Abstract

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

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

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

We evaluate the proposed techniques assessing the extent to which they identify text that developers deem relevant in different kinds of artifacts associated with Android development tasks. We then investigate how a tool that embeds the most promising semantic-based technique might assist developers while they perform a task. Results show that semantic-based techniques perform equivalently well across multiple artifact types and that a tool that automatically provides task-relevant text assists developers effectively complete a software development task.

Glossary

ERB Research Ethics Board

UBC University of British Columbia

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

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

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

Chapter 6

Evaluating an Automated Approach to Task-Relevant Text Identification

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

For that, we embed the approach into a web browser plug-in that automatically highlights text relevant to a particular software task in web pages inspected by a developer. We investigate benefits brought by using this tool, which we call TARTI, through a controlled experiment, thus addressing whether developers can effectively complete a software task when provided with task-relevant text automatically extracted from natural language artifacts. We report how 24 participants completed three Python programming tasks using (or not) TARTI where results show that participants considered the majority of the text automatically identified in the artifacts they perused useful and that the tool assisted them in producing a more correct solution for one of the experimental tasks.

We start by outlining the evaluation approach (Section 6.1). We then detail experimental procedures (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 TARTI—a tool that automatically identifies task-relevant text in pertinent documents—can affect a developer's work. Building on work presented earlier in this thesis, the tool uses a semantic-based technique. By identifying information that is useful to the developer's task, a developer's burden to find task-relevant information [17] can be lowered, allowing them to focus their time on other activities such as judging how the information found applies to their task.

To be helpful, TARTI must direct a developer's attention to text that assists them 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 TARTI is not *useful*—either because it is not relevant for the task at hand or because it is unsurprising, i.e., a developer finds the text identified by the tool as "common-knowledge" [4, 17]. 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—TARTI—help developers produce more correct solutions?

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

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

To answer these questions, we designed an experiment where 24 participants with software development 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*.

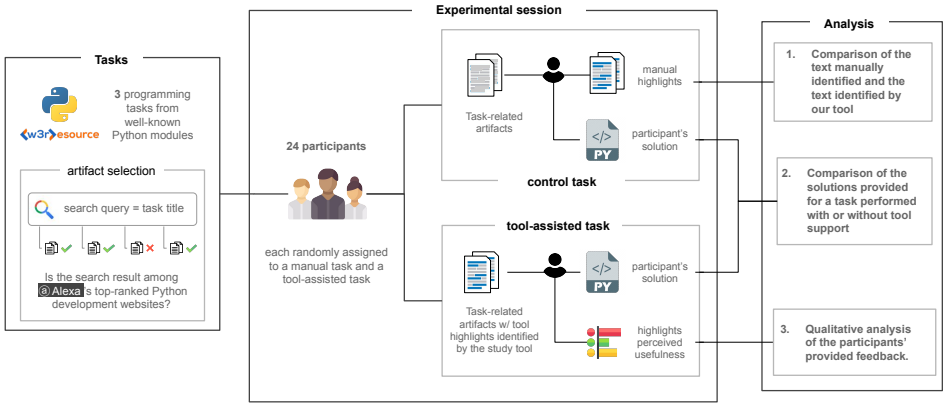


Figure 6.1: Summary of experimental procedures

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 [19, 20]. Since participants would follow instructions on their

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

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 [24]. 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

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

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

participants who perform a task assisted by our tool and without it as well as our need to compare the text that participants deem relevant to the text automatically identified by our tool means that all participants must have the exact same artifacts for a task.

Therefore, we follow procedures similar to the ones we used to create the $\text{DS}_{android}$ dataset to produce the list of artifacts for each of the tasks in Table 6.1. That is, 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 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.

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

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

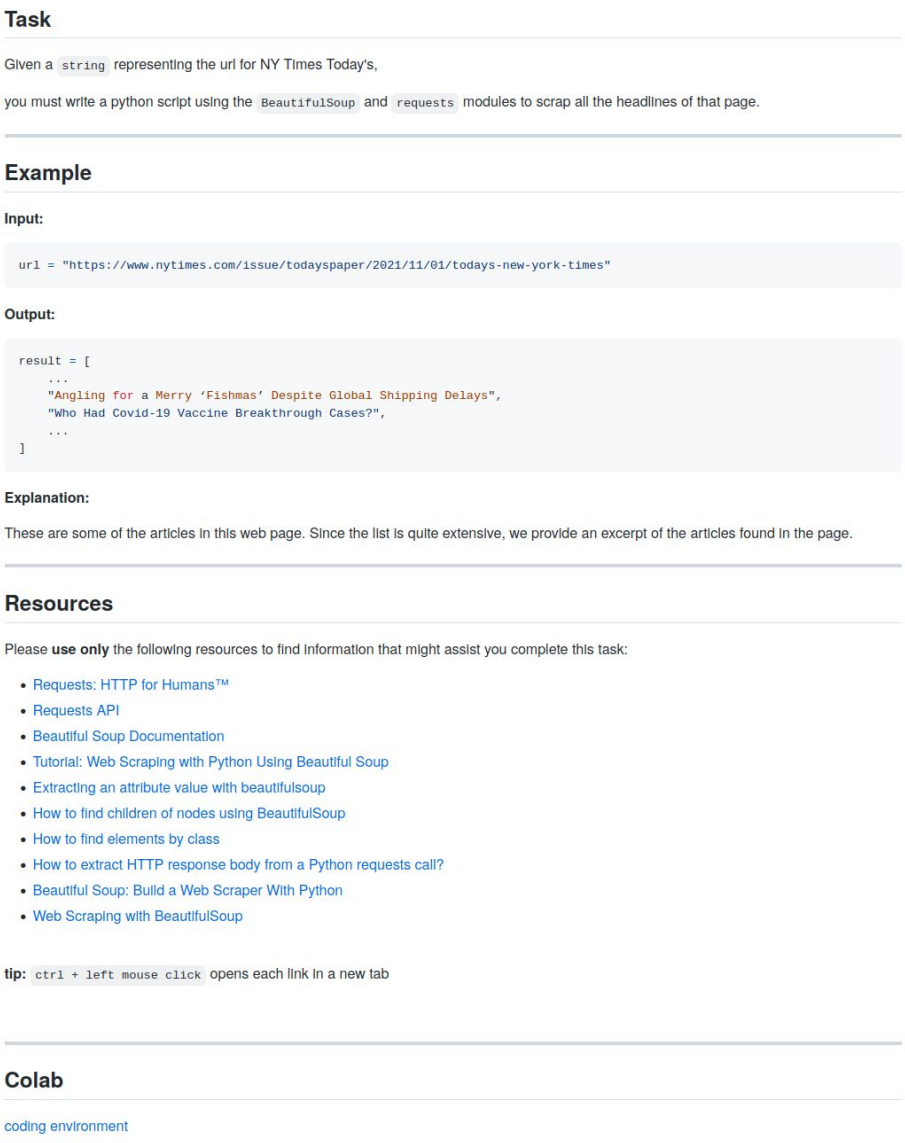
Table 6.2: List of artifact types per task

6.2.3 Coding environment

To ensure that participants had the same conditions to perform each task and also to minimize setup instructions, we used Google Colab[3] as our coding environment. Colab is a product from Goole Research that allows people to write and execute Python code through their browser [1].

Colab provided participants with a code editor with amenities commonly found

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



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

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: Colab environment

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

6.2.4 Participants

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

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

On average, participants self-reported 8 years of programming experience ($\pm$ 3.8, 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.

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.

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* ☐ 4*th* ☐ 5*th*+ year ☐ *graduate student*

For how many years have you been developing software?

For how many years have you been developing software professionaly?

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

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

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

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

[a] closed or open intervals notation

Figure 6.4: Background questions asked to a participant

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 **??**).

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

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

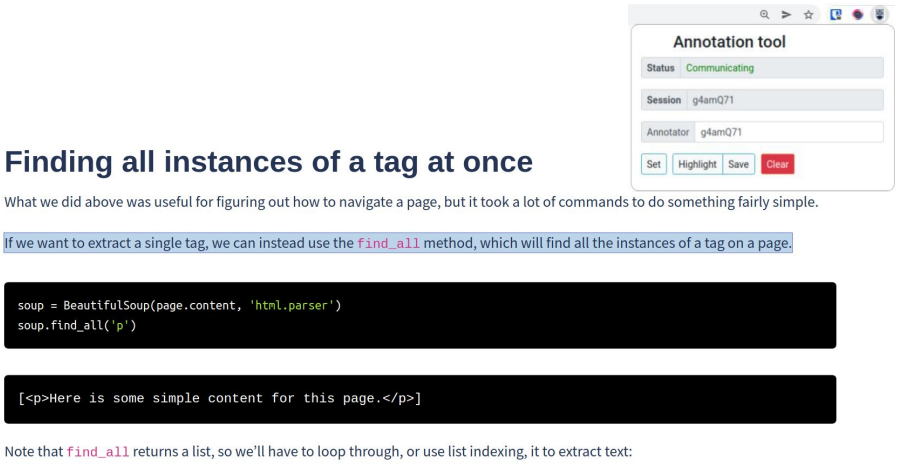


Figure 6.5: Study's tool context menu (top-right corner) and a sentence highlighted by a participant

Tool-assisted Task

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

For this task, when a participant submitted their solution, we asked them to rate on a 5 points Likert scale [9] how helpful were the highlights shown by the tool. Figure 6.6 shows an example of how we gathered data about the usefulness of the text automatically identified. Participants rated highlights on a per artifact basis. We gather input at the artifact level because it would be too demanding for a participant to provide individual feedback on each of the highlights shown in the time that we estimated and that we advertised for the study. Section 6.3.5 discusses threats that arise from this decision.

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

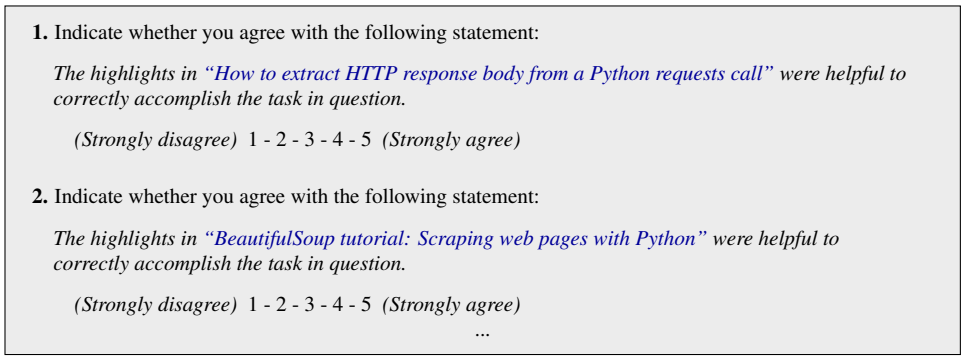


Figure 6.6: Questions asking a participant to rate the usefulness of the highlights shown in two artifacts; by clicking on the name of an artifact, a participant could revisit the highlights of that artifact

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.

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

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 [25] to check for normality and then, due to deviation from a normal distribution ($p\text{-}value < 0.05$), applying a Wilcoxon-Mann-Whitney test [11]. Results from this test indicate that we cannot draw statistically significant conclusions for the solution scores

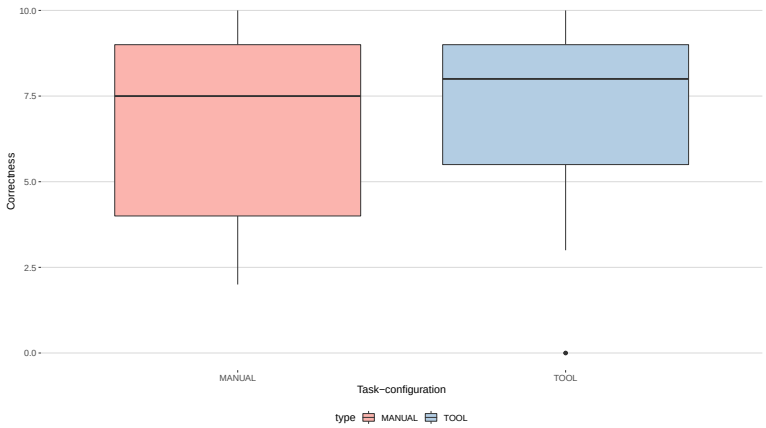


Figure 6.7: Boxplots showing aggregated correctness scores for task performed with and without tool support

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

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

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

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

[5] As Section 6.3.5 details, the lack of statistical power is not surprising and is a common risk to between groups comparisons [8].

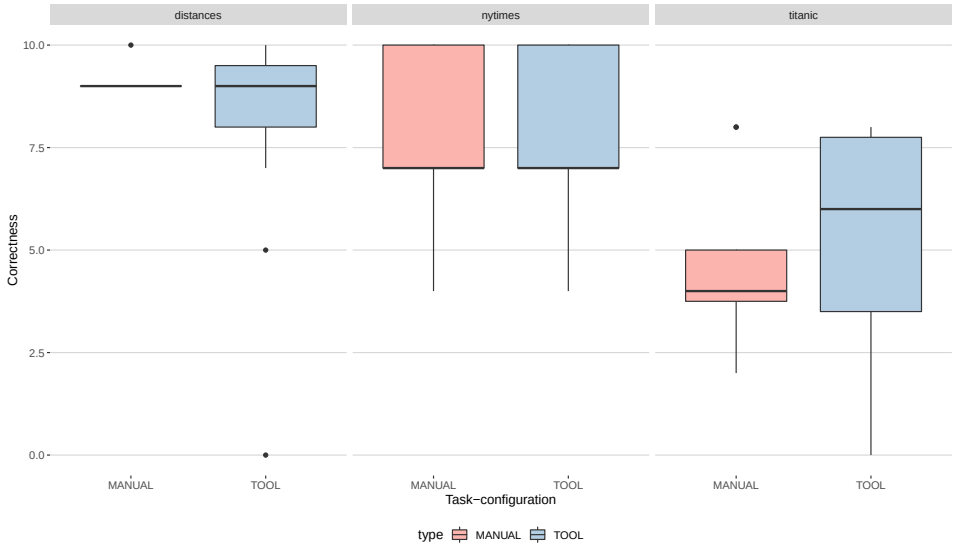


Figure 6.8: Boxplots showing correctness scores for each task performed with and without tool support

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

6.3.2 Usefulness Analysis

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

Metrics

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

complete each task in our experiment, plotting responses using a diverging stacked bar chart [23].

Data

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

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

Results

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

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

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

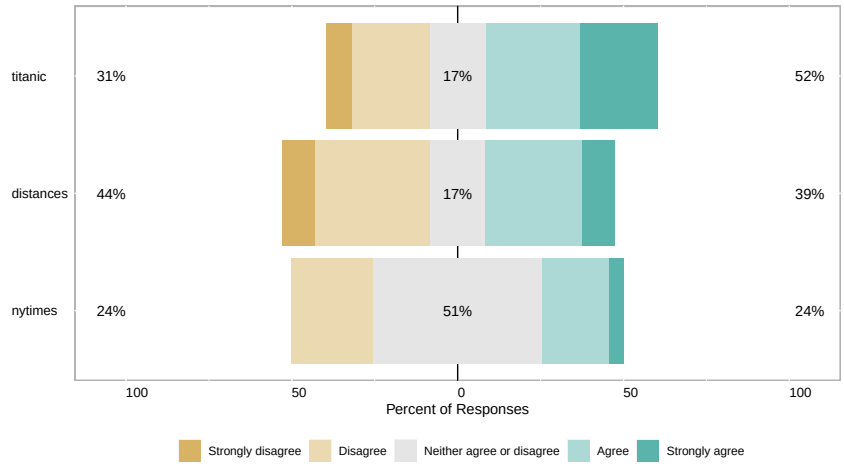


Figure 6.9: Diverging stacked plot of the usefulness of the text automatically identified for each task

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

Surprisingly, when we look at the ratings per type of artifact (Figure 6.10), API documents had the most useful highlights. For this type of artifact, we observe that positive ratings originate from artifacts that follow a *'how to'* format, which is not conventional for API documents [2, 18]. The highlights on Stack Overflow artifacts were also perceived as useful and participants expressed familiarity with this type of artifact, where the highlights helped them determine parts of the page to ignore—"*the highlights [on Stack Overflow] helped me quickly determine which parts of the page to ignore*".

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

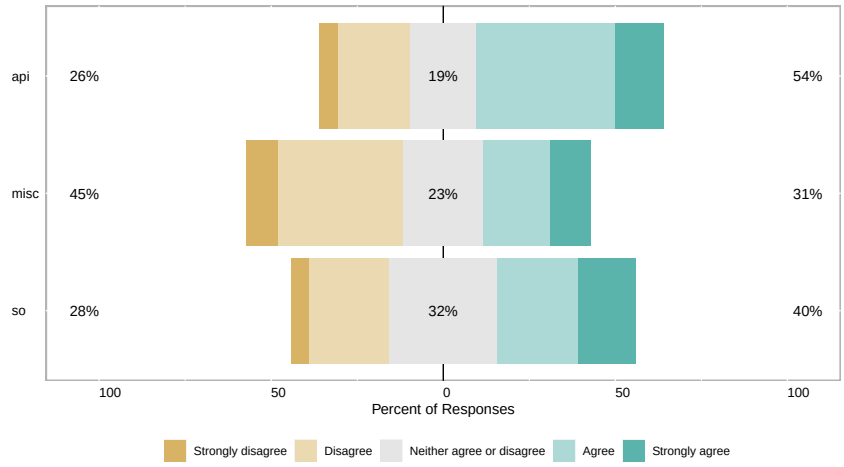


Figure 6.10: Diverging stacked plot of the usefulness of the text automatically identified for each type of artifact

6.3.3 Comparison of manual and automatically identified task-relevant text

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

Metrics

To investigate the overlap between the participants' manual highlights and the the automatic highlights identified by our semantic-based tool we use *precision*, *recall* [12], and *pyramid precision* [15]. We compute these metrics for each artifact of each task and report their average.

For this analysis, we follow Lotufo et al.'s procedures [10] and we consider any text marked by any participant as relevant. We investigate if our tool automatically identifies text that multiple participants deemed relevant via *pyramid precision*. Details for each metric are as follows.

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

pants deemed relevant (Equation 6.2).

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

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

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

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

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

To illustrate these metrics, consider an artifact with 4 sentences $\{s_1, s_2, s_3, s_4\}$ that have been selected by $\{2, 0, 1, 1\}$ participants, respectively. 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 ($\alpha = 0.68$) [7, 16]. We compare these manually produced highlights to the text automatically identified by our tool.

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

Results

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

Table 6.6 details evaluation metrics artifact-type wise. Stack Overflow posts and API documentation have the highest precision scores. For these types of artifacts, pyramid precision indicates that the text automatically identified on Stack Overflow was the text that several participants deemed relevant. The same does

	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

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 **??**. This comparison is interesting because it let us cross-examine our findings with new data [5, 22]. Figure 6.11 presents boxplots for precision and recall of the BERT technique with no filters for the tasks in the $\text{DS}_{android}$ corpus and the Python tasks (experiment). We observe that the technique's accuracy is comparable across the two evaluations.

6.3.4 Summary of results

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

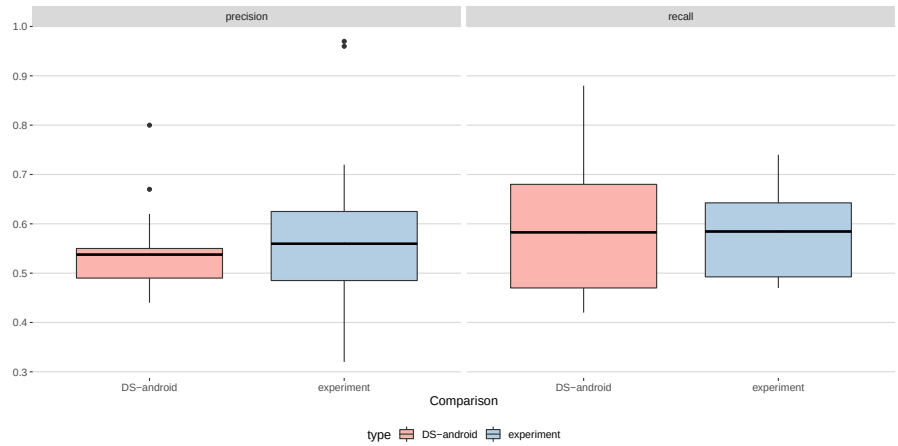


Figure 6.11: Boxplots comparing precision and recall values when applying our semantic-based technique to the DS_{python} and the $\text{DS}_{android}$ corpora

6.3.5 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 [8] 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 [8]. 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

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

programming language of a task [3]. 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 [14, 21], developers work on many different tasks, some of which focus on code [13] while others on information seeking [6], e.g., finding duplicated bug reports or researching visualization libraries to identify the most suitable one [21]. 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 **??**). 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 [26]. 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 TARTI, a tool that embeds a semantic-based technique, assists a developer working on a software task. The experiment examined how 24 participants with software development 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] Colaboratory. https://research.google.com/colaboratory/faq.html. Verified: 2022-03-24. → page 5

[2] D. M. Arya, J. L. Guo, and M. P. Robillard. Information correspondence between types of documentation for apis. *Empirical Software Engineering*, 25(5):4069–4096, 2020. → page 17

[3] 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. → page 23

[4] K. Cwalina and B. Abrams. *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Pearson Education, 2008. → page 2

[5] 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 21

[6] M. K. Gonçalves, C. R. de Souza, and V. M. González. Collaboration, information seeking and communication: An observational study of software developers' work practices. *J. Univers. Comput. Sci.*, 17(14): 1913–1930, 2011. → page 23

[7] K. Klaus. Content analysis: An introduction to its methodology, 1980. → page 20

[8] J. Lazar, J. H. Feng, and H. Hochheiser. *Research Methods in Human Computer Interaction*. Morgan Kaufmann, Boston, second edition edition, 2017. ISBN 978-0-12-805390-4. → pages 14, 22

[9] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932. → page 11

[10] 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. → page 18

[11] 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. → page 13

[12] C. Manning, P. Raghavan, and H. Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010. → page 18

[13] 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. → page 23

[14] 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 23

[15] 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. → page 18

[16] 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 20

[17] 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. → page 2

[18] M. P. Robillard and R. Deline. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011. → page 17

[19] 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 3

[20] 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 3

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

[22] 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 21

[23] R. Spence. *Information visualization*, volume 1. Springer, 2001. → page 16

[24] 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 4

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. → page 13

[26] 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. → page 23