On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization

Junayed Mahmud
junayed.mahmud@ucf.edu
University of Central Florida
Orlando, FL, USA

Nadeeshan De Silva
kgdesilva@wm.edu
William & Mary
Williamsburg, VA, USA

Safwat Ali Khan
skhan89@gmu.edu
George Mason University
Fairfax, VA, USA

Seyed Hooman Mostafavi
smostaf6@gmu.edu
George Mason University
Fairfax, VA, USA

SM Hasan Mansur
smansur4@gmu.edu
George Mason University
Fairfax, VA, USA

Oscar Chaparro
oscarch@wm.edu
William & Mary
Williamsburg, VA, USA

Andrian Marcus
amarcus7@gmu.edu
George Mason University
Fairfax, VA, USA

Kevin Moran
kpmoran@ucf.edu
University of Central Florida
Orlando, FL, USA

ABSTRACT

One of the most important tasks related to managing bug reports is *localizing the fault* so that a fix can be applied. As such, prior work has aimed to automate this task of bug localization by formulating it as an information retrieval problem, where potentially buggy files are retrieved and ranked according to their textual similarity with a given bug report. However, there is often a notable *semantic gap* between the information contained in bug reports and identifiers or natural language contained within source code files. For user-facing software, there is currently a key source of information that could aid in bug localization, but has not been thoroughly investigated – information from the graphical user interface (GUI).

In this paper, we investigate the hypothesis that, for end user-facing applications, connecting information in a bug report with information from the GUI, and using this to aid in retrieving potentially buggy files, can improve upon existing techniques for text retrieval-based bug localization. To examine this phenomenon, we conduct a comprehensive empirical study that augments four baseline text-retrieval techniques for bug localization with GUI interaction information from a reproduction scenario to (i) filter out potentially irrelevant files, (ii) boost potentially relevant files, and (iii) reformulate text-retrieval queries. To carry out our study, we source the current largest dataset of fully-localized and reproducible real bugs for Android apps, with corresponding bug reports, consisting of 80 bug reports from 39 popular open-source apps. Our results illustrate that augmenting traditional techniques with GUI information leads to a marked increase in effectiveness across multiple metrics, including a relative increase in Hits@10 of 13-18%. Additionally, through further analysis, we find that our studied augmentations largely complement existing techniques, pushing additional buggy files into the top-10 results while generally preserving top ranked files from the baseline techniques.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Application specific development environments.*

KEYWORDS

Bug Localization, GUI, Natural Language Processing, Mobile apps

ACM Reference Format:
Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, SM Hasan Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. 2024. On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization . In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3608139

1 INTRODUCTION

The process of bug report management has been demonstrated to consume large amounts of developer's time [61, 81]. One of the more difficult bug management tasks is related to *localizing the described fault*, as it requires reasoning between the description of a bug and the source code of a software project. This process is often further complicated by quality issues related to various elements of bug descriptions, such as reproduction steps, stemming from inaccurate or incomplete information provided by reporters [15, 18, 22, 72].

Researchers have been working to automate *bug localization* by developing approaches that automatically retrieve and rank potentially buggy files or code snippets to help expedite localization effort. A substantial body of research formulates bug localization as a text retrieval-based (TR) problem [27] — see Section 6. In general, these approaches use the bug report to formulate a query and return a list of source code elements (files, classes, methods, *etc.*), ranked by their likelihood that they contain the bug.

The key assumption made by TR-based bug localization approaches is also their biggest limiting factor. That is, while such techniques operate on the premise that bug reports and the corresponding buggy source code will share terms, research has also documented a notable *semantic gap* between the information that reporters provide in bug reports, and the identifiers and the documentation written by developers in source code [54, 57, 101]. Researchers have recognized this issue, and have attempted to augment TR-based bug localization approaches with various techniques.

Many approaches focused on processing the text in the bug reports or the source code (*e.g.,* through abbreviation expansion [39]), while others focused on query reformulation, or automatically augmenting a query generated from a bug report using information various sources [66]. Another line of research focused on using information orthogonal to the code and bug report vocabulary to boost the ranks of the retrieved buggy code elements, such as, execution information (extracted form execution or stack traces [82, 94]), code dependencies (extracted via static source code analysis [29]), or historical information (extracted from repositories [94]).

In this paper, we explore whether it is possible to improve TR-based bug localization leveraging an information source not yet explored by prior work – *information from the graphical user interface (GUI).* GUIs encode latent patterns related to application features in both pixel-based (*i.e.,* screenshots) and metadata-based (*i.e.,* `html/uiautomator`) representations [59]. Our rationale is that GUI interaction information can be easily obtained and represents "high-level" execution related information, where code elements are directly linked to higher level program functionality through the UI, as opposed to "low-level" execution information extracted from execution traces, which can be difficult to acquire. Once collected, this high-level GUI information can then be used to boost rankings of related buggy code elements. Further, unlike low-level execution traces, GUI-related information is rich in textual elements, and can also be used for query reformulation. Intuitively, if the buggy code is *related* to the app screen where the bug is observed, then the GUI and interaction information from that screen can be used to help locate the buggy code easier. Conversely, if the buggy code is not related to the buggy screen or user interactions, then we expect that the GUI-related information will not hinder the bug localization process. We refer to the collective data related to both user interactions and the software interface itself as *GUI interaction data.*

To investigate whether GUI interaction data can aid in TR-based bug localization, we carry out a comprehensive empirical study that *augments* four baseline TR-based approaches: BugLocator [99], a Lucene-based approach [2], and two neural-based text embedding approaches (based upon the sentenceBERT [69] and UniXCoder [36] neural language models). The GUI information that we use for augmentation is collected from a recorded set of GUI interactions that reproduce a given bug, which can be easily collected manually by developers, or automatically by any of a number of bug reproduction techniques [31, 42, 98]. Once these GUI interactions are collected, we assess the effect on retrieval performance by using information from the GUI to: (i) filter out potentially irrelevant files that are not related to the buggy GUI screen; (ii) boost potentially relevant files that are related to the buggy GUI screen; and (iii) reformulate queries using information from the buggy GUI screen.

In this study, we focus on localizing bugs in Android apps, which typically manifest themselves in the GUI. This means that these bugs lead to unexpected app behavior (or a faulty state) that is visible to the user, including app crashes (*e.g.,* when the app suddenly closes), navigation issues (*e.g.,* when the app leads the user to an unexpected screen), and incorrect output shown on the screen, among others. As such, to support our study, we have manually sourced and validated the current largest dataset of fully-localized and reproducible bugs for Android applications with corresponding bug reports, consisting of 80 bug reports from 39 popular open source Android applications. We compared our baseline TR-based bug localization approaches with thousands of augmented configurations using different types and amounts of GUI interaction data, as well as different query reformulation techniques.

Our results illustrate the benefit of leveraging GUI interaction information, as the best-performing configurations of the techniques augmented with GUI information outperformed their baseline for *all* TR-based techniques, with Hits@10 improving by 13-18% overall. A deeper investigation into these results show that our studied augmentations help rank more bugs in the top-10 retrieved results, while generally preserving the top-ranked buggy files from the baseline techniques. Overall, the results support our rationale for leveraging GUI information to improve bug localization, and suggest that future work should explore this topic further.

In summary, this paper makes the following contributions:

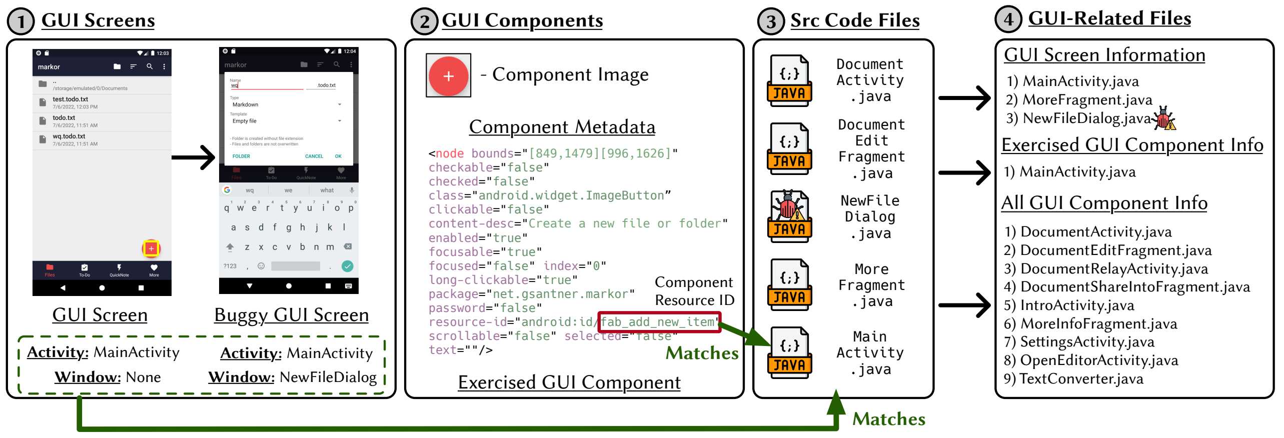
- A new dataset of 80 fully-localized and reproducible Android bugs from 39 popular Android apps, complete with bug reports and recorded scenarios (and metadata) that reproduce each bug,
- A quantitative analysis of the effect of using GUI interaction information on the effectiveness of four TR-based bug localization techniques, via three augmentation methods: (i) filtering, (ii) boosting, and (iii) query reformulation, and
- A replication package [10] that contains our dataset, code, and experimental infrastructure to aid in the replication and reproduction of our results.

2 BACKGROUND & MOTIVATION

In this section, we provide background on the various GUI-related terms and concepts related to our study, explain our methods for augmenting text-retrieval techniques for bug localization with GUI-related information, and include a motivating scenario that illustrates the intuition behind the augmentation methods.

2.1 GUI-related Information

In this paper, we analyze GUI information of Android apps, namely **GUI screens**, **GUI components** (*i.e.,* the GUI widgets/elements that compose those screens), and **Exercised Components** (*i.e.,* the GUI components that the user interacts with via taps, swipes, *etc.*). To illustrate the definitions of these various GUI-related concepts, we provide an example in Figure 1, oriented around a set of screens that reproduce the bug described in the report shown in Figure 2. ① **GUI screens** represent the UI canvas upon which **GUI components** are drawn, wherein the screen is composed of a hierarchy of interactive components and GUI containers that group individual components together such that they may adapt to various screen sizes and dimensions. In Android, **screens** are referred to as Activities [5], and each activity corresponds to one or more `.java`/`.kotlin` class files that define the functionality of the screen, and a set of resource `.xml` files that describe the layout of components on the screen. The code and resource files directly make up the static definition of the screen in the code. In addition to Activities, Android also allows for the definition of Fragments [3], which are reusable groups of **GUI components** (*e.g.,* menus, dialog boxes...). A **GUI screen** can display a Window, as shown in the *buggy screen* of Figure 1-①, which can display a dialog, toast, or other **GUI component** in the foreground of the Activity. Fragments and Windows are also defined in their own `.java`/`.kotlin` class files and `.xml` resource files.



```
<node bounds="[849,1479][996,1626]"
checkable="false"
checked="false"
class="android.widget.ImageButton"
clickable="false"
content-desc="Create a new file or folder"
enabled="true"
focusable="true"
focused="false" index="0"
long-clickable="true"
package="net.gsantner.markor"
password="false"
resource-id="android:id/fab_add_new_item"
scrollable="false" selected="false"
text=""/>
```

Figure 1: Example of GUI-related information used in this study

In addition to the static definitions of **GUI screens** in class and resource files, it is also possible to extract a runtime representation of a given **GUI screen** using the Android `uiautomator` framework [1], which queries a device's `ViewServer` to extract metadata about the various **GUI components** currently rendered on the screen. Figure 1 illustrates the last two **screens** of the bug reproduction for the Markor app [6], along with their Activity (`MainActivity`), and Window (`NewFileDialog`) information. Note that both screens correspond to the same Activity, but the second screen displays a foreground popup defined in the `NewFileDialog` Java class file.
② **GUI Components** are the UI elements, defined by developers and rendered to the screen, with which end-users interact via GUI-level actions (taps, swipes, long touches, *etc.*). The presentation attributes (color, size, component type, *etc.*) are defined in the app source code and in the resource files described earlier. These component definitions can be attached to `Event Listeners` that cause Java/Kotlin code to be executed when a certain GUI-level action (*e.g.*, tap) is performed on the component. To link the **component** definition to the app Java/Kotlin code, unique Resource IDs are used. As mentioned earlier, in addition to the static definition of GUI components, a dynamic representation can also be extracted using the `uiautomator` framework. This dynamic **component** metadata is shown as in Figure 1-② for the + button in the bottom right hand side of the first GUI Screen (highlighted in yellow). This metadata contains information related to the `size`, `class`, and `resource-id` of this **component**, among other attributes. If you refer to the screen flow shown in Figure 1-①, the + button was the **component** that the user interacted with to navigate to the buggy **screen** with the dialog box. We refer to **components** that a user interacted with during a GUI-level execution scenario as **Exercised Components**.

2.2 Motivating Example

In order to illustrate our intuition about leveraging GUI interaction information to improve bug localization, in this subsection we illustrate the effect that this information has on an example bug report from the Markor [7] app shown in Figure 2.

The Markor app is a text-editor primarily aimed at supporting quick note-taking and managing to-do lists. It is also relatively popular – the GitHub repository currently has 2.6k stars. For each file created, Markor supports different ways of formatting the text (*e.g.*, Markdown) that is configurable through a note creation dialog. The observed buggy behavior of the report (illustrated in Figure 2) is related to the formatting method of a given file being reset in the File creation/editing dialog, wherein it always defaults to "Markdown", even when it is initially set to something else. For instance, if a user were to set the note to "plain-text" formatting, and then later return to edit the note, the formatting type would be reset to "Markdown".

Bug Report Title: New file dialogue always says Type: Markdown rather than previous type used

Steps to reproduce:

1. Create a new file, change type to e.g. todo.txt, save it
2. Create another new file
3. Observe the dialogue correctly remembers .todo.txt as the previously used file extension, but below that it says Type: Markdown

... the second new file is named something.todo.txt, so the app is correctly remembering the user's last settings, but nonetheless it looks odd and confusing to the user.

It also makes it harder when you want to make a Markdown file as you have to choose something other than Markdown then choose Markdown again before the file extension will change to .md.

Figure 2: Example Bug Report from the Markor[7] App

Let's consider an existing TR-based bug localization approach, BugLocator [99], which uses term similarity and document length in conjunction with a Vector Space Model (VSM), to rank buggy files. When we use the bug report from Figure 2 to BugLocator as a query, the top 3 returned files are (1) `SearchOrCustomTextDialogCreator.java`, (2) `TodoTxtHighlighter.java`, and (3) `TextFormat.java`. However, as shown in Figure 1-③ the actual buggy file is `NewFileDialog.java`, as the bug occurs in the Dialog box used for creating and modifying files. This file is ranked ***35th*** by BugLocator. However, for this particular bug, we can observe in Figure 1-① that the screen which exhibits the buggy behavior is composed of the `MainActivity` activity, and the `NewFileDialog` fragment that makes up the popup dialog box shown in the screenshot. For this particular bug, *the java class file* `NewFileDialog.java` *that corresponds to the window fragment in the buggy screen* *is the buggy file* (shown in Figure 1-③). Hence, if the TR-based bug localization approach had knowledge of this GUI-related information, it likely could have improved the ranking of the buggy `NewFileDialog.java` file.

This example illustrates the potential promise of leveraging GUI interaction data to improve existing text retrieval-based bug localization approaches. While the buggy file for a given report may not always correspond neatly to the class that implements a buggy screen's Activity or Window, information related to event-listeners for various GUI-components that are relevant to a bug reproduction scenario, or screen information from earlier bug reproduction steps, may help identify and rank buggy files. In this study, we aim to investigate the potential for such *GUI Interaction Information* to augment TR-based bug localization approaches.

3 DESIGN OF EMPIRICAL STUDY

The goal of this study is to investigate: (1) the extent to which GUI interaction information from Android apps leads to more effective automated bug localization, and (2) how this information can be most effectively used to increase bug localization effectiveness. With this in mind, we formulate the following overarching research question (RQ), refined into four specific research RQs:

RQ: *What is the impact of using GUI interaction information on text-retrieval bug localization performance?*

- **RQ$_1$:** *What is the effect of using GUI interaction information from different numbers of screens on bug localization performance?*
- **RQ$_2$:** *What is the effect of the type of GUI interaction information and augmentation method on bug localization performance?*
- **RQ$_3$:** *What are the overall best-performing combinations of using GUI interaction information?*

To answer these research questions, we have developed methods to link GUI Information to potentially buggy files and to augment existing TR-based bug localization techniques with this file information, which we describe in Secs. 3.1 and 3.2. Additionally, we have manually sourced the largest corpus of fully-localized and reproducible Android bugs, consisting of 80 bug reports from 39 popular open source Android apps (see Sec. 3.3).

We study four *baseline* techniques – one existing TR-based bug localization technique, one traditional term matching technique, and two neural text-embedding approaches, described in Sec. 3.4. We assess the performance of these baseline techniques in locating buggy files for the 80 bugs, with and without using GUI-information augmentation. We measure bug localization performance using the metrics described in Sec. 3.6, and answer the RQs in Sec. 4.

3.1 Mapping GUI Terms to GUI-Related Files

Given that the goal of our study is to determine how GUI information can be used to augment bug report-based fault localization, we aim to map **terms** extracted from **GUI Screens** and **Components** to source code **files** that may be useful for the localization process. These **GUI-related terms** are later used for query reformulation, whereas the **GUI-Related Files** are used to re-rank retrieved files. Our study assumes the scenario wherein the developer has access to a GUI-level reproduction scenario, which contains screenshots, the uiautomator metadata, and the `resource-id` and GUI-level action (*e.g.,* tap, long touch) for each screen interaction. This information is easy to collect, and could be collected manually, wherein the developer records a video and translates this video to a scenario using a tool such as V2S [13, 14, 38] or GifDroid [32], or can be collected automatically using tools that reproduce Android bug reports such as Yakusu [31], RecDroid+ [97, 98], or the recent approach by Zhang *et al.* [96]. Given this information, we aim to link *key terms* from **GUI-screens** and **GUI-components** to potentially buggy files, or **GUI-Related Files**. Given that the bug occurs at the end of a given reproduction scenario, but the buggy behavior may be triggered or exercised earlier in the scenario, we explore using information from the buggy **screen** and the information from the prior 1-3 **screens**. Next, we describe how we identify **GUI-Related Files** from **GUI Screen** and **component** metadata (using the relevant **terms**).

Table 1: Mapping of GUI Terms to GUI-Related Files

GUI Information	Terms	Files
Screen	Activity and Window names queried from the Android `ViewServer`	`.java` files with file names matching the terms
(Exercised) GUI Screen Components	`resource-id(s)` of the (interacted) components from the dynamic `uiautomator` metadata	`.java` files that contain invocations of the `resource-id(s)` in event listeners

We define three different types of **GUI-Related Files**: (1) those related to the Activity and Window information from a given **GUI Screen**; (2) those related to the **Exercised Components**; and (3) those related to all the components that appear on the selected screens of a given reproduction scenario, which we refer to as **GUI Screen Components**. We next discuss how each type of **GUI-Related File** is derived, also summarized in Table 1.

<u>- Mapping GUI Screen terms to Files:</u> To derive the GUI-Related Files for a given *GUI screen*, the Activity and Window names from the dynamic UI metadata generated by the Android `ViewServer` are taken as terms and directly matched with their corresponding `.java` class file names. For instance, the *GUI Screen* information for the two screens shown in Figure 1 is `MainActivity`, `MoreFragment`, and `NewFileDialog` as shown in Figure 1-④, which map to the Java files with the same names.

<u>- Mapping GUI Screen Component terms to Files:</u> To derive GUI-Related Files for *all* GUI components of the considered screens, the `resource-id` for all components for a given screen that support interaction are extracted from the `uiautomator` metadata and used as terms, and then invocations of these `resource-id` terms are automatically identified in event-listeners in the app source code. The corresponding files that contain these event listeners are taken as the GUI-Related Files for Screen Components. This generally leads to a large set of GUI-Related Files, as matching the `resource-ids` of a large number of interactive components on a given set of screens to event-listeners typically leads to many affected files. For the example shown in Figure 1, the `resource-ids` for 41 components are extracted, which in turn map to event-listeners in 77 code files, some of which are shown in Figure 1-④.

<u>- Mapping Exercised GUI Component terms to Files:</u> To derive GUI-Related Files for an *Exercised GUI Component*, the same approach as described for GUI Screen Components is followed, but *only* for those components on the screen which were exercised as part of a bug reproduction scenario. Any source code file that contains an event-listener for the component is identified is considered as a GUI-Related File. For the example in Figure 1, there is only one Exercised Component, and only one invocation of this exercised component, in the `MainActivity.java` file (shown in Figure 1-④).

It is important to note that in our study we do not only consider each type of GUI information in isolation, but also consider combinations, represented as unions of the GUI Information Terms for two different types. We illustrate our considered combinations in Table 2, organized according to the number of files they typically return. Note that GUI Information terms stemming from Exercised

Table 2: Types of GUI-Related Files

Low Number of Files
GUI Screen
Exercised GUI Components
Medium Number of Files
GUI-Screen + Exercised GUI Components
High Number of Files
GUI Screen Components
GUI Screen + GUI Screen Components

GUI Components are a subset of the GUI Screen Components, thus we do not combine these information types. Additionally we ignore components without `resource-ids` and do not consider interactions with the Android `Back` button from the bottom navigation bar as exercised components, as interactions with this component cannot be mapped back to the event listeners in the app code.

3.2 Text-Retrieval Augmentation Methods

Below, we describe how the GUI-Related Files can be used to augment existing techniques for text retrieval via Query Reformulation and Re-Ranking, to (potentially) improve bug-localization.

In the context of our bug localization process, typically text-retrieval techniques use a pre-processed version of the bug report as a query to retrieve source code files that contain similar terms to those used in the bug report. The method of calculating query-to-document similarity can vary from relatively simple methods, such as using term frequency (*e.g.,* `tf-idf` vectors), to more complicated techniques that use neural text embeddings.

3.2.1 ***Reformulating Queries using GUI Terms.*** The first type of augmentation techniques that we define are **Query Reformulation** techniques [33]. These techniques modify the textual query used by TR-based techniques to retrieve relevant files. Given this setting, we define the following two reformulation techniques:
- Query Expansion: In this technique, the dynamic Activity/Window names for GUI screens, and GUI component resource IDs for a given GUI component type (*e.g.,* Exercised Components or Screen Components) are appended to the bug report to form the query.
- Query Replacement: In this technique, the dynamic Activity/Window names for GUI screens and GUI component resource ids for a given GUI component type (*e.g.,* Exercised Components or Screen Components) are used to replace the bug report as the query.

3.2.2 ***Re-Ranking using GUI-Related Files.*** We explore three different techniques for **re-ranking** files using GUI information:
- Filtering: In this strategy, all files that *do not* match the GUI-Related Files for a given information type are filtered out from the corpus of potentially buggy files.
- Boosting: In this strategy, files that *match* the GUI-Related Files are boosted to the *top* of the ranked list of results returned by a given text-retrieval technique while preserving the relative order of those files originally ranked by the technique.
- Filtering + Boosting: The final re-ranking strategy combines both filtering and boosting, wherein files are filtered using a GUI information type that has a higher number of files, and boosting is performed with a type that returns a lower number of files. This is due to the fact that filtered files, cannot be subsequently boosted.

3.3 Dataset Construction

3.3.1 ***Bug Report Selection.*** Given that no prior dataset of fully-localized and reproducible bug reports for Android apps exists, we constructed our own using a rigorous manual process. We built a *ground-truth* bug localization dataset by using as many bug reports as possible from the AndroR2 dataset [42, 83], which consists of 180 manually reproduced bug reports for popular open source Android applications hosted on GitHub. These reports were systematically collected from the project's issue trackers according to the following criteria, as reported by Wendland & Johnson [42, 83]: they contain the label "bug", were opened in the past five years and closed at the time of the mining (November 2020), contain the word "steps" in their content, and report a non-trivial bug (*i.e.,* did not occur by opening the app). The bug reports are grouped into four categories that represent a bug type, namely *output*-, *cosmetic*-, *navigation*-, and *crash*-related bugs. Furthermore, each bug report is associated with additional (meta)data, including the commit ID of the app version that contains the bug, the buggy app's `apk` file, and the link to the GitHub issue where it was originally submitted.

We explored all bug reports in the AndroR2 dataset and utilized a subset of reports that fit the necessary criteria for our study (*i.e.,* they were reproducible and able to be localized to source code files). To identify the buggy files among the set of all files from the buggy version of the app, we followed a systematic procedure that involved at least two authors examining each bug report. Specifically, two authors inspected the content of the bug reports (including the comments) to find any references to the commits that fixed the bug. They then inspected the commit messages and specific code changes to determine if they appeared to fix the bugs. Among the 180 bug reports in AndroR2, one bug report does not exist anymore, and eight bug reports do not contain any obvious commit ID or version information for the bug being reported, making it difficult to extract the buggy source code. As a first step in filtering the dataset, we excluded these bug reports and investigated the remaining 171 bug reports.

In checking for bug-fixing commits, the two authors were able to source 120 bug reports with bug-fixing commit IDs that were confirmed to fix the reported bug, and on average, each bug report contained ≈1.90 bug-fixing commit IDs. When no commit was mentioned in the bug reports, or the fixed commit ID mentioned in bug reports did not appear to resolve the error, the two authors followed references to duplicate reports and collected the referenced commits, again confirming that these commits did indeed fix the bug. We followed this procedure for the remaining 51 bug reports (of the 171 filtered reports) and were successful in gathering bug-fixing commit IDs in 27 cases – resulting in 147 bug reports with confirmed bug-fixing commit IDs. Although in certain cases our ground-truth uses the bug fixing commits from duplicate bug reports, in this study we use the original bug report contents as reported by AndroR2 as the query for TR-based bug localization. Once bug-fixing commits were identified, two authors performed two rounds of coding, during which they compared the file diffs between the buggy (i.e., the latest app release commit ID before the bug was reported) and the fixed version of the app (based on the commit IDs) to identify the files that contained bug fixing changes. If there were disagreements, a third author discussed the cases with the two coders and the three authors reached a consensus. Note that this process only identified code files from the app's buggy version that had bug fixes (*i.e.,* code changes), thus excluding code files that were *added* in the fixed app version and ignoring changes in white space and code comments. This set of 147 bug reports formed the set used for isolating the buggy files and collecting the required bug reproduction scenarios and GUI information.

3.3.2 *Coding & Collecting GUI Interaction Data.*

The ground-truth construction process was executed independently by two authors across two sessions. During the first session, two authors randomly selected 24 bug reports from all failure types of the AndroR2 dataset among the 147 bug reports filtered as described above. The two authors worked together in this session and discussed each bug in order to derive a common understanding of the coding process. Given that some of the studied baseline techniques *only* operate on `.java` files, we discarded two bugs where the erroneous behavior was isolated the `.xml` resource files and one bug containing buggy Kotlin files. As such, 21 bug reports were coded as part of this first session.

To collect the GUI interaction data for these 21 bug reports, we followed the record-and-replay methodology used by Cooper *et al.* [25], which includes installing and using the buggy app on an emulator, recording a usage scenario that reproduces the bug while collecting *screen recordings* and `getevent` traces (*i.e.,* traces that include low-level GUI-related information) using the Android `getevent` utility. These scenarios are then replayed in a step-by-step manner by converting the recorded `getevent` actions to a set of `adb` commands. During the step-by-step replay, screenshots and GUI metadata were collected before and after each GUI action that includes information on the Activity and Window, as well the GUI hierarchy extracted via the `uiautomator` tool that contains the UI metadata for all components displayed on the screen. Since the AndroR2 bug reports provide the steps to reproduce the bug (S2Rs) and the buggy APK of the apps, we were able to reproduce the bugs on an emulator. Specifically, two authors reproduced the bugs manually on Pixel 2 Android emulator with the specific Android version mentioned in the AndroR2 dataset. Among the 21 bug reports, we could not reproduce the bug for one bug report due to a lack of `getevent` support for recording rotation events. Therefore, we included the remaining 20 bug reports in our dataset.

In the second round of coding, the two authors worked with the remaining 123 bug reports. During this process 16 additional bug reports were identified to have bug fixing changes on `.xml` files only, 14 more contained Kotlin code, one used web-based technologies and hence had no Java/Kotlin/resource files, and two bug reports with tangled commits (i.e., a large number of file changes) in the fixed commits. We excluded these 33 bug reports resulting in 90 bug reports having at least one buggy Java file.

During the collection process for the GUI Interaction data for the remaining 90 bug reports, were not able to collect GUI interaction data for 30 bug reports due to the one of following reasons: (1) the buggy behavior could not be reproduced; (2) bugs could not be reproduced due to the constraints of the step-by-step replay process (*e.g.,* logging into some apps was not possible); and (3) the step-by-step replay process has limitations including the inability to rotate the screen or execute fast swipe gestures. We excluded these 30 reports from our dataset, leaving 60 bug reports.

Finally, the two authors then worked separately to identify relevant buggy files reaching consensus in 53/60 (≈88.33%) of the cases (*i.e.,* the set of buggy code files matched). When there was no consensus, the three authors mutually finalized the buggy files. In total, we identified the buggy files and GUI metadata containing screenshots, XMLs and event execution information for 80 bug reports (20 in the initial round and 60 in the subsequent round). The collection of the GUI interaction data took ≈60-70 hours for the 80 bug reports originating from 39 apps (*i.e.,* ≈1.5 hours per app). However, it should be noted that in practice, the GUI interaction data that we collect in this study could be collected automatically through any number automated input generation tools for Android [31, 45, 96–98].

3.4 Baseline Techniques

3.4.1 *BugLocator.*

Zhou *et al.* introduced BugLocator [99], which uses a revised vector space model (rVSM) to obtain a ranked list of buggy files when a bug report is used as a query. Initially, a classic VSM based approach calculates the cosine similarity between the vector representations of the query and a document. BugLocator then ranks longer documents higher, assuming that these files are more likely to contain bugs. BugLocator is also capable of learning from previously fixed bugs by constructing a three-layer heterogeneous graph and computing a similarity score between past confirmed buggy files and the corpus of files under analysis. However, in this paper, we do not make use of this feature of BugLocator, as we do not have the necessary data.

3.4.2 *Neural Embeddings via SentenceBERT.*

In the second baseline technique, we use the SentenceBERT [69] neural language model, which is a modification of a pre-trained Bert model [28]. SentenceBERT augments the traditional Bert model with siamese and triplet networks allowing for better support of tasks such as clustering and semantic search with less computational overhead. The model was fine-tuned on a popular natural language inference dataset [84] and outperforms state-of-the-art approaches.

We use the sentence transformer *msmarco-distilbert-base-v3* implementation [8] from the HuggingFace library [4] for our implementation of SentenceBERT. The model uses 768-dimensional vector space and has a maximum sequence length of 510. As such, while all of our bug reports fit within this sequence length, certain source code files may exceed it. Thus, we split source code files into different segments with a maximum length of 510 tokens, and created an embedding for each segment. We compute cosine similarities between the bug report embedding and each segment of each source code file, and take the segment with the highest similarity value to the query as the similarity value for a given file, which we use for ranking the files.

3.4.3 *Neural Embeddings via UniXCoder.*

In addition to neural language models trained on general natural language understanding tasks, we also wanted to explore how GUI information may complement document embeddings generated from a model trained primarily for *code understanding* tasks. As such, for the third baseline, we use the UniXCoder [36] model that is based on a multi-layer Transformer [76] architecture. We use the open-source implementation of UniXCoder [11] to create embeddings of the source code and bug reports. The *unixcoder-base* model is pre-trained on the CodeSearchNet [41] dataset, one of the largest model training datasets for code understanding tasks containing two million code-comment pairs (across six programming languages). Similar to SentenceBERT, this model also inputs a maximum token

Table 3: Combinations of the Filtering+Boosting re-ranking methods and GUI-Related File types considered. GS = GUI Screen; EGC = Exercised GUI Component; SC = GUI Screen Components.

Flt \ Bst	GS	EGC	GS+EGC	SC	GS+SC
GS					
EGC					
GS+EGC	✓	✓			
SC	✓	✓	✓		
GS+SC	✓	✓	✓	✓	

length, but of 512 opposed to 510. Therefore, we follow the same segmentation and similarity score calculation method as we do for SentenceBERT, again using a cosine similarity measure.

3.4.4 *Lucene*.

For the fourth baseline, we use Lucene [2], an open-source Java project that provides features to retrieve relevant documents. Lucene uses a vector space model and TF-IDF document vector representations to rank buggy files based on an input query.

3.4.5 *Preprocessing of Queries and Source Code Files*.

We perform the following preprocessing steps, commonly used in past work on TR-based bug localization, for both queries (bug reports/reformulated queries) and source code files: splitting camel case and removing numbers, punctuation, tokens of length $1-2$, any special characters not part of English alphabets, and Java keywords. These steps were used for all techniques except source code for BugLocator as this technique applies its own preprocessing [99].

3.5 Approach Configurations

Our study has four main configuration parameters: (i) the number of GUI screens preceding the buggy screen to be used for GUI-Related File derivation (we investigate between 2-4 screens, including the buggy screen); (ii) the type of GUI interaction information used (we investigate the five combinations shown in Table 2); (iii) the query reformulation techniques used (query replacement or expansion); and (iv) the re-ranking techniques used (filtering only, boosting only, and filtering + boosting).

We examine ***all*** combinations of these four parameters. However, one of our re-ranking techniques (filtering + boosting) is tightly coupled to the type of GUI information used, and as such we only investigate the feasible combinations (*i.e.,* where boosted files are a subset of filtered files) of the information shown in Table 3. The reason we cannot explore all types of information is due to the fact that we cannot filter using a more restrictive set of GUI-Related Files (*e.g.,* GUI Screen-related Files) and boosting with a less restrictive set (*e.g.,* GUI Screen Component-related Files) as many (if not all) of the files filtered out would be the same that would then be subsequently boosted. In total, **we explore 657 configurations** of GUI Information for each baseline, for a **total of 2,628 configurations** across all of our experiments. Table 4 shows the total number of configurations, where the number of configurations for each augmentation technique is calculated by multiplying feasible combinations of GUI types with the number of GUI screens.

3.6 Metrics and Comparative Evaluation

We adopt Hits@K, a metric widely used in the literature [20, 65, 99], to evaluate the performance of augmenting our baseline techniques with GUI-related information.

Table 4: Number of configurations. GIT = GUI Information Type; RRT = Re-ranking Techniques; QE = Query Expansion; QR = Query Replacement

Augmentation	# GIT (RRT)	# GIT (QE)	# GIT (QR)	# Screens	× configs
Filtering	5			3	15
	5	5		3	75
	5		5	3	75
Boosting	5			3	15
	5	5		3	75
	5		5	3	75
Filtering + Boosting	9			3	27
	9	5		3	135
	9		5	3	135
Query Reform.		5		3	15
			5	3	15
Total Number of Configurations					657

Hits@K: This metric computes the percentage of queries for which a bug localizer retrieves at least one buggy file within the top-K files returned. We report results for K=1,5,10 as past work has illustrated that the likelihood that a developer would look beyond 10 results is low [78]. We adopt this metric as it supports a practical scenario for bug localization. Hits@K values fall in [0, 1], where higher values mean higher bug localization effectiveness.

Relative Improvement to Hits@10: In addition to the Hits@K metric, given that the aim of our study is to compare the baseline techniques with their augmented counterparts, we also defined a comparative metric that measures the improvement of Hits@10 of one of our studied GUI Information configurations to a given baseline technique. This is defined as:

$$\frac{Hits@10GUI - Hits@10Base}{Hits@10Base}$$

4 EMPIRICAL RESULTS

In this section, we present the results of our empirical analysis organized by RQ. Of the 2,628 configurations of baseline techniques augmented with GUI interaction information, 1,080 configurations resulted in improvement over the baseline in terms of Hits@10, and 1,548 configurations resulted in no improvement or a degradation in effectiveness over the baselines. However, encouragingly, we find that a *small set* of similar configurations of GUI-based augmentation methods tend perform best *across all baseline techniques*, and more encouraging still, these best performing configurations result in marked improvements to Hits@10 (*e.g.,* up to 18%) with little degradation to the ranks of buggy files already ranked within the top 10 by the respective baseline techniques. That is, for a small set of configurations that perform well across baselines, augmenting TR-based bug localization techniques with GUI interaction information provides a largely *complementary* improvement in effectiveness.

4.1 RQ$_1$: Impact of Number of Screens

The number of configurations across different numbers of screens that exhibit *positive* % improvement over baselines are shown in Table 5. Table 6 reports the average, minimum, and maximum *positive* improvement over each respective baseline technique across *all* studied GUI Information configurations when different numbers of

Table 5: Number of configurations exhibiting positive % improvement in Hits@10 over baselines across # of screens.

Approach	2 Screens	3 Screens	4 Screens	# Screens
BugLocator	90	115	110	315
SentenceBERT	61	69	75	205
UnixCoder	101	114	97	312
Lucene	80	86	82	248
Total	**332**	**384**	**364**	**1,080**

Table 6: Average positive % improvement of Hits@10 over baselines across the number of screens.

Approach	2 Screens			3 Screens			4 Screens		
	min	**avg**	**max**	**min**	**avg**	**max**	**min**	**avg**	**max**
BugLocator	1.75	**7.86**	17.54	1.75	7.78	17.54	1.75	7.74	17.54
SentenceBERT	1.72	5.51	12.07	1.72	8.05	15.52	1.72	**8.11**	15.52
UnixCoder	1.79	7.28	14.29	1.79	**7.69**	12.50	1.79	6.76	14.29
Lucene	1.56	5.55	9.37	1.56	5.65	10.94	1.56	**5.79**	12.50
Overall	1.56	6.55	9.37	1.56	**7.29**	10.94	1.56	7.10	12.50

screens from the bug reproduction scenario are used. Given that we want to understand which screen configuration provides the best *improvement*, here we do not discuss configurations that did not improve over the baseline. The highest *average* improvement for each baseline technique is shown in bold. From the table we can observe that there is no one configuration for number of screens that performs best across all techniques. However, considering average overall improvement, using information from 3 screens (*i.e.,* the buggy screen and two prior screens) provides the highest overall improvement over the baseline techniques, whereas using information from 4 screens provides the largest improvement for two techniques (Lucene and SentenceBERT). It should be noted that 4 screens leads better performance in terms of the *max improvement* of a single configuration over the baselines. However, the average values correspond to the highest increase across *all* studied configurations.

Summary of Findings for RQ$_1$: We find that using GUI information from the buggy screen, and two preceding screens provides the highest overall increase in effectiveness across our studied baseline TR-based bug localization techniques. This indicates that relevant GUI information for bug localization is contained in not only in the buggy screen, but also in the preceding screens.

4.2 RQ$_2$: Impact of GUI Information Type & Augmentation Method

In RQ$_2$, we first aim to investigate the impact of using GUI-related files sourced from different types and combinations of GUI-Interaction Information (*e.g.,* GUI Screens (**`GS`**), Exercised GUI Components (**`ECG`**), **`GS+ECG`**, Screen Components (**`SC`**), and **`GS+SC`**) on our augmentation methods. To do this, in Table 7, which is organized by augmentation method, we report the best performing GUI Interaction Information type for each of our five augmentation methods and for each baseline technique. This allows us to examine whether there are trends in the best performing information types across our studied techniques. Dashes signify that no configuration of the reported augmentation method improved over the baseline. Table 7 illustrates a few notable trends across the different augmentation techniques. For instance, we can observe that for *Filtering*,

Table 7: Best performing GUI Information configurations according to % relative improvement for Hits@10.

Approach	Augmentation	Information Type	# Screens	HIT@10 Improvement
BugLocator	Filtering	SC	3	1.75
	Boosting	GS	4	12.28
	Filtering+Boosting	SC(F)+GS(B)	4	14.04
	Query Expansion	SC	2	8.77
	Query Replacement	GS+SC	2	1.75
SentenceBERT	Filtering	SC	2	5.17
	Boosting	GS+EGC	3	15.52
	Filtering+Boosting	SC(F)+[GS+EGC(B)]	3	15.52
	Query Expansion	—	—	—
	Query Replacement	—	—	—
UniXCoder	Filtering	—	—	—
	Boosting	GS	4	12.50
	Filtering+Boosting	SC(F)+GS(B)	4	12.50
	Query Expansion	SC	2	14.29
	Query Replacement	GS+SC	3	8.93
Lucene	Filtering	SC	3	4.69
	Boosting	GS	4	9.38
	Filtering+Boosting	SC(F)+GS(B)	4	12.50
	Query Expansion	GS	4	6.25
	Query Replacement	—	—	—

`SC` (which map to the largest number of GUI-related files) is always the best performing information type (except for UniXCoder). This is not entirely surprising as, due to the large number of associated GUI-related files, it is the least restrictive filtering.

For *Boosting*, `GS` (or `GS+ECG`) performs best across all baseline techniques. Given that `GS` information targets a smaller number of GUI-related files that often encompass large portions of screen functionality, it follows that boosting such targeted information is likely to have a larger positive effect on ranking buggy files. These same trends also hold for *Filtering+Boosting*, wherein filtering with GUI Interaction Information that is mapped to a larger number of GUI-related files (*e.g.,* `SC`) and boosting with a more targeted information type that maps to fewer files (*e.g.,* `GS`, `ECG`) generally leads to the best results. Finally, we find that *Query Replacement* generally does not lead to an improvement over the baseline, but Query Expansion does lead to improvements – however, there is no clear trend in the best performing Information type for expansion.

Turning our attention to the impact of different augmentation methods we observe further confirmation of trends that began to surface when examining the impact of different GUI-Interaction Information types. First, we examine the reformulation and reranking methods in isolation to better understand their effects. Figure 3 shows a box-and-whisker plot illustrating the relative improvement that both query expansion and replacement have over our studied baseline techniques. The number of configurations for each augmentation method are given above each plot. This plot shows that, in general, Query Replacement rarely results in any improvement over the baseline technique. On the contrary, query expansion does generally result in improvement for every baseline except SentenceBERT. This is likely due to the abstract semantic embeddings that are produced by SentenceBERT which may be able to more naturally retrieve GUI-related information due to its learned, rich term similarities. Furthermore, query expansion seems to have the largest positive effect on UniXcoder, which is another neural model trained on code as opposed to natural language. We speculate that this improvement may be due to the fact that embedding file names into UniXCoder's code-specific embedding space makes it far more likely for those files to be retrieved as compared to SentenceBERT. The observation that query replacement tends to perform poorly indicates that there is often important lexical

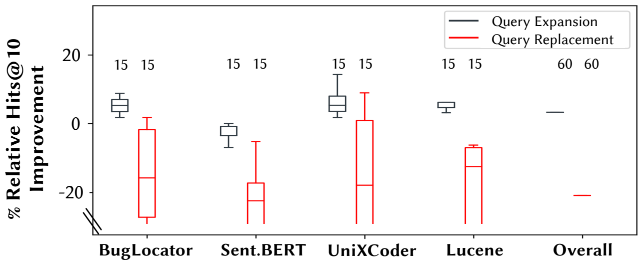


Figure 3: Relative % Improvement of Query Reformulation

information contained within bug reports for TR-based localization techniques, as removing this information and replacing it with GUI-related file names generally degrades performance (sometimes markedly so). Conversely, *expanding* the bug report query with GUI-related terms does appreciably improve most techniques, signaling that expanding queries with GUI-related file names is helpful.

Figure 4 illustrates the same box and whisker plot, but for our three re-ranking methods. Form this plot we can observe that only a small number of filtering configurations improve upon the baseline techniques. This follows from our findings in RQ$_2$ which illustrate that only filtering configurations with a GUI information type that maps to a large number of GUI-related files perform well, as this limits the overall number of files filtered out of the searchable corpus. However, while filtering only sometimes leads to improvements, nearly every configuration of Boosting leads to appreciable improvements over the baseline techniques. Combining Filtering and Boosting together leads to the highest overall median improvements, but with more variability in the results due to poor performance of filtering with certain information types. The results related to Boosting illustrate that re-ranking GUI-related files to the top of list of retrieved results is nearly always beneficial, which indicates that these files have a far higher probability of being buggy compared to other files that were not linked to GUI information.

Summary of Findings for RQ$_2$: We find that for Filtering, SC information leads to the largest improvement, and for Boosting, GS information leads to the largest improvement. Overall, filtering with information types which map to a higher number of GUI-related files (*i.e.,* filtering out fewer files), and boosting with information types that map to a lower number of files, tend to perform best. Filtering only provides benefit in a small number of cases with limited GUI Information types, whereas boosting is nearly always beneficial. We also observe that Query Replacement rarely leads to performance improvements, whereas Query Expansion does lead to improvement for every baseline except SentenceBERT, albeit without any single GUI Information type performing best. Combining Filtering and Boosting together leads to the highest overall improvements, but these configurations are more sensitive to the information types used.

4.3 RQ$_3$: Best Performing Configurations

RQ$_3$ aims to take a deeper look at the best performing individual GUI-augmentation configurations for each baseline technique. We report this information in Table 8 where the best performing configurations were selected by taking the top performing techniques according to Hits@10, and breaking ties according to Hits@5. If there were still ties after considering Hits@5, then we report all

Table 8: Best Performing Combinations

Baseline/ Config	Filtering GUI Info	Boosting GUI Info	GUI Info Query Exp.	# Scrns	H@1	H@5	#Bugs Top10 (H@10)
BugLocator	None	None	None		0.39	0.60	57 (0.71)
	SC	GS	GS+SC	3	0.33	0.76	67 (0.84)
SentenceBERT	None	None	None		0.23	0.56	58 (0.72)
	SC	GS+EGC	None	3	0.30	0.72	67 (0.84)
	GS+SC	GS+EGC	None	3	0.30	0.72	67 (0.84)
	SC	GS+EGC	EGC	3	0.30	0.72	67 (0.84)
	GS+SC	GS+EGC	EGC	3	0.30	0.72	67 (0.84)
UnixCoder	None	None	None		0.14	0.62	56 (0.70)
	SC	GS	SC	4	0.31	0.75	64 (0.80)
	GS+SC	GS	SC	4	0.31	0.75	64 (0.80)
Lucene	None	None	None		0.40	0.75	64 (0.80)
	SC	GS	None	4	0.36	0.80	72 (0.90)
	GS+SC	GS	None	4	0.36	0.80	72 (0.90)

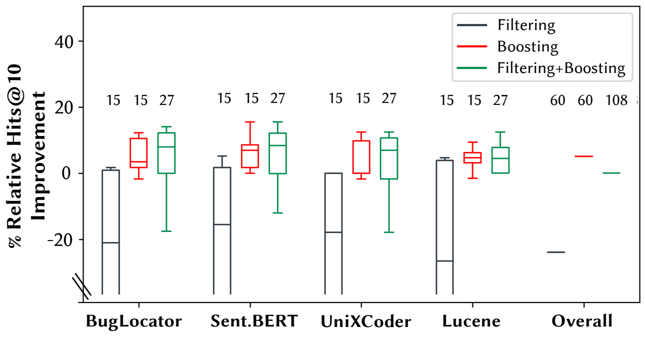


Figure 4: Relative % Improvement of Re-Ranking

such configurations. The first major observation that can be made from Table 8 is that BugLocator and Lucene, which use more traditional text-retrieval baselines (*i.e.,*tf-idf document representations) tend outperform the techniques that use neural embeddings (SentenceBERT and UniXCoder). It should be noted that, given the goal of our study is to examine the benefit of GUI interaction information, we used SentenceBERT and UniXCoder in a zero-shot setting, wherein they were not fine-tuned on bug report information. Future work may examine these in a fine-tuned setting.

The next major trend illustrated in Table 8 is that the best performing configurations offer a marked improvement (in terms of Hits@5 and Hits@1-) compared to the baseline techniques, with relative improvements ranging from 12.5%-18% for Hits@10, and 6%-29% for Hits@5. This means that, for all baselines, the best performing configurations result in the inclusion of buggy files for an additional 9-10 bugs in the top 10 results. Furthermore, we find that the best performing configurations of our GUI-augmentation methods are strikingly consistent. That is, Filtering+Boosting is always the best re-ranking technique, using SC or GS+SC GUI information for Filtering, and either GS or GS+EGC information for boosting always leads to the largest improvements over the baseline techniques. This means that any one of these configurations can likely be applied to a TR-based baseline and improve the results. We find that certain baselines and configurations seem benefit from query expansion with varying GUI information types. This is consistent with the observations from the previous RQ. Finally, we performed a Wilcoxon signed rank test on the first rank of a buggy file for all bugs (even those ranked outside top-10) across all techniques using a 95% confidence interval. We found that all of the best performing GUI-augmentation methods outperform their

Table 9: In-Depth Analysis of Best Performing overall GUI-related augmentation configuration.

Baseline/ Config	Filtering GUI Info	Boosting GUI Info	Number of Screens	Hits@10	# Bugs Out10 → In10	# Bugs In10 → Out10	# Bugs Inside 10 Improved	# Bugs Inside 10 Deteriorated	# Bugs Inside 10 Unchanged	# Bugs Outside 10 Improved	# Bugs Outside 10 Deteriortated
BugLocator	SC	GS	4	0.81	8	0	16	21	20	10	4
SentenceBERT	SC	GS	4	0.81	7	0	22	15	21	12	1
UniXCoder	SC	GS	4	0.79	7	0	21	13	22	15	0
Lucene	SC	GS	4	0.90	8	0	16	24	24	6	1

corresponding baseline techniques to a statistically significant degree, save for BugLocator, which had a p-value of 0.44. However, given that past work has illustrated that TR-based bug localization typically only provides benefits to developers if the buggy files is ranked within the top-10 results [78], we must note that BugLocator saw the benefit of the largest overall increase in Hits@10 from our GUI-augmentation methods across all of our studied baselines.

Summary of Findings for RQ$_3$: We find that the best performing configurations of our GUI-augmentation methods combine filtering and boosting, using `SC` or `SC+GS` to filter, and `GS` or `GS+EGC` to boost, and lead to an improvement in Hits@10 ranging from 12.5%-18%, and 6%-29% for Hits@5.

4.4 Discussion

We aim to see whether a single GUI augmentation configuration can perform well across all baselines. To this end, we first ranked our 657 configurations for each baseline in terms of their performance in Hits@10, using the average performance of HITS@1 and HITS@5 to break ties. We found two configurations that perform identically, and chose the simpler configuration of the two to analyze. This configuration uses the Filtering+Boosting with `SC` information for Filtering and `GS` information for boosting, using data from 4 screens.

In general, an augmentation affects the rankings of most of the bugs, however we are less concerned with those that are far from top-10 (they are still hard to retrieve), or those already in top-10 (they are still easy to retrieve). We consider an augmentation to be good if it ranks more bugs in top-10 than its baseline (*i.e.,* at least one of their buggy files rank in top-10), in other words bugs that were hard to retrieve with the baseline are now easy to retrieve with the augmentation. With that in mind, for the GUI augmentation method configuration we identify as best, we examined how it performed in terms of moving bugs from outside the top 10 ranks to inside the top 10 ranks (and vice versa) as well as the number of buggy file that improved, degraded and remained unchanged both inside and outside the top 10 ranks – we report these results in Table 9. This table illustrates that the best performing GUI-augmentation configuration brings buggy files for 7-8 bugs from outside the top-10 to inside the top-10 without moving any buggy files that were previously in the top 10 outside of the top 10. This signals that the GUI-related augmentations are largely complementary to existing techniques and do not hurt existing bugs ranked within the top-10. When examining how ranks of buggy files change within the top 10 ranks, we find a relatively even mix of files that improved (16-22), deteriorated (13-24), and remained unchanged (20-24). This signals that there is limited net impact on the ranks of the files within the top-10 ranks. Furthermore, we find that this configuration improves more buggy files outside the top-10 than it deteriorates, and our studied configuration outperform **all** baselines in terms of top ranks of buggy files according to a Wilcoxon signed rank test at a 95% confidence interval.

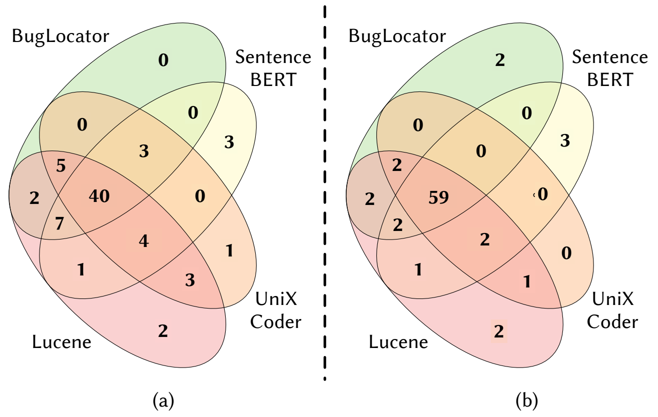


Figure 5: Venn Diagram illustrating the overlap where buggy files appear in the top-10 results for our studied baselines (a) and for the best performing GUI augmentation methods (b).

In addition, we wanted to understand the degree of orthogonality, in terms of Hits@10, across our baseline techniques, and how our GUI augmentation methods effect this orthogonality. In Figure 5-a, we provide a Venn Diagram that includes each baseline, where overlapping sections indicate a shared bug for which a buggy file was ranked within the top-10. From this diagram, we can observe that between 61% and 70% of bugs with buggy files ranked within the top 10 are shared across all baselines (40 bugs), indicating that the different approaches do exhibit some degree of orthogonality, with Lucene having the highest amount of orthogonality in comparison to the other techniques. However, taking the best performing GUI-augmented configuration for each baseline decreases the overall amount of orthogonality, as between 92% and 82% of bugs with buggy files ranked within the top-10 are shared. *This indicates that even when we take baseline techniques that operate using different text representations and exhibit orthogonality, our GUI-augmentation methods are able to improve upon all of them despite their differences.*

5 THREATS TO VALIDITY

Construct Validity: Our study's main threat to construct validity is rooted in the potential subjectivity of deriving our manually labeled bug localization dataset. To minimize this threat, we performed a rigorous manual labeling procedure (see Sec. 3.3.2), wherein two authors independently coded the buggy files for each bug report, and conflicts were resolved via discussion with a third participant. All authors involved in this process were experienced in Android development and achieved a high inter-coder agreement (88.33%).
Internal Validity: Threats to internal validity of our study conclusions stem from our selected baseline TR-based bug localization techniques. We observed a variation in effectiveness across our four studied TR-based localization baselines, however, despite this differing performance, the main commonality in our experiments

was the introduction of our GUI-based augmentation techniques, which were the main factor in the improvements we observed.

Conclusion Validity: To mitigate threats to conclusions, we made use of the common Hits@K metric, focusing our analysis on Hits@10. This choice was motivated by prior work that illustrated that automated bug localization techniques are generally only helpful to developers when they return relevant buggy files in the top 10 results [78]. Furthermore, in addition to examining effectiveness in terms of Hits@10, we also performed a fine-grained analysis into the impact that the GUI-based augmentation techniques have on the number of buggy files that get moved into/out of the top 10.

External Validity: To mitigate threats to external validity we studied 2,628 combinations of our GUI-based augmentation methods, for 80 bug reports from 39 diverse, and popular applications. Given the effort required to build our dataset, the amount of data we analyze is sizable, but a larger set of reports would provide additional confidence in our results. We analyzed four TR-based bug localization techniques, and our results may not generalize beyond these.

6 RELATED WORK

Text-Retrieval-based Bug Localization (TRBL) Techniques. A TRBL retrieval engine, powered by a TR technique, typically leverages the textual similarity between the code and the bug report to the determine the relevance of a code artifact to a query (less/-more likely to be buggy). The relevance score is determined using different techniques and representations of artifacts and reports.

Researchers have proposed a variety of TRBL techniques over the past two decades [12], spanning Information Retrieval (IR) techniques [34, 46, 47, 50, 55, 63, 67, 67] and, more recently, Deep Learning (DL)-based approaches [24, 30, 40, 77, 87, 88, 91]. Common IR techniques include classical algorithms such the Unigram Model (UM) [67], Cluster Based Document Model (CBDM) [67], Vector Space Model (VSM) [92, 99], Latent Semantic Indexing (LSI) [55], and Latent Dirichlet Allocation (LDA) [53]. DL-based approaches include models such as CNNs [40, 77, 87, 88], RNNs [30, 91], Transformers [24] and combinations of these [86, 89, 90]. Other approaches combine IR and DL techniques [48]. The advantage of IR techniques is their simplicity and efficiency, since they do not require training. Conversely, DL-based approaches are more expensive as they require training using a large amount of labeled data. The advantage of DL techniques is they can generate more informative document representations. To assess the effect of a technique on TRBL when leveraging GUI interaction data, we used two IR-based techniques, namely BugLocator [26, 27] and Lucene [2], and two DL models, namely SentenceBERT [69] and UniXCoder [36].

Information Sources for TRBL. To address the lexical gap between bug reports and source code and produce more accurate suggestions of buggy artifacts [15, 56, 93], researchers have utilized additional information related to the bug reports and code. The most prominent sources of information used by existing approaches include similar bug reports [68, 85], code structure [75, 80, 94], code version history [79, 80, 94, 95], stack traces [60, 82, 94], part-of-speech information [100], and combinations of the above [71, 85].

Particularly relevant is the work that leverages system execution traces for TRBL [60], however, collecting these traces are typically expensive since they require program instrumentation or a suite of test cases with high-coverage [49]. We study how GUI interaction data of Android apps can be used to help bridge the lexical gap between bug reports and code, and thus help retrieve relevant buggy files more effectively. To the best or our knowledge, we are the first to explore the use of such data for TRBL. This data can be easily collected via the Android infrastructure without requiring app instrumentation [9], when the apps are used or the bugs are reproduced, manually or via automated execution tools [58].

Re-ranking Methods for TRBL. Prior work has used the aforementioned sources of information to produce a ranking of code artifacts or to re-rank them based on an existing ranking. The goal is to rank the buggy artifacts higher than non-buggy artifacts. We identify at least three general re-ranking methods used in the literature: *boosting*, *filtering*, and *query (re)formulation*. Prior techniques have *boosted* the similarity/relevance score of code artifacts to improve their ranking [52, 70, 85]. Other techniques *filter* out irrelevant code artifacts from the artifact search space or initial ranking [51]. Query (re)formulation aims to encode relevant textual information in the query for improving retrieval [19–21, 23, 33, 56, 64]. Researchers focused on three main reformulation methods: query expansion [17, 43] (which adds extra terms to the query), query replacement [35, 37] (which substitutes the query with a new one), and query reduction [23, 33, 65, 66] (which identifies/removes terms in the query that hinder retrieval). Inspired by this work, we assess the effect of using GUI interaction information on TRBL performance via boosting, filtering, query expansion, and query replacement. Our future work will explore how query reduction along with GUI information can be combined to improve TRBL performance.

Software Types for TRBL. The vast majority of prior work has proposed system-agnostic TRBL approaches [53, 62, 99]. This means they were designed to operate on any kind of software system (*e.g.,* desktop applications, libraries, and command line projects) and any bug report, without differentiating the bug type. Recent work has target deep learning projects [44]. Our study focuses on Android apps and GUI-based bugs, which represents the majority of bug types found in the Android ecosystem [16, 73, 74].

7 CONCLUSION & FUTURE WORK

We reported an empirical study that found a high positive effect of mobile app GUI interaction data on text-retrieval-based bug localization, using a manually-constructed bug localization dataset consisting of 80 bug reports with GUI metadata and four bug localization baseline techniques. The measured effect indicates GUI interaction data can help bridge the lexical gap between bug reports and source code, resulting in better rankings of buggy files suggested to developers. Our future work will investigate other ways of augmenting existing localization techniques via GUI interaction data for mobile apps and other types of systems and conducting human studies that aim to validate the benefits of using GUI interaction information for bug localization in practice.

ACKNOWLEDGEMENTS

This work is supported by the U.S. NSF under grants: CCF-1910976, CCF-2343057, CCF-1955837, CCF-2007246, and CCF-1955853. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] 2023. Android UIAutomator- https://developer.android.com/training/testing/other-components/ui-automator.

[2] 2023. Apache Lucene- https://lucene.apache.org.

[3] 2023. Create a fragment - https://developer.android.com/guide/fragments/create.

[4] 2023. HuggingFace - https://huggingface.co/.

[5] 2023. Introduction to activities - https://developer.android.com/guide/components/activities/intro-activities.

[6] 2023. Markor - https://f-droid.org/packages/net.gsantner.markor/.

[7] 2023. Markor - https://github.com/gsantner/markor/issues/1020/.

[8] 2023. sentence-transformers/msmarco-distilbert-base-v3 - https://huggingface.co/sentence-transformers/msmarco-distilbert-base-v3.

[9] 2023. System Tracing - https://developer.android.com/topic/performance/tracing.

[10] 2023. UI-Bug-Localization- https://github.com/SageSELab/UI-Bug-Localization-Study.

[11] 2023. UniXcoder - https://github.com/microsoft/CodeBERT/tree/master/UniXcoder#2-similarity-between-code-and-nl.

[12] Shayan A Akbar and Avinash C Kak. 2020. A large-scale comparative evaluation of IR-based tools for bug localization. In *MSR'20*. 21–31.

[13] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. 2023. Translating Video Recordings of Complex Mobile App UI Gestures into Replayable Scenarios. *TSE* 49, 4 (2023), 1782–1803.

[14] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *ICSE'20*. 309–321.

[15] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *SIGSOFT'08/FSE-16*. 308–318.

[16] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *CSMR'13*. 133–143.

[17] Claudio Carpineto and Giovanni Romano. 2012. A survey of automatic query expansion in information retrieval. *CSUR* 44, 1 (2012), 1–50.

[18] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the Quality of the Steps to Reproduce in Bug Reports. In *ESEC/FSE'19*. 86–96.

[19] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2017. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *ICSME'17*. 376–387.

[20] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2019. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *EMSE* 24 (2019), 2947–3007.

[21] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. 2019. Reformulating queries for duplicate bug report detection. In *SANER'19*. 218–229.

[22] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *ESEC/FSE'17*. 396–407.

[23] Oscar Chaparro and Andrian Marcus. 2016. On the reduction of verbose queries in text retrieval based software maintenance. In *ICSE'16-C*. 716–718.

[24] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast changeset-based bug localization with BERT. In *ICSE'22*. 946–957.

[25] Nathan Cooper, Carlos Bernal-C'ardenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. In *ICSE'21*. 957–969.

[26] Steven Davies and Marc Roper. 2013. Bug localisation through diverse sources of information. In *ISSREW'13*. 126–131.

[27] Steven Davies, Marc Roper, and Murray Wood. 2012. Using bug report similarity to enhance bug localisation. In *WCRE'12*. 125–134.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL'19*.

[29] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *JSEP* 25, 1 (2013), 53–95.

[30] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. 2021. On the classification of bug reports to improve bug localization. *Soft Computing* 25 (2021), 7307–7323.

[31] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *ISSTA'18*. 141–152.

[32] Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. In *ICSE'22*. 1045–1057.

[33] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. 2021. Combining query reduction and expansion for text-retrieval-based bug localization. In *SANER'21*. 166–176.

[34] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 2009. On the use of relevance feedback in IR-based concept location. In *ICSM'09*. 351–360.

[35] Marek Gibiec, Adam Czauderna, and Jane Cleland-Huang. 2010. Towards mining replacement queries for hard-to-retrieve traces. In *ASE'10*. 245–254.

[36] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *ACL'22* (2022).

[37] Jin Guo, Marek Gibiec, and Jane Cleland-Huang. 2017. Tackling the term-mismatch problem in automated trace retrieval. *EMSE* 22 (2017), 1103–1142.

[38] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvanyk, and Kevin Moran. 2021. V2S: A Tool for Translating Video Recordings of Mobile App Usages into Replayable Scenarios. *ICSE'21-C* (2021), 65–68.

[39] Emily Hill, Zachary P Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K Vijay-Shanker. 2008. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR'08*. 79–88.

[40] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2019. Deep Transfer Bug Localization. *TSE* PP (06 2019), 1–1.

[41] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436

[42] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. *SANER'22* (2022).

[43] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. 2021. A Novel Automatic Query Expansion with Word Embedding for IR-based Bug Localization. In *ISSRE'21*. 276–287.

[44] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. 2022. An Empirical Study of IR-based Bug Localization for Deep Learning-based Software. In *ICST'22*. 128–139.

[45] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).

[46] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Monperrus Martin, and Yves Le Traon. 2019. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. *ArXiv* abs/1902.02703 (2019).

[47] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. 2007. Semantic clustering: Identifying topics in source code. *IST* 49, 3 (2007), 230–243.

[48] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *ICPC'17*. 218–229.

[49] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *ESEC/FSE'15*. 579–590.

[50] Wei Li, Qingan Li, Yunlong Ming, Weijiao Dai, Shi Ying, and Mengting Yuan. 2022. An empirical study of the effectiveness of IR-based bug localization for large-scale industrial projects. *EMSE* 27, 2 (2022), 47.

[51] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. 2007. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE'07*. 234–243.

[52] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *ESEC/FSE'21*. 664–676.

[53] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2008. Source code retrieval for bug localization using latent dirichlet allocation. In *WCRE'08*. 155–164.

[54] Andrian Marcus and Jonathan I Maletic. 2001. Identification of high-level concept clones in source code. In *ASE'01*. 107–114.

[55] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic. 2004. An information retrieval approach to concept location in source code. In *WCRE'04*. 214–223.

[56] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. 2020. On the relationship between bug reports and queries for text retrieval-based bug localization. *EMSE* 25 (2020), 3086–3127.

[57] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing Bug Reports for Android Applications. In *(FSE'15)*. 673–686.

[58] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *ICST'16*. 33–44.

[59] Kevin Moran, Ali Yachnes, George Purnell, Junayed Mahmud, Michele Tufano, Carlos Bernal Cardenas, Denys Poshyvanyk, and Zach H'Doubler. 2022. An Empirical Investigation into the Use of Image Captioning for Automated Software Documentation. In *SANER'22*. 514–525.

[60] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *ICSME'14*. 151–160.

[61] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2013. The design of bug fixes. In *ICSE'13*. 332–341.
[62] Denys Poshyvanyk. 2009. Using information retrieval to support software maintenance tasks. In *ICSME'09*. 453–456.
[63] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *TSE* 33, 6 (2007), 420–432.
[64] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K Roy. 2021. The forgotten role of search queries in ir-based bug localization: an empirical study. *EMSE* 26, 6 (2021), 116.
[65] Mohammad Masudur Rahman and Chanchal K Roy. 2017. Improved query reformulation for concept location using coderank and document structures. In *ASE'17*. IEEE, 428–439.
[66] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-Based Bug Localization with Context-Aware Query Reformulation. In *ESEC/FSE'18*. 621–632.
[67] Shivani Rao and Avinash Kak. 2011. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *MSR'11*. 43–52.
[68] Michael Rath, David Lo, and Patrick Mäder. 2018. Analyzing requirements and traceability information to improve bug localization. In *MSR'18*. 442–453.
[69] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *EMNLP'19* (2019).
[70] Joseph John Rocchio Jr. 1971. Relevance feedback in information retrieval. *The SMART retrieval system: experiments in automatic document processing* (1971).
[71] Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, and Xingjun Zhang. 2018. Comparing learning to rank techniques in hybrid bug localization. *Applied Soft Computing* 62 (2018), 636–648.
[72] Yang Song and Oscar Chaparro. 2020. Bee: A tool for structuring and analyzing bug reports. In *ESEC/FSE'20*. 1551–1555.
[73] Yang Song, Junayed Mahmud, Nadeeshan De Silva, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2023. BURT: A Chatbot for Interactive Bug Reporting. In *ICSE'23*. 170–174.
[74] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (android app) end-users. In *ESEC/FSE'22*. 344–356.
[75] Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. 2018. A preliminary study on using code smells to improve bug localization. In *ICPC'18*. 324–327.
[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
[77] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. 2020. Multi-dimension convolutional neural network for bug localization. *TSC* 15, 3 (2020), 1649–1663.
[78] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-Based Fault Localization Techniques. In *ISSTA'15*. 1–11.
[79] Shaowei Wang and David Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *ICPC'14*. 53–63.
[80] Shaowei Wang and David Lo. 2016. Amalgam+: Composing rich information sources for accurate bug localization. *JSEP* 28, 10 (2016), 921–942.
[81] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug?. In *MSR'07: ICSE Workshops 2007*. 1–1.
[82] M. Wen, R. Wu, and S. Cheung. 2016. Locus: Locating bugs from software changes. In *ASE'16*. 262–273.
[83] Tyler Wendland, Jingyang Sun, Junayed Mahmud, Syeda Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A Dataset of Manually-Reproduced Bug Reports for Android apps. In *MSR'21*. 600–604.
[84] Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *NAACL'18*. 1112–1122.
[85] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *ICSME'14*. 181–190.
[86] Yan Xiao and Jacky Keung. 2018. Improving bug localization with character-level convolutional neural network and recurrent neural network. In *APSEC'18*. 703–704.
[87] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *IST* 105 (2019), 17–29.
[88] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E Bennin. 2017. Improving bug localization with an enhanced convolutional neural network. In *APSEC'17*. 338–347.
[89] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E Bennin. 2018. Bug localization with semantic and structural features using convolutional neural network and cascade forest. In *EASE'18*. 101–111.
[90] Geunseok Yang and Byungjeong Lee. 2021. Utilizing topic-based similar commit information and CNN-LSTM algorithm for bug localization. *Symmetry* 13, 3 (2021), 406.
[91] Shouliang Yang, Junming Cao, Hushuang Zeng, Beijun Shen, and Hao Zhong. 2021. Locating faulty methods with a mixed RNN and attention model. In *ICPC'21*.
[92] Zhou Yang, Jieke Shi, Shaowei Wang, and David Lo. 2021. IncBL: Incremental Bug Localization. In *ASE'21*. 1223–1226.
[93] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *FSE'14*. 689–699.
[94] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *IST* 82 (2017), 177–192.
[95] Tao Zhang, Wenjun Hu, Xiapu Luo, and Xiaobo Ma. 2019. A commit messages-based bug localization for android applications. *IJSEKE* 29, 04 (2019), 457–487.
[96] Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William GJ Halfond. 2023. Automatically Reproducing Android Bug Reports Using Natural Language Processing and Reinforcement Learning. In *ISSTA'23*.
[97] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *TOSEM* 31, 3 (2022), 1–33.
[98] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *ICSE'19*. 128–139.
[99] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *ICSE'12*. 14–24.
[100] Yu Zhou, Yanxiang Tong, Taolue Chen, and Jin Han. 2017. Augmenting bug localization with part-of-speech and invocation. *IJSEKE* 27, 06 (2017), 925–949.
[101] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *TSE* 36, 5 (2010), 618–643.