, it to extract text:

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

a participant to provide individual feedback on each of the highlights shown in the time that we estimated and that we advertised for the study. Section 6.3.5 discusses threats that arise from this decision.

1. Indicate whether you agree with the following statement:

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

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

2. Indicate whether you agree with the following statement:

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

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

...

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

6.2.6 Summary of experimental procedures

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

1. a participant's submitted solution (written Python code) for each task;

2. text that participants deemed relevant in the artifacts of the control task;

3. the usefulness of the highlights shown in a tool-assisted task; and

4. any additional feedback (written text) that a participant wished to provide.

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

6.3 Results

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

6.3.1 Tasks Correctness

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

Metrics

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

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

Data

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

Results

Figure 6.7 aggregates all correctness scores for tasks done with and without tool support. We compare correctness scores first using a Shapiro-Wilk test [159] to check for normality and then, due to deviation from a normal distribution (*p-value* < 0.05), applying a Wilcoxon-Mann-Whitney test [102]. Results from this test indicate that we cannot draw statistically significant conclusions for the solution scores of the two groups (*p-value* ≥ 0.10)[5]. Hence, we evaluate scores on a per-task basis, as shown in Figure 6.8.

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

The two groups have the same correctness scores in the `NYTimes` tasks. We found that the most notable differences in the solutions of this task arise from participants who did (or not) consider corner-case scenarios. To illustrate this, we quote one participant who stated that their solution could "*fetch all the articles, but*

[5] As Section 6.3.5 details, the lack of statistical power is not surprising and is a common risk to between groups comparisons [88].

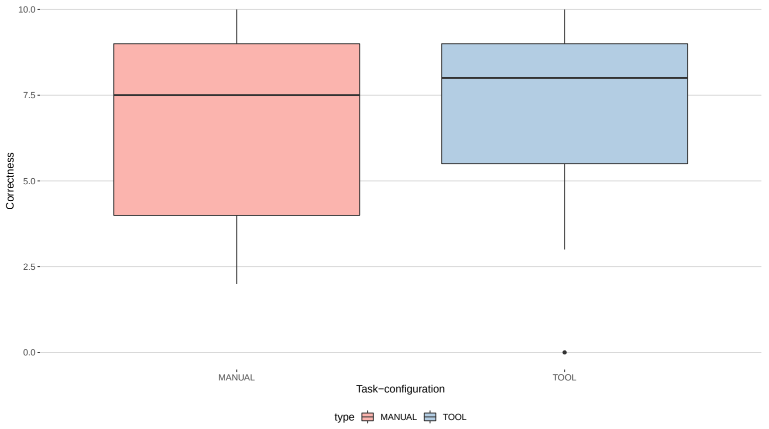


Figure 6.7: Boxplots showing aggregated correctness scores for task performed with and without tool support

it could also get some noise", e.g., getting all the headlines of the web page but also an unrelated ad, which is indeed one of the test cases for this task.

The `titanic` task required using two data science modules and it is the one with the most differences. Participants who did the task with tool support were able to produce more correct solutions, with an average correctness score of 6. In contrast, participants in the control group had an average correctness of 4. As we detail in the next sections, we found that the tool identified much of the text that the participants deemed relevant and that participants also indicated that the text automatically identified was indeed useful.

6.3.2 Usefulness Analysis

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

Metrics

To investigate the usefulness of the highlights shown by our tool, we asked participants to indicate on a 5 point Likert scale whether the highlights of each artifact

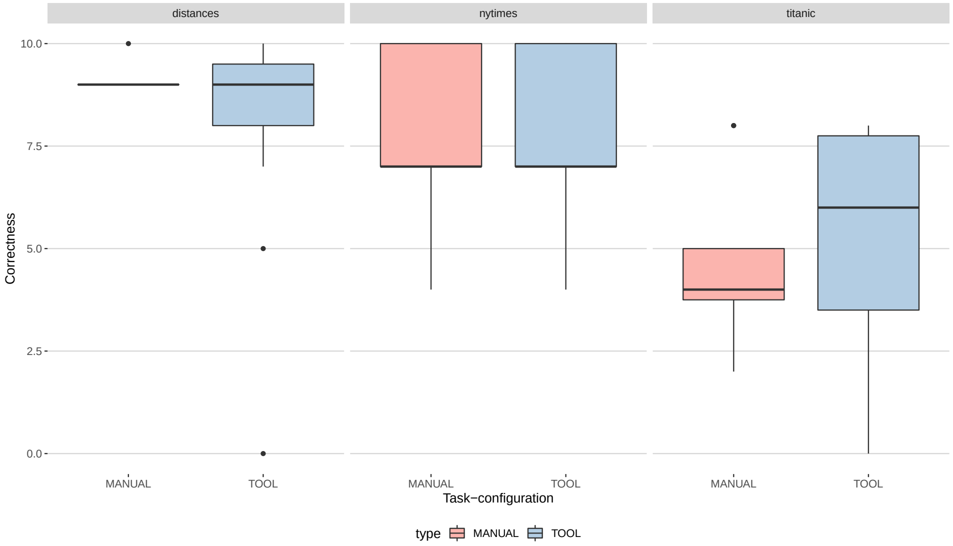


Figure 6.8: Boxplots showing correctness scores for each task performed with and without tool support

were helpful to correctly accomplish their assigned task (Figure 6.6). We aggregate individual responses to measure how useful the tool was in assisting developers complete each task in our experiment, plotting responses using a diverging stacked bar chart [148].

Data

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

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

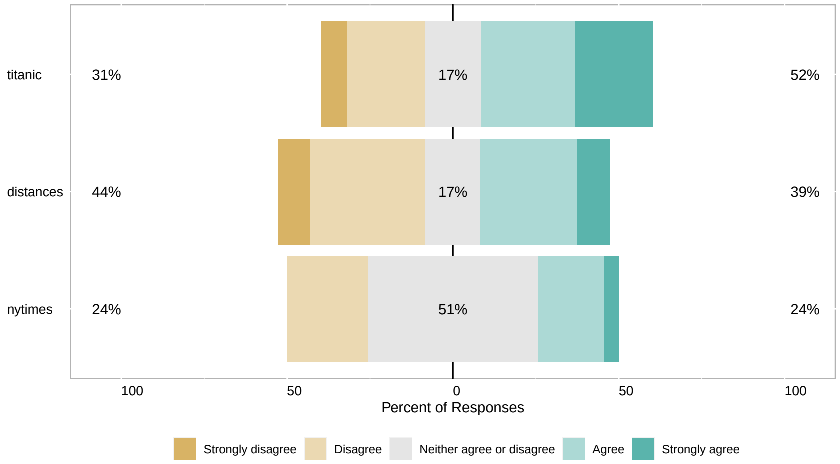


Figure 6.9: Diverging stacked plot of the usefulness of the text automatically identified for each task

Results

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

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

The mixed results for the `NYTimes` task might explain the lack of differences in the correctness scores observed for this task. Although anecdotal, one interesting feedback for this task was from a participant who described that their experience

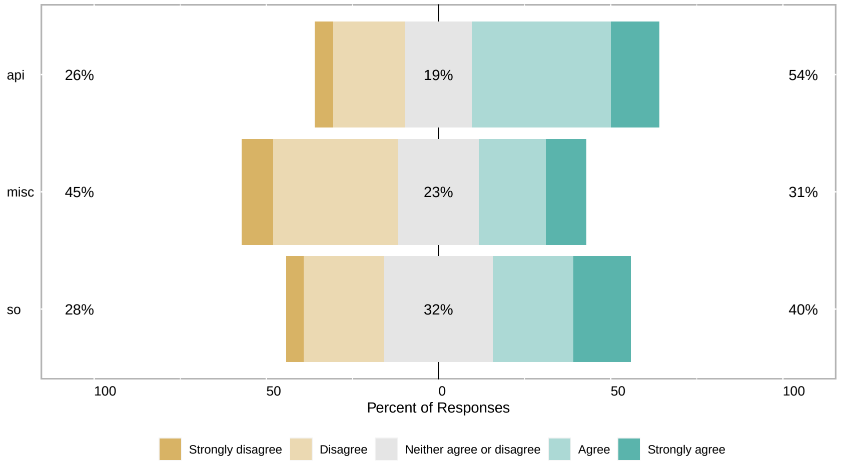


Figure 6.10: Diverging stacked plot of the usefulness of the text automatically identified for each type of artifact

with the `BeautifulSoup` module influenced their negative ratings—"*I have experience with BeautifulSoup and webscraping, and so my [negative] ratings of the usefulness of the highlights may have been influenced by that*".

Surprisingly, when we look at the ratings per type of artifact (Figure 6.10), API documents had the most useful highlights. For this type of artifact, we observe that positive ratings originate from artifacts that follow a *'how to'* format, which is not conventional for API documents [19, 136]. The highlights on Stack Overflow artifacts were also perceived as useful and participants expressed familiarity with this type of artifact, where the highlights helped them determine parts of the page to ignore—"*the highlights [on Stack Overflow] helped me quickly determine which parts of the page to ignore*".

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

6.3.3 Comparison of manual and automatically identified task-relevant text

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

Metrics

To investigate the overlap between the participants' manual highlights and the the automatic highlights identified by our semantic-based tool we use *precision*, *recall* [103], and *pyramid precision* [121]. We compute these metrics for each artifact of each task and report their average.

For this analysis, we follow Lotufo et al.'s procedures [100] and we consider any text marked by any participant as relevant. We investigate if our tool automatically identifies text that multiple participants deemed relevant via *pyramid precision*. Details for each metric are as follows.

Precision measures the fraction of the text automatically identified that participants deemed relevant (Equation 6.2).

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

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

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

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

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

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

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

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

Data

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

Some participants also selected code snippets as relevant to a task—a threat that we discuss in Section 6.3.5. Code snippets account for 30% of the highlights produced, but we remove them from our analysis since our semantic-based approach operates on text only. For the textual highlights, Krippendorf's alpha indicates good agreement of what text in an artifact participants deemed relevant ($\alpha = 0.68$) [82, 124]. We compare these manually produced highlights to the text automatically identified by our tool.

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

Results

Table 6.5 summarizes the average of precision, pyramid precision, and recall metrics for each of the tasks in the experiment. Precision scores range from 0.55 to 0.68, while pyramid precision scores range from 0.55 to 0.57, which suggests that our tool failed to identify some of the text that participants deemed the most relevant.

The results in Table 6.5 corroborate the correctness scores detailed in Figure 6.8. For example, in the `distances` task, participants who performed the task with tool support had solutions less correct than participants in the control group. This was the task with the lowest precision, recall and pyramid precision values. In contrast, the task where participants assisted by our tool obtained the best correctness scores, `titanic`, is the one with the best precision, recall and pyramid precision values.

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

We also compare evaluation results to the ones in Chapter 5. This comparison is interesting because it let us cross-examine our findings with new data [53], which Seaman emphasizes as an important method for empirical studies in software engineering [144]. Figure 6.11 presents boxplot for precision and recall of the BERT technique with no filters for each evaluation. We observe that the technique accuracy is comparable across the artifacts on each evaluation.

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

Table 6.5: Evaluation metrics per task

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

Table 6.6: Evaluation metrics per type of artifact

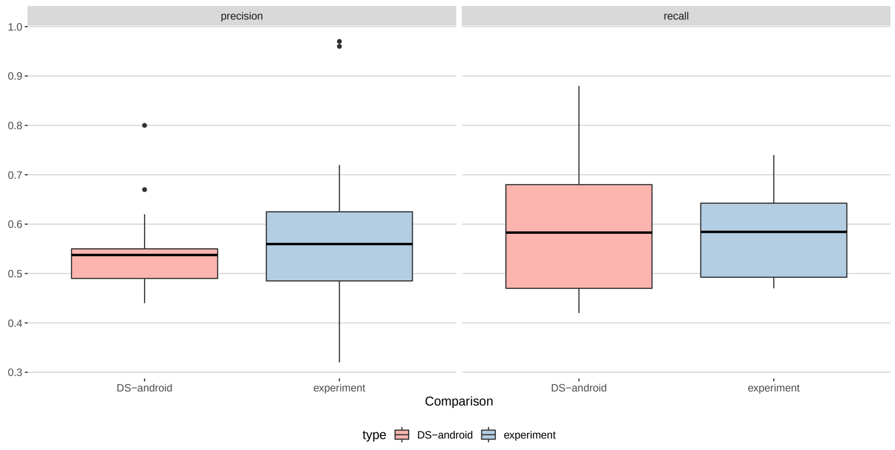


Figure 6.11: Boxplots comparing precision and recall values when applying our semantic-based technique to the DS_{python} and the $\text{DS}_{android}$ corpora

6.3.4 Summary of results

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

6.3.5 Threats to Validity

Our experiment compares solutions submitted by participants who attempted each task with and without tool support. This represents a between groups design [88] and we discuss threats inherent to it.

Since we compare results from different participants, our analysis might be subject to substantial impact from individual differences [88]. For example, participants who performed a task with tool support may have been more experienced than participants who did the same task without tool support what affects correctness scores. As another example, participants' skill and background influences the text that they indicate as relevant in the control task as well what text they perceive as useful in the tool-assisted task. We minimized these threats by recruiting participants of varied background and randomly assigning tasks to each participant.

The tasks in our experiment impact generalizability. Although we opted for simple tasks, we ensured they modules used in our tasks were representative. For example, we found open-source systems[6] using `BeautifulSoup` with function calls similar to the ones needed to complete the `NYTimes` task. Nonetheless, there are clear differences in the artifacts one can gather based on the domain or programming language of a task [23]. Hence, we consider other domains and a wider range of task and artifacts for future work.

The selection of tasks also affects our conclusions. We opted for Python programming tasks that required writing code, which we use to assess correctness. As observed by other researchers [111, 143], developers work on many different tasks, some of which focus on code [110] while others on information seeking [65],

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

e.g., finding duplicated bug reports or researching visualization libraries to identify the most suitable one [143]. Had we decided to use information-seeking tasks, participants could have produced a different set of highlights, perhaps selecting fewer code snippets. Given that our experiment was completely remote, instructing participants on how to perform information-seeking tasks would have been more difficult. Furthermore, objectively judging their correctness would also be more strenuous, which would lead to a different experiment with challenges and risks of its own.

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

Concerning the text automatically identified by our tool, we gather usefulness at the artifact level. Suppose we had gathered usefulness at the sentence level. In that case, we could have used this information to further refine our analysis, for example, reporting precision and recall at different usefulness levels or computing accuracy based on the participants' input, as done by Xu et al [163]. However, asking participants to provide feedback at the sentence level would have considerably increased the time we estimated that the experiment would take, which would impact recruitment. We weighed the benefits and drawbacks of a fine-grained or more coarse-grained analysis, and we opted for the latter so that this would not be a barrier to people deciding on whether to participate in our experiment.

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

6.4 Summary

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

Bibliography

[1] Stanford CoreNLP. https://stanfordnlp.github.io/CoreNLP/. Verified: 2021-06-09. → page 26

[2] PyGithub. https://pypi.org/project/PyGithub/. Verified: 2021-06-09. → page 26

[3] StackAPI. https://stackapi.readthedocs.io/en/latest/. Verified: 2021-06-09. → page 26

[4] Alexa. https://alexa.com. Verified: 2021-04-08. → page 25

[5] Lock task mode. https://developer.android.com/work/dpc/dedicated-devices/lock-task-mode, . Verified: 2021-04-08. → page 22

[6] Managing WebView objects. https://developer.android.com/guide/webapps/managing-webview, . Verified: 2021-04-08. → page 23

[7] apriori-python. https://pypi.org/project/apriori-python/. Verified: 2021-10-11. → page 40

[8] beautifulsoup4. https://pypi.org/project/beautifulsoup4/. Verified: 2021-06-09. → page 26

[9] No lock screen controls ever #3578. https://github.com/AntennaPod/AntennaPod/issues/3578. Verified: 2021-04-08. → page 26

[10] Colaboratory. https://research.google.com/colaboratory/faq.html, . Verified: 2022-03-24. → page 55

[11] googlesearch. https://pypi.org/project/googlesearch-python/, . Verified: 2020-10-27. → page 25

[12] Create A React Native App Which works on Lock Screen (Android). https://bit.ly/3dNF8I5. Verified: 2021-04-08. → page 22

[13] *Evaluating the Use of Semantics for Identifying Task-relevant Textual Information Supplementary Material*, Oct. 2021. Zenodo. doi:10.5281/zenodo.5591010. URL https://doi.org/10.5281/zenodo.5591010. → pages 21, 40

[14] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994. ISBN 1558601538. → page 40

[15] F. N. A. Al Omran and C. Treude. Choosing an NLP library for analyzing software documentation: A systematic literature review and a series of experiments. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 187–197, 2017. doi:10.1109/MSR.2017.42. → page 26

[16] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015. → page 17

[17] 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 16

[18] 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 13, 22, 26

[19] D. M. Arya, J. L. Guo, and M. P. Robillard. Information correspondence between types of documentation for apis. *Empirical Software Engineering*, 25(5):4069–4096, 2020. → pages 29, 30, 68

[20] S. Baltes and S. Diehl. Usage and attribution of stack overflow code snippets in github projects. *Empirical Software Engineering*, 24(3): 1259–1295, 2019. → pages 22, 49

[21] S. Baltes, C. Treude, and S. Diehl. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *2019 IEEE/ACM*

16th International Conference on Mining Software Repositories (MSR), pages 191–194, 2019. → page 23

[22] S. Baltes, C. Treude, and M. P. Robillard. Contextual Documentation Referencing on Stack Overflow Supplementary Material. Zenodo, Mar. 2019. doi:10.5281/zenodo.2585828. URL https://doi.org/10.5281/zenodo.2585828. → page 23

[23] S. Baltes, C. Treude, and M. P. Robillard. Contextual documentation referencing on stack overflow. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi:10.1109/TSE.2020.2981898. → pages 23, 73

[24] 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 18

[25] C. L. Barry. User-defined relevance criteria: An exploratory study. *Journal of the American Society for Information Science*, 45(3):149–159, 1994. → page 32

[26] C. L. Barry. Document representations and clues to document relevance. *Journal of the American Society for Information Science*, 49(14): 1293–1303, 1998. ISSN 1097-4571. → page 32

[27] 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, 7, 12, 14, 15, 37

[28] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015. doi:10.1109/TSE.2014.2367027. → page 21

[29] 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 3

[30] E. Ben Charrada and S. Mussato. An exploratory study on managing and searching for documents in software engineering environments. In *Proc of*

the 2016 ACM Symp. on Document Engineering, pages 189–192, 2016. ISBN 978-1-4503-4438-8. → page 7

[31] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 121–130, 2009. → page 7

[32] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, Dec. 2017. doi:10.1162/tacl_a_00051. URL https://www.aclweb.org/anthology/Q17-1010. → page 16

[33] H. Borges and M. T. Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018. → page 22

[34] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016. → page 22

[35] 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. → page 3

[36] 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 25

[37] K. Byström and K. Järvelin. Task complexity affects information seeking and use. *Information Processing and Management*, 31(2):191–213, 1995. ISSN 03064573. → page 4

[38] 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 13

[39] 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 7, 14

[40] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 86–96, 2019. ISBN 9781450355728. doi:10.1145/3338906.3338947. → pages 8, 17, 35

[41] C. Chen, S. Gao, and Z. Xing. Mining Analogical Libraries in Q&A Discussions – Incorporating Relational and Categorical Knowledge into Word Embedding. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 338–348, 2016. doi:10.1109/SANER.2016.21. → page 15

[42] C. Chen, Z. Xing, Y. Liu, and K. O. L. Xiong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 47(3):432–447, 2019. → page 17

[43] E. H. Chi, M. Gumbrecht, and L. Hong. Visual foraging of highlighted text: An eye-tracking study. In *International Conference on Human-Computer Interaction*, pages 589–598. Springer, 2007. → page 7

[44] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. → page 17

[45] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 2126–2136, July 2018. doi:10.18653/v1/P18-1198. → page 37

[46] 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 12, 18

[47] K. Cwalina and B. Abrams. *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Pearson Education, 2008. → page 52

[48] D. Das, D. Chen, A. F. Martins, N. Schneider, and N. A. Smith. Frame-semantic parsing. *Computational linguistics*, 40(1):9–56, 2014. → page 19

[49] 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. → page 7

[50] L. Deng and Y. Liu. *Deep Learning in Natural Language Processing*. Springer Publishing Company, Incorporated, 1st edition, 2018. ISBN 9789811052088. → page 17

[51] 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 8, 18, 36, 38

[52] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall. Development emails content analyzer: Intention mining in developer discussions (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 12–23, 2015. doi:10.1109/ASE.2015.12. → pages 12, 13, 17

[53] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008. → page 71

[54] V. Efstathiou, C. Chatzilenas, and D. Spinellis. Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 38–41, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi:10.1145/3196398.3196448. → page 37

[55] 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, 3

[56] G. Erkan and D. R. Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *J. Artif. Int. Res.*, 22(1):457–479, Dec. 2004. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622487.1622501. → page 16

[57] 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 16, 17

[58] I. Ferreira, E. Cirilo, V. Vieira, and F. Mouro. Bug report summarization: An evaluation of ranking techniques. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 101–110, Sep. 2016. doi:10.1109/SBCARS.2016.17. → page 22

[59] C. J. Fillmore. Frame semantics and the nature of language. *Annals of the New York Academy of Sciences*, 280(1):20–32, 1976. → pages 8, 36

[60] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proc. of the 2002 ACM Symposium on Document Engineering*, pages 26–33, 2002. ISBN 1-58113-594-7. → page 7

[61] V. d. C. L. Freire et al. *Characterization of design discussions in modern code review.* PhD thesis, 2021. → page 18

[62] 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 45

[63] 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 14, 16, 17

[64] J. Goldstein et.al. Summarizing Text Document: Sentence Selection and Evaluation Metrics. *In Proceeding of SIGIR'99*, pages 121–128, 1999. → page 15

[65] M. K. Gonçalves, C. R. de Souza, and V. M. González. Collaboration, information seeking and communication: An observational study of software developers' work practices. *J. Univers. Comput. Sci.*, 17(14): 1913–1930, 2011. → pages 4, 73

[66] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14. IEEE, 2017. → page 17

[67] 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 25

[68] Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954. → pages 17, 37

[69] 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. → pages 12, 18

[70] Q. Huang, X. Xia, D. Lo, and G. C. Murphy. Automating intention mining. *IEEE Transactions on Software Engineering*, 2018. → page 13

[71] 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. → pages 15, 16, 17

[72] B. Hutchinson, A. Smart, A. Hanna, E. Denton, C. Greer, O. Kjartansson, P. Barnes, and M. Mitchell. Towards accountability for machine learning datasets: Practices from software engineering and infrastructure. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 560–575, 2021. → page 7

[73] 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 16, 29

[74] 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 16, 30

[75] J. Josyula, S. Panamgipalli, M. Usman, R. Britto, and N. B. Ali. Software practitioners' information needs and sources: A survey study. In *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 1–6. IEEE, 2018. → page 25

[76] D. Jurafsky and J. H. Martin. *Speech and language processing*, volume 3. Pearson London, 2014. → page 19

[77] 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. → page 3

[78] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101, 2014. ISBN 9781450328630. doi:10.1145/2597073.2597074. → page 22

[79] 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 18

[80] 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 25

[81] W. Kintsch and T. A. van Dijk. Toward a model of text comprehension and production. *Psychological Review*, 85(5):363–394, 1978. ISSN 0033295X. → page 8

[82] K. Klaus. Content analysis: An introduction to its methodology, 1980. → page 70

[83] 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 4, 7, 45

[84] 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 2

[85] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018. → page 30

[86] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012. → page 17

[87] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006. → page 25

[88] J. Lazar, J. H. Feng, and H. Hochheiser. Chapter 3 - experimental design. In J. Lazar, J. H. Feng, and H. Hochheiser, editors, *Research Methods in Human Computer Interaction (Second Edition)*, pages 45–69. Morgan Kaufmann, Boston, second edition edition, 2017. ISBN 978-0-12-805390-4. doi:https://doi.org/10.1016/B978-0-12-805390-4.00003-0. → pages 64, 73

[89] 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 2, 3, 7, 8, 24, 25

[90] L. Li, J. Gao, T. Bissyand, L. Ma, X. Xia, and J. Klein. Characterising deprecated android apis. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 254–264, 2018. → page 21

[91] 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. → pages 15, 16, 17

[92] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang. Deep learning in software engineering. *arXiv preprint arXiv:1805.04825*, 2018. → page 17

[93] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932. → page 61

[94] 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 7

[95] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto. Sentiment analysis for software engineering: How far can we go? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 94–104, May 2018. doi:10.1145/3180155.3180195. → page 17

[96] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 83–94, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi:10.1145/2597008.2597155. → page 21

[97] M. X. Liu, N. Hahn, A. Zhou, S. Burley, E. Deng, J. Hsieh, B. A. Myers, and A. Kittur. UNAKITE: Support developers for capturing and persisting design rationales when solving problems using web resources. 2018. → pages 8, 18

[98] 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 12, 18

[99] A. Lopez. Statistical machine translation. *ACM Computing Surveys (CSUR)*, 40(3):1–49, 2008. → page 17

[100] 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 15, 16, 27, 42, 69

[101] 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 12, 13, 14, 16

[102] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50 – 60, 1947. doi:10.1214/aoms/1177730491. → pages 32, 44, 64

[103] C. Manning, P. Raghavan, and H. Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010. → pages 41, 69

[104] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2009. → pages 37, 39

[105] A. Marques, N. C. Bradley, and G. C. Murphy. Characterizing task-relevant information in natural language software artifacts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 476–487, 2020. doi:10.1109/ICSME46990.2020.00052. → pages 26, 27, 39, 40, 45

[106] A. Marques, G. Viviani, and G. C. Murphy. Assessing semantic frames to support program comprehension activities. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 13–24, 2021. doi:10.1109/ICPC52881.2021.00011. → pages 8, 39

[107] B. G. Mateus and M. Martinez. On the adoption, usage and evolution of kotlin features in android development. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020. ISBN 9781450375801. doi:10.1145/3382494.3410676. → page 21

[108] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79, 2013. doi:10.1109/ICSM.2013.18. → page 21

[109] A. Meyer, E. T. Barr, C. Bird, and T. Zimmermann. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering*, 2019. → page 8

[110] 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, 3, 12, 18, 23, 73

[111] A. N. Meyer, C. Satterfield, M. Züger, K. Kevic, G. C. Murphy, T. Zimmermann, and T. Fritz. Detecting developers' task switches and types. *IEEE Transactions on Software Engineering*, 2020. → page 73

[112] Q. Mi, J. Keung, Y. Xiao, S. Mensah, and Y. Gao. Improving code readability classification using convolutional neural networks. *Information and Software Technology*, 104:60–71, 2018. → page 17

[113] R. Mihalcea, C. Corley, C. Strapparava, et al. Corpus-based and knowledge-based measures of text semantic similarity. In *Aaai*, volume 6, pages 775–780, 2006. → page 17

[114] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. 2013. → page 17

[115] 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 8, 16, 17, 36, 37

[116] L. Moreno and A. Marcus. Automatic software summarization: The state of the art. *Proc. - 2017 IEEE/ACM 39th Int'l Conf. on SE Companion*, pages 511–512, 2017. → page 15

[117] 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, 7, 35, 41

[118] G. Murray and G. Carenini. Summarizing spoken and written conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 773–782, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1613715.1613813. → page 15

[119] S. Nadi and C. Treude. Essential sentences for navigating stack overflow answers, Dec 2019. URL https://figshare.com/articles/software/Essential_Sentences_for_Navigating_Stack_Overflow_Answers/10005515/3. → page 7

[120] 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 4, 14, 22, 26, 27, 29

[121] 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 30, 32, 69

[122] 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. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120. → page 16

[123] 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, 21

[124] R. Passonneau. Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy, May 2006. European Language Resources Association (ELRA). → page 70

[125] 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. → page 17

[126] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath. To fix or to learn? How production bias affects developers' information foraging during debugging. *2015 IEEE 31st Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 11–20, 2015. → page 7

[127] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett. Foraging and navigations, fundamentally: Developers' predictions of value and cost. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 97–108, 2016. ISBN 9781450342186. doi:10.1145/2950290.2950302. → pages 4, 7, 26

[128] P. L. T. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, Inc., New York, NY, USA, 1 edition, 2007. ISBN 0195173325, 9780195173321. → page 4

[129] 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. → pages 15, 16

[130] 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 8, 16, 17

[131] N. Rao, C. Bansal, T. Zimmermann, A. H. Awadallah, and N. Nagappan. Analyzing web search behavior for software engineering tasks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 768–777, 2020. doi:10.1109/BigData50022.2020.9378083. → pages 2, 23

[132] S. Rastkar. *Summarizing software artifacts*. PhD thesis, University of British Columbia, 2013. → page 12

[133] S. Rastkar and G. C. Murphy. Why did this code change? In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 1193–1196, 2013. ISBN 978-1-4673-3076-3. → page 13

[134] 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 7, 15, 16, 27, 29

[135] 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 3, 4, 7, 13, 14, 26, 27, 35, 52

[136] M. P. Robillard and R. Deline. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011. → pages 1, 3, 24, 29, 68

[137] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, et al. On-demand developer documentation. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 479–483. IEEE, 2017. → page 12

[138] D. Russo, P. H. Hanel, S. Altnickel, and N. Van Berkel. The daily life of software engineers during the covid-19 pandemic. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 364–373. IEEE, 2021. → page 54

[139] D. Russo, P. H. Hanel, S. Altnickel, and N. van Berkel. Predictors of well-being and productivity among software professionals during the covid-19 pandemic–a longitudinal study. *Empirical Software Engineering*, 26(4):1–63, 2021. → page 54

[140] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, Nov. 1975. ISSN 0001-0782. doi:10.1145/361219.361220. URL https://doi.org/10.1145/361219.361220. → page 16

[141] T. Saracevic. Relevance: A review of the literature and a framework for thinking on the notion in information science. part iii: Behavior and effects of relevance. In *Journal of the American Society for Information Science and Technology*, volume 58, pages 2126–2144, 2007. doi:https://doi.org/10.1002/asi.20681. → page 49

[142] T. Saracevic. Relevance: A review of the literature and a framework for thinking on the notion in information science. part ii: nature and manifestations of relevance. In *Journal of the American Society for Information Science and Technology*, volume 58, pages 1915–1933, 2007. doi:https://doi.org/10.1002/asi.20682. → page 49

[143] C. Satterfield, T. Fritz, and G. C. Murphy. Identifying and describing information seeking tasks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 797–808, 2020. → pages 18, 73, 74

[144] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999. ISSN 0098-5589. doi:10.1109/32.799955. → page 71

[145] 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 2, 18

[146] 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, 4, 8, 12, 15, 16, 25, 35

[147] 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 2

[148] R. Spence. *Information visualization*, volume 1. Springer, 2001. → page 66

[149] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009. → page 19

[150] 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, 7, 25

[151] M. Storey, A. Zagalsky, F. F. Filho, L. Singer, and D. M. German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2):185–204, Feb 2017. doi:10.1109/TSE.2016.2584053. → page 7

[152] E. Thiselton and C. Treude. Enhancing python compiler error messages via stack overflow. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. doi:10.1109/ESEM.2019.8870155. → page 54

[153] C. Treude and M. P. Robillard. Augmenting API documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 392–403, 2016. doi:10.1145/2884781.2884800. → pages 12, 15

[154] 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, 12

[155] 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 38

[156] G. Viviani, M. Famelis, X. Xia, C. Janik-Jones, and G. C. Murphy. Locating latent design information in developer discussions: A study on pull requests. 2019. doi:10.1109/TSE.2019.2924006. → pages 18, 30

[157] S. Wang, N. Phan, Y. Wang, and Y. Zhao. Extracting api tips from developer question and answer websites. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 321–332. IEEE, 2019. → page 17

[158] 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 17

[159] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. → page 64

[160] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 137–146, 2020. ISBN 9781450375177. doi:10.1145/3379597.3387459. → page 22

[161] 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 2, 17, 23

[162] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin. Bug localization with semantic and structural features using convolutional neural network and cascade forest. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 101–111. ACM, 2018. → page 17

[163] 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 4, 7, 8, 9, 15, 17, 22, 25, 35, 42, 74

[164] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 127–137, 2016. doi:10.1109/ISSRE.2016.33. → pages 8, 35

[165] M. Yazdaninia, D. Lo, and A. Sami. Characterization and prediction of questions without accepted answers on stack overflow. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 59–70, 2021. doi:10.1109/ICPC52881.2021.00015. → page 22

[166] 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 17, 37

[167] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021. → page 17

[168] D. Zhang and J. J. Tsai. *Machine learning applications in software engineering*, volume 16. World Scientific, 2005. → page 16

[169] H. Zhang, S. Wang, T.-H. Chen, and A. E. Hassan. Reading answers on stack overflow: Not enough! *IEEE Transactions on Software Engineering*, 2019. → page 29

[170] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010. → page 29