2022

Changeset-based Retrieval of Source Code Artifacts for Bug Localization

Agnieszka Ciborowska
Virginia Commonwealth University

CHANGESET-BASED RETRIEVAL OF SOURCE CODE ARTIFACTS FOR BUG LOCALIZATION

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

AGNIESZKA CIBOROWSKA

Ph.D. Candidate

Director: Kostadin Damevski,

Associate Professor, Department of Computer Science

Virginia Commonwealth University

Virginia Commonwealth University

Richmond, Virginia

April, 2022

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Kostadin Damevski, who has inspired me to be a better researcher, and, more importantly, to be a better person. His patient guidance and constant encouragement have made this an insightful and rewarding journey.

This dissertation would have never happened without the long-distance support of my family. Warmest thanks to my parents, whose love and guidance are with me whatever I pursue. I also want to thank my sister for showing me how to dream and supporting me along the way.

Thanks to all my friends, both in Poland and in the USA. Thank you in particular to Adrian Malinowski and Malgorzata Machon, who are my lifelong friends and always had my back, even from far away. Thank you to Maciej Moscinski, Dominika and Michal Polanscy, Magdalena and Mateusz Koguc, and Damian Wiecek for making Poland feels like I have never left.

Finally, thank you to all my Richmond friends, especially to Hannaneh Barahouei Pasandi, Jorge Gonzalez Lopez, and Andriy Mulyar. Warm thanks to Adam Ziemba and Alberto Cano, who became my new small family that I can always count on.

The last words of acknowledgment are saved for my dear husband, Łukasz, for his endless support and patient love. Thank you for believing in me against all odds. I am blessed to have you in my life.

Table of Contents

List of Tables

List of Figures

Abstract

Modern software development is extremely collaborative and agile, with unprecedented speed and scale of activity. Popular trends like continuous delivery and continuous deployment aim at building, fixing, and releasing software with greater speed and frequency. Bug localization, which aims to automatically localize bug reports to relevant software artifacts, has the potential to improve software developer efficiency by reducing the time spent on debugging and examining code. To date, this problem has been primarily addressed by applying information retrieval techniques based on static code elements, which are intrinsically unable to reflect how software evolves over time. Furthermore, as prior approaches frequently rely on exact term matching to measure relatedness between a bug report and a software artifact, they are prone to be affected by the lexical gap that exists between natural and programming language.

This thesis explores using software changes (i.e., changesets), instead of static code elements, as the primary data unit to construct an information retrieval model toward bug localization. Changesets, which represent the differences between two consecutive versions of the source code, provide a natural representation of a software change, and allow to capture both the semantics of the source code, and the semantics of the code modification. To bridge the lexical gap between source code and natural language, this thesis investigates using topic modeling and deep learning architectures that enable creating semantically rich data representation with the goal of identifying latent connection between bug reports and source code. To show the feasibility of the proposed approaches, this thesis also investigates practical aspects related to using a bug localization tool, such retrieval delay and training data availability.

The results indicate that the proposed techniques effectively leverage historical data about bugs and their related source code components to improve retrieval accuracy, especially for bug reports that are expressed in natural language, with little to no explicit code references. Further improvement in accuracy is observed when the size of the training dataset is increased through data augmentation and data balancing strategies proposed in this thesis, although depending on the model architecture the magnitude of the improvement varies. In terms of retrieval delay, the results indicate that the proposed deep learning architecture significantly outperforms prior work, and scales up with respect to search space size.

CHAPTER 1

INTRODUCTION

Over the years, software development processes have been modernized to support efficient building and deployment of software systems. Practices such as continuous integration and continuous deployment focus on capturing source code evolution over time and allow to coordinate efforts of multiple developers to instantly deliver software to end users [1, 2, 3]. As a result, modern software development is highly dynamic with extremely high code churn and the number of contributing developers [4, 5]. At the same time, performing common development tasks, such as debugging, maintaining, or updating the code, has become more challenging due to the large volumes of code that have to be analyzed and comprehended by software developers to successfully complete the task at hand.

Locating and fixing software bugs has persisted as one of the most common and important tasks software developers face on a daily basis. In fact, the year 2017 has been marked as "*The year that software bugs ate the world*" [6], which serves as a reminder that in an environment in which software size and complexity grows constantly, and production deadlines are pressing, software bugs are bound to happen, and, in due time, have to be resolved. To fix a bug, a developer first analyzes the bug report looking for hints about bug location and then explores the source code to identify a few potential bug-related code entities for which undesired behavior is later confirmed through debugging [7]. The process of identifying relevant software artifacts, such as classes or methods, given a bug report is typically referred to as bug localization [8, 9].

Supporting developers during bug localization is one of the long-standing goals in the software engineering research community, due to its potential to improve practice by reducing the time developers spend examining code when addressing a newly reported bug. Automatically linking a bug report to its most relevant software artifacts is predominantly performed using Information Retrieval (IR) methods, where the bug report text is used to formulate a query that is matched to a corpus of code elements, i.e. classes or methods. The retrieved software artifacts are subsequently ranked according to their relatedness to the bug report.

Unfortunately, despite numerous efforts, the accuracy of bug localization approaches is not yet high enough for widespread use, especially as it applies to different software projects that vary in bug report and code style [10, 11]. Recently, researchers observed that bug localization techniques are strongly influenced by the types of bug reports in the project [12, 13]. Bug reports differ based on how they describe the software failure, e.g., when bug reports are written by expert developers they tend to include detailed information about the source of the bug, such as stack traces or even direct references to relevant source code artifacts [10], while bug reports created by end-users that are unfamiliar with the source code, typically provide only high-level description of the observed faulty behavior [14]. For this second category of bug reports more sophisticated bug localization techniques that, e.g., mine revision histories or build higher-level representations of the source code, are required.

Many tools and techniques proposed to date, view the source code as a static set of documents, which do not change over time [15, 16, 17, 18, 19, 20, 21, 22, 23]. However, unlike some natural language corpora, source code is a subject of constant evolution. As a result, techniques based on static data are susceptible to become quickly outdated. In addition, as the software evolution process itself encodes important information about the project, disregarding the notion of change may also

lead to ignoring the key data to locate relevant software artifacts.

To introduce dynamics of software evolution into tools and techniques targeting bug localization, researchers have recently investigated using changesets (or commits) as the main unit to construct retrieval models [24, 25, 26]. Changesets, which represent the differences between two consecutive versions of the source code, are the primary dimension of data created as software evolves. Changesets have a unique property of capturing both the semantics of the source code, expressed in each local change, and the semantics of the change, encoded by the grouping of multiple local changes within a changeset boundary. Moreover, as changesets are committed into a source code repository, they also capture additional important information about a software development process, such as the time and the author of the modification.

The primary goal of this thesis is to study the usage of changesets towards constructing models effective at retrieving source code artifacts (i.e., classes or changesets) related to a newly reported bug. We specifically focus on models that are able to create context-aware data representations to bridge the gap between a programming language, which defines software functionalities in a rigid and highly-structured manner, and a natural language that is used to describe software failures.

1.1 Contributions of the thesis

Investigating contextual models for bug localization. As software evolves rapidly and is actively maintained by multiple developers, different portions of the code base become affected by distinctive identifier naming patterns and conventions, which exacerbate the already existing semantic gap between bug reports and related code elements, posing a significant challenge to bug localization techniques based solely on token similarity [27]. Surveys of practitioners have also indicated that bug reports that explicitly mention the names of classes or methods relevant to the bug

fix do not require automated bug localization while assisting in localizing bug reports with large semantic gaps with the code base is likely more valuable to developers [28].

To bridge this gap, this work explores using probabilistic and deep learning models to build semantically rich representations of bug reports and source code artifacts. More specifically, first, we focus on using a topic model that leverages the history of previously fixed bug reports to improve the accuracy of the model. Next, encouraged by recently observed improvement in NLP domain fueled by deep learning approaches, we move toward a deep learning architecture to explore different strategies to encode changesets semantics.

Addressing different types of bug reports. Bug reports may contain a variety of content relevant to a specific bug, including description of observed abnormal behavior, relevant class names or components (when known to the reporter), and observed stack traces if the bug resulted in a failure. Depending on the content present in a bug report, different strategies can be employed to best utilize provided information, leading to increased retrieval accuracy.

In this work, we predominantly focus on improving retrieval for bug reports which lack code references or other localization hints, hence are typically the most challenging to locate from the perspective of software developers. To this end, we leverage contextual models trained on historical data to capture correlation between natural language and the code base, and propose an adaptive approach that dynamically adjusts the prediction based on the character of a bug report.

Efficient approaches to bug localization. There are two main disadvantages of using IR-based models for bug localization. First, IR models typically do not support adding new information to the already trained model, hence, they require periodical re-training, which can be expensive and time-consuming even on the latest hardware.

Secondly, as IR-based models require performing pairwise comparison between a bug report and *all* software artifacts, the cost of applying such techniques for large software projects may be prohibitively high due to retrieval delay.

To address the first problem, we propose an approach leveraging an online variant of the probabilistic topic model, Online Latent Dirichlet Allocation, which provides an update procedure that allows introducing new data as they arrive. For the second issue, we explore a recently proposed late interaction deep learning architecture that delays interaction between a bug report and a software artifact (i.e. similarity computation), which, in turn, enables efficient retrieval through an approximated search strategy.

Data augmentation for bug localization. While deep learning models have shown great potential towards bug localization, they are often hamstrung by insufficient training data, particularly in the context of project-specific data. Training a deep learning model for bug localization requires a substantial training set of fixed bug reports, which are at a limited quantity even in popular and actively developed software projects.

To address that, this work explores using synthetic training data on transformer-based deep learning models for bug localization. To generate high-quality synthetic data, we propose novel data augmentation operators that act on different constituent components of bug reports. We also describe a data balancing strategy that aims to create a corpus of augmented bug reports that better reflects the entire source code base.

1.2 Structure of the thesis

The organization of this thesis is as follows. Chapter 2 describes how to apply information retrieval toward bug localization, and discusses advantages and challenges

associated with leveraging changesets and addressing different types of bug reports. In Chapter 3, we review the related work and contrast it with the proposed approaches. Chapter 4 introduces JINGO, an Online LDA model build on changesets that leverages historical data and adapts to different types of bug reports. Chapter 5 describes a deep learning model that uses highly efficient retrieval architecture for bug localization. Chapter 6 proposes a novel bug report augmentation strategy that increases the number of training examples to further improve the performance of deep learning models for bug localization. Finally, chapter 8 outlines the future work.

CHAPTER 2

BACKGROUND

This section starts with an introduction to applying IR-based models in the context of bug localization, followed by a description of common metrics used to evaluate the performance of such models. Next, we define and outline the structure of a changeset, and discuss the benefits and challenges associated with using changesets as a primary data unit. Finally, we characterize different types of bug reports that can occur in a software project and examine how they unique properties may affect a bug localization technique.

2.1 Information retrieval-based bug localization

Bug localization is often framed as an Information Retrieval (IR) task: given a text of a bug report (i.e. query) find the most appropriate program elements (i.e. documents). Figure 1 depicts an overview of an IR system towards bug localization, which comprises of two essential steps: building a retrieval model and retrieving software artifacts for a new bug report.

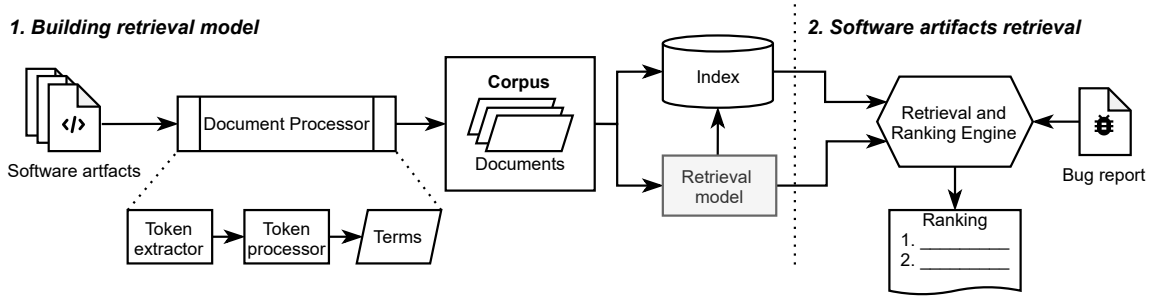


Fig. 1.: Information retrieval system for bug localization.

The first step of building an IR model is collecting a set of input software artifacts. Although the majority of bug localization techniques use classes or methods as the input documents [29, 30, 31, 32, 33], the focus of this work is exclusively on changesets. Hence, the input to the model is a collection of changesets, extracted from a project's repository. Next, the input documents are passed to a document processor which is responsible for creating a corpus from raw data provided on its input (i.e. text of changesets). The process starts with a tokenizer splitting each document into a list of tokens based on whitespace characters. The tokens are then passed to a token processor that applies a set of transformations to each of them, producing a final set of terms that forms a document in a corpus. The following set of transformations is commonly used for processing code-related data [25, 34, 35].

1. **Split:** divide tokens based on non-alphabetical characters or conventions (e.g., "ClassName" becomes "Class" and "Name");
2. **Normalize:** convert all upper case letter to lower case letters;
3. **Filter:** remove common words and stop words, such as "and", "or", "a"; remove programming language keywords, such as "class", "void", "int" (Java);
4. **Stem:** extract the root word of a token by removing prefixes and suffixes (e.g., "writing" becomes "writ", "classes" becomes "class");
5. **Prune:** remove tokens that are too frequent (e.g., more than 80% of documents), or too rare (e.g., less than 5% of documents);
6. **Encode:** replace tokens with unique numeric identifiers, such that the same tokens are encoded with the same identifier.

The resulting corpus of documents is fed into a retrieval model with the objective of building an efficient data representation that groups similar documents together.

After the training is completed, the corpus of documents is encoded by the model and forms an index. Depending on the underlying model, the notion of similarity and how it is computed can vary. Here, we describe three main types of models used for IR-related tasks.

Vector Space Model (VSM). Proposed by Salton et al. [36], VSM is an algebraic model for representing a text document. Based on the corpus of documents, VSM constructs a term-document matrix C of size $N \times D$, where N is the number of unique terms and D stands for the number of documents in the corpus, such that each document from a corpus is represented by a vector of term weights. Computing term weights, given a document in a corpus, is most often based on the classical weighting scheme *term frequency* and *inverse document frequency* (tf-idf), which increases a term weight accordingly to the term frequency in a document, while decreases it proportionally to the term frequency in other documents. Intuitively, the more frequently term i occurs within a document d, and the less frequently it occurs in other documents, the higher the weight of term i in the document d.

Similarity between two documents in VSM model is estimated based on a vector similarity metric, such as cosine similarity. Given the data representation used by VSM, two documents are considered similar if they exhibit a similar frequency of terms, while the context in which the terms occur is ignored, hence all bug localization techniques based on VSM have a limited accuracy ceiling [11].

Topic model. A topic model is a probabilistic model that identifies an abstract set of *topics* (or common themes) in a corpus, such that documents describing similar concepts should be characterized by similar topics. For instance, documents referring to "soccer" and "swimming" should be grouped under a common theme of "sport", while "doge" and "bitcoin" should belong together in "unstable cyrptocurrencies" category. Given a corpus of documents, a topic model discovers K latent topics,

which can be subsequently used to encode documents. In particular, each document is represented by a topic vector of size K, such that i-th position in the topic vector corresponds to the strength of topic k_i in the document.

Similarly as for VSM, a vector similarity metric can be used to compare topic vectors of a pair of documents. However, unlike VSM, a similarity in the context of topic models relies not only term frequency, but, through the topics, captures implicit context expressed within a document.

There are multiple approaches to constructing a topic model. In this work, we focus on Latent Dirichlet Allocation (LDA) model, proposed by Blei et al. [37]. More detailed description of LDA is provided in Chapter 4.2.

Word embedding-based models. A word embedding refers to a real-valued vector representation of a word, such that words with a similar *meaning* have a similar representation. Word embeddings are typically learned using a neural network with the goal of creating a d dimensional embedding space that groups words with analogous meaning next to each other in the embedding space. For instance, "soccer" and "swimming" should be close to each other, and away from "doge" and "bitcoin". Given that each word is represented by a vector, each document becomes a matrix, which is more computationally expensive to use for the purpose of similarity computation. Hence, most often an aggregation operation (e.g., average) is used to create 1d vector to represent a document.

As before, pairwise comparison between two documents is based on applying selected vector similarity metric to documents vectors. Compared to VSM-based models, which disregard the notion of context, embedding-based approaches have the advantage of capturing the meaning of words. Compared to topic models, that infer a document-level context, embeddings build the context at a word-level.

Powered by the success of word embeddings models [38, 39], in the recent years

we observed an emergence of advanced neural network architectures targeting various natural language processing tasks. In particular, introduced by Devlin et al. [40] Bidirectional Encoder Representations from Transformers (BERT) based on a deep learning architecture, Transformer, proposed by Vaswani et al. [41], brought the significant improvement in how machines can interpret and reason about natural language. In this work, we leverage BERT-based model to explore its potential in the context of bug localization. Detailed description of BERT is provided in Chapter 5.2.

The second step in the IR-based bug localization is retrieving relevant software artifacts for a newly reported bug. To this end, a bug report is passed to the model that (1) encodes a bug report in a same manner as the corpus of documents, and (2) performs a pairwise comparison between encoded bug report and documents in the index, and subsequently ranks the documents based on the similarity.

2.2 Evaluation of IR models

Given a software project with a collection of B bug reports, and a corpus of S software artifacts ranked by an IR-based bug localization model, the retrieval performance is measured with the following set of metrics [29, 25, 32].

Mean Reciprocal Rank: MRR quantifies the ability of a model to locate *first* relevant software artifact to a bug report. For each bug report b, the first relevant software artifact with a rank N, is used to compute a Reciprocal Rank (RR).

$$RR_b = \frac{1}{N}$$

The MRR score is then calculated as an average of RR across all B bug reports.

$$MRR = \frac{1}{|B|} \sum_{i=1}^{|B|} \frac{1}{RR_b}$$

Values of MRR ranges between 0 and 1, where 1 corresponds to the perfect score achieved when one of the relevant artifacts is ranked first for all bug reports. Intuitively, by focusing only on the first relevant artifact, MRR ignores the overall ranking. However, typically only a few artifacts are related to each bug, and correctly identifying at least one of them can help a developer to discover other culprits more effectively [42, 29].

Precision@K: P@K evaluates how many of the top-K software artifacts in the ranking are relevant to a bug report. The value of precision for a bug report b, $P@K_b$, is equal to the number of relevant artifacts $Rel(i)$ located in the top-K positions in the ranking.

$$P@K_b = \frac{\sum_{i=0}^{K} Rel(i)}{K}, \quad Rel(i) \begin{cases} 1 & \text{if artifact at position } i \text{ is relevant} \\ 0 & \text{otherwise} \end{cases}$$

The value of P@K across a collection of B bug reports is computed as an average of $P@K_b$.

$$P@K = \frac{1}{|B|} \sum_{i=1}^{|B|} P@K_b$$

P@K value can implicitly indicate how much effort is required from developer to examine the ranking and find the relevant artifacts. For instance, in an ideal scenario (P@K=1), all the relevant information are on the top of the ranking, while when P@K ¡ 1, the developer is required to identify false positives manually. To account for the developers efforts, P@K is computed at $K = [1, 3, 5]$.

Mean Average Precision: MAP measures how well a model can locate *all* software artifacts relevant to a bug report. For each bug report b, the positions of all relevant software artifacts in the ranking are used to compute an Average Precision ($AvgP$)

value.

$$AvgP_b = \frac{\sum_{i=1}^{|S|} P@i_b \times Rel(i)}{\#\text{ relevant artifacts}}$$

MAP is then calculated as an average of $AvgP$ across all B bug reports.

$$MAP = \frac{1}{|B|} \sum_{i=1}^{|B|} \frac{1}{AvgP_b}$$

Similarly, as for MRR, values of MAP range between 0 and 1, with 1 representing perfect retrieval result, such that all relevant artifacts occupy consecutive top positions in the ranking.

2.3 Leveraging changesets for bug localization

2.3.1 Structure of changesets

Changesets encode a set of related code modifications that are committed to a software repository (e.g., git) by a specific developer [43]. Figure 2 shows an exemplary changeset that modify two Java classes in order to introduce a timeout mechanism. Based on their structure, changesets can be decomposed into two major parts: header information and code modifications. The header information includes all of the commit metadata, such as unique commit identifier, author name, date of the commit, and commit message. While the commit message in Figure 2 is long and extensive, and includes information about relevant project components, reason for the performed modification, suggested code reviewer, and a reference to an external resource relevant to the change, in practice, commit message quality and length vary strongly across different projects and development teams [44, 45].

The actual code modifications form the core part of the changeset. The modifications are grouped by each affected file and are computed using a specific differencing algorithm. For instance, Figure 2 shows the standard output of the git diff command,

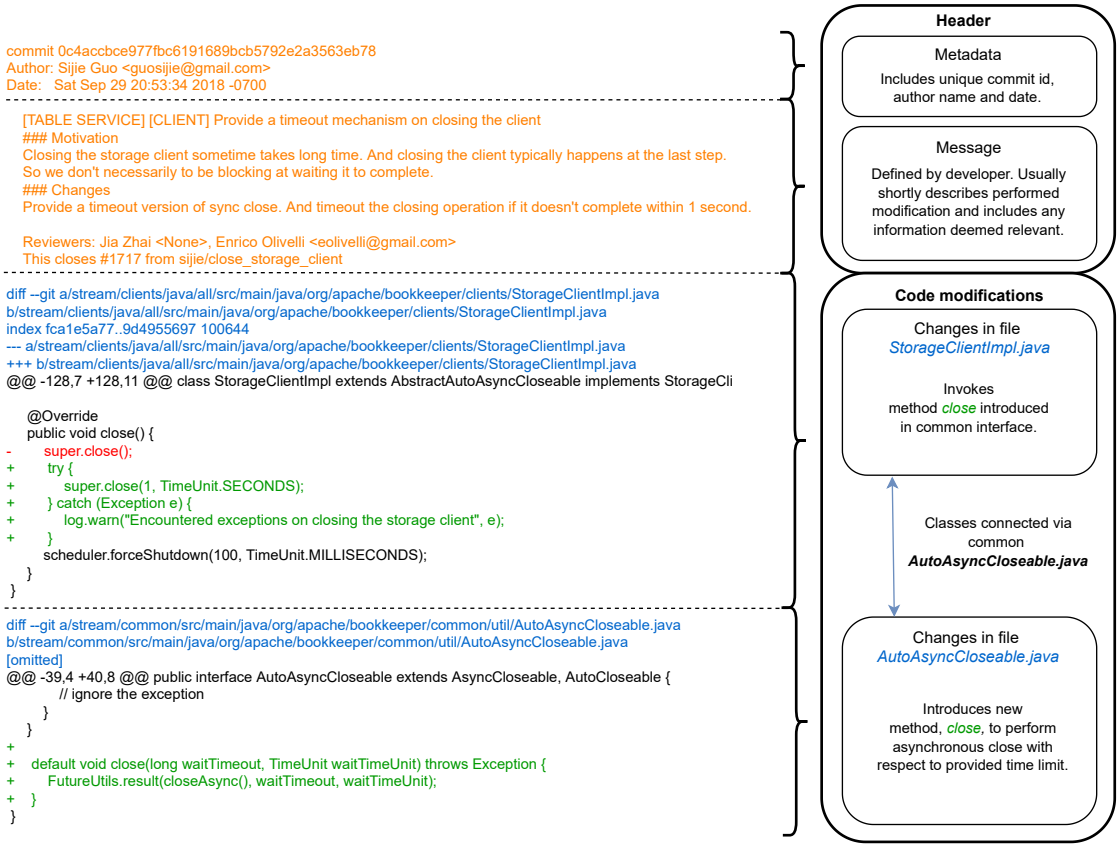


Fig. 2.: Exemplary changeset extracted from the Apache BookKeeper project. The upper half shows the header, while the body, including the code modifications, is displayed in the lower half. The added lines are marked with green color and "+" sign, and removed lines are denoted with red color and "-" sign.

which uses the Myers differencing algorithm [46]. Depending on the performed modification, the Myers algorithm encodes 2 types of line-related operations: addition and deletion, whereas line modifications are represented by both an addition and a deletion. The set of altered lines is surrounded by several unmodified lines, called context lines.

2.3.2 Advantages and challenges of using changesets

A key design choice in IR-based techniques that are applied to source code is what constitutes a document: classes, methods, files, or changesets. The document choice has a strong influence on, e.g., how topics in a topic model are formed or how an embedding space is constructed, therefore utilizing changesets as documents brings several advantages as well as challenges inherent to this data type.

Co-changed code. Changesets have the benefit of capturing co-change code, which is known to be an important predictor of software maintenance activity [47]. When leveraging changesets as a primary document dimension, co-changed code entities will often appear within the same document boundary. In the result, techniques utilizing co-occurring terms should recognize code entities that are often modified together leading to better IR performance. To confirm this hypothesis, we empirically examine whether frequently co-changed classes are likely to be expressed by similar document vectors. For this purpose, we use an LDA model and compare distributions of topics between frequently and rarely co-changed classes in 4 different popular software projects, BookKeeper, OpenJPA, Pig and ZooKeeper. The co-changed classes in each project are divided into three categories: 1) class pairs that are co-changed more than or equal to 20% of the time; 2) class pairs that are co-changed less than 20% of the time but more than or equal to 5% of the time; and 3) class pairs that are co-changed less than 5%. For instance, a class pair is considered co-changed more than or equal to 20% of the time, when the classes share at least 20% of commits in their respective modification histories. For each of these three categories of class pairs, we compute the cosine similarity of the topic distribution vectors of the class pairs using LDA model built from all of the changesets in each project. To limit the computational time and avoid corner cases, we only consider the 100 most changed classes for each

Table 1.: How the co-change relationship between classes is reflected in a LDA model built from changesets. The figure reports the average cosine similarity of different categories of co-changed classes, where the similarity is computed using the inferred LDA distribution of the text of each class.

	Average cosine similarity of classes		
	co-changed >= 20% of time	co-changed < 20% and >= 5% of time	co-changed < 5% of time
BookKeeper	0.571	0.172	0.056
OpenJPA	0.418	0.152	0.059
Pig	0.591	0.198	0.062
ZooKeeper	0.251	0.132	0.107

project. Higher cosine similarity among frequently co-changed classes indicates that the co-change relationship is also reflected by the LDA model trained on changesets. The results, shown in Table 1, demonstrate that the more often classes are co-changed, the more similar are their topic distributions, which indicates the LDA model is able to inherently reflects the co-change relationship when provided with changesets as an input.

Frequently changed code. Another similar advantage is in frequently changed code, which is likely to appear in many documents. Researchers and practitioners have observed that, over time, bug disproportionally appear in code that has been recently and frequently changed [48, 49]. Therefore, observing recent code changes, rather that only considering static source code snapshots, can have significant value to bug localization techniques.

Data quality. Challenges to using changesets for bug localization can arise from noise that can be introduced by tangled changesets, which reflect multiple unrelated changes to the source code, split changesets, where a bug fix is split across multiple

changesets, or refactoring changesets, which result in many small unrelated changes to the entire source code repository [50, 51]. However, as long as such noisy changesets are in the minority relative to ones that reflect semantically related modifications to the source code, it is likely for a topic- or embedding-based model to still produce a reasonable representation that can model how source code is changed in a particular project [52].

2.4 Diversity of bug reports

The content of bug reports can vary as some reports provide explicit localization hints through stack traces or code element names, while others contain only high-level textual description [13]. In examining the trends from interviews conducted with a large cohort of software developers from industry and open-source software, Zou et al. report that developers do not trust bug localization tools due to their inability to adapt to different types of bug reports, specifically noting that existing techniques only work on the most simple cases, with straightforward textual similarity between the bug report and code base [53].

To illustrate the different types of bug reports and their properties, in Figure 3 we show three exemplary bug reports we encountered when examining reports from the BookKeeper project. Each example contains a summary and description of the bug report and a list of fixed classes, sorted by to the number of changed lines.

Figure 3a displays bug report #600, which describes a situation causing the GarbageCollectorThread to throw an exception and provides the resulting stack trace. To fix this bug, a developer modified waitEntrylogFlushed() method in the GarbageCollectorThread by adding a check for the offset size. This bug report provides a comprehensive source of information for the developer as it directly points to the class to be fixed, with additional information on finding the method to fix available in the

Summary: GarbageCollectorThread exiting with ArrayIndexOutOfBoundsException

Description: After completing compaction, GarbageCollectorThread will do flush any outstanding offsets. When there is no offset present, its throwing following exception and exiting.

```
[stack trace]
at org.apache.bookkeeper.bookie.GarbageCollectorThread
        $CompactionScannerFactory.flush
        (GarbageCollectorThread.java:175)
[stack trace continues]
```

Fixed: GarbageCollectorThread

(a) BookKeeper-700; code references (CR) bug report.

Summary: AutoRecovery should consider read only bookies.

Description: Autorecovery Auditor should consider the readonly bookies as available bookies while publishing the under-replicated ledgers.
Also AutoRecoveryDaemon should shutdown if the local bookie is readonly.

Fixed: Auditor, BookKeeperAdmin, BookieWatcher, BookiesListener, AuditorRecoveryMain, ReplicationWorker

(b) BookKeeper-632; shared terms (ST) bug report.

Summary: Fix for empty ledgers losing quorum.

Description: If a ledger is open and empty, when a bookie in the ensemble crashes, no recovery will take place (because there's nothing to recover). This open, empty, unrepaired ledger can persist for a long time. If it loses another bookie, it can lose quorum. At this point it's impossible for the bookie to know that its an empty ledger, and the admin gets notified of missing data.

Fixed: Auditor, ReplicationWorker, AutoRecoveryMain, BookKeeperAdmin, AuditorElector

(c) BookKeeper-742; natural language (NL) bug report.

Fig. 3.: Bug reports with different characteristics.

stack trace. We refer to this type of bug reports as *code references (CR) bug reports*.

Bug report #632, shown in Figure 3b, refers to the correct way of handling

"readonly bookies when publishing ledgers". To fix the bug, the developer modified multiple classes, some of which were already mentioned in the report's text. We observed numerous common tokens shared between the bug report and modified classes: *readonly, available, bookie, publish.* Bug report #632 highlights that bug reports, even if not providing explicit localization hints, still often contain common tokens that, when grouped into common themes, reflect higher-level similarity between concepts presented in the bug report and in the source code, thus we refer to this type of bug reports as *shared terms (ST) bug reports.*

Figure 3c shows bug report #742, describing a chain of faulty behaviors starting when "recovery is not performed for an open and empty ledger after bookie crashes". When fixing the bug, the developer modified multiple classes, none of which is mentioned in the report. This bug report poses the most challenging task for automated bug localization based on IR, since it provides only textual description with no code references and no unique terms that clearly map to source code locations. However, we observe that the set of modified classes for bug #632 and #742 is similar, indicating that there exists an upper layer abstraction that correlates the set of fixed classes to bug #742. In fact, after further examination of the bug fixing history, we noticed that this set of classes was frequently modified together in bug reports that were addressing similar topics. As these bug reports do not provide any code references and are expressed in natural language, we refer to them as *natural language (NL) bug reports.*

Analyzing the examples of different types of bug reports presented above led us to the following two observations. Firstly, bug reports display different levels of details, requiring adopting different strategies to maximize performance of automated bug localization [13]. For instance, in the case of CR bug reports, it is sufficient to rely on matching code terms from a bug report to code tokens in a source code

base. Conversely, ST bug reports can leverage general context similarity. However, neither of these approaches is able to address NL bug reports. This leads to our second observation, namely that even when common themes are not present, correlated high-level concepts are still expressed by the bug report and source code [14] and can be identified by mining bug fixing history. In a results, a bug report can be matched to the most relevant code entities by examining bug fixing history to identify similar bug reports and their related code entities.

CHAPTER 3

RELATED WORK

Automatically retrieving a list of code elements based on a newly written bug report has generated significant interest among software engineering researcher community over the years. This chapter describes prior work related to bug localization, changesets, and diverse characteristics of bug reports, discussing the key ideas behind the most recent and transformative approaches to IR-based bug localization explored by researchers to date.

3.1 Code element-based bug localization

Bug localization techniques predominantly focus on identifying code elements, such as classes or methods (on the contrary to changesets), and, to this end, they typically use the VSM model. For instance, BugLocator [54] combines two rankings, one produced by similarity between the bug report and code elements and another based on similarity of the bug report to previously fixed bug reports. BLUiR [29] improves over BugLocator by using program structure to boost specific terms (e.g., class names). Similarly, Dilshener et al. [55] recognized occurrences of class names in bug reports as an essential part of bug localization and introduced a heuristic for boosting different parts of bug reports, such as stack traces or text terms. Wang et al. [56] proposed to join various techniques, creating an ensemble consisting of BugLocator, BLUiR and a defect predictor leveraging development history of a project. Lukins et al. [57] investigated using topic modeling towards bug localization. In their work, an LDA model is applied to bug reports and the relevant code elements are

identified based on the similarity between topic distribution of the bug report and source code files. Nguyen et al. proposed BugScout [31], a modified LDA model that correlates bug reports and source code elements via shared topics.

More recently, researchers have been exploring deep learning-based techniques for bug localization. Lam et al.'s technique, DNNLOC, combines a deep neural network with the VSM in order to be effective across different types of similarity [58]. Huo et al. proposed a novel convolutional neural network to learn unified feature representation from natural and programming language that captures both lexical and program structure information [59]. Their work was extended later on by modeling the sequential nature of source code using LSTM [60].

While previous body of work focuses on exploring static source code entities and not changesets, it also brought numerous discoveries in how to best apply information retrieval to bug localization. This work builds upon prior studies by incorporating strategies observed to improve bug localization performance, such as leveraging history of fixed bugs, weighting code terms and capturing contextual information.

3.2 Changeset-based bug localization

The earliest work on changeset-based bug localization is Locus [24], which is based on VSM matching of bug reports to hunks and commit logs. To adjust for localization hints, Locus adapts its similarity scores based on the proportion of code element mentions in a bug report. Bhagwan et al. [5] introduced Orca, a tool that uses a provenance graph to identify commits leading to faulty builds, which are subsequently fed into a VSM to measure the relevance of the commit to a search query. ChangeLocator [61] utilizes historical data on software crashes to build a model identifying relevant changesets based on collection on crash reports. Although this approach allows to retrieve changesets, it requires sufficient amount of historical data to

train the model, and a stack trace as an input, which is not always present in a bug report. Corley et al. [25] proposed using an Online LDA model trained on changesets to retrieve source code files. The key advantage of this technique is in building the model incrementally from a stream of changeset, hence avoiding periodic retraining.

Recently, researchers have shifted their attention to deep learning models for changeset-based localization. For instance, Murali et al. [11] proposed Bug2Commit, an unsupervised model leveraging multiple dimension of data associated with bug reports and commits, such as metrics, stack traces or commit meta data. They observed that using embeddings can lead to improvement in model accuracy when compared to BM25. Lin et al. [26] studied the trade-offs between different BERT architectures for the purpose of changeset retrieval, and observed that the speed of the model is significantly affected by the model architecture, while the retrieval accuracy remains fairly the same. However, considering that deep learning models requires significant computational resources, the speed and interactivity of these approaches makes them particularly challenging to apply in an industrial environment without further optimizations.

In this work, we explore two approaches towards changeset-based bug localization. First, we propose a topic modeling technique that improves upon Corley's approach by modeling bug reports and leveraging the history of fixed bug reports. Secondly, we investigate how to use BERT-based model to increase the retrieval accuracy without compromising the speed of the model.

3.3 Changeset representation

Building a semantically rich representations of changesets is relevant to other software engineering applications beyond bug localization, i.e., just-in-time defect prediction, recommendation of a code reviewer for a patch, tangled change predic-

tion. Approaches that define novel changeset embeddings (vector representations of changeset), including CC2Vec [62] and Commit2Vec [63], leverage the difference between added and removed lines of code, among other changeset characteristics. Corley et al. [25] studied how including different types of lines from a changeset affects the performance of LDA-based feature location, observing that including context, additions, and log messages, but excluding removed lines, achieves the best performance.

Unlike previous studies, this work investigate modeling changeset representation with BERT. To this end, we explore using different level of granularity inherent to changesets (i.e. changeset-file, file or hunk), and propose two strategies to encode code modifications.

3.4 Diverse characteristics of bug reports

Researchers recognize that differences in the content of bug reports can strongly influence the effectiveness of a bug localization technique. Kochhar et al. were among the first to report that evaluation of bug localization was biased by explicit localization hints in a significant subset of the included bug reports [64]. VSM-based techniques are likely to perform well on such bug reports, though localizing them may not be as useful to developers [28]. Mills et al. refute the idea that VSM-based bug localization are significantly aided by hints, and note that VSM can perform well for bug localization if more attention is paid to how the query is constructed from the bug report text [10]. However, their findings do not preclude additional accuracy improvements by using more complex, semantic models. At the same time, it has been reported that bug reports usually contain all the necessary information for effective IRBL [65]. In order to reduce the noise present in bug report and focus IRBL on the most relevant terms, Chaparro et al. present a query reformulation strategy based on identifying sentences within a bug report that describe the observable behavior of a system [66],

while Misoo et al. explore bug report attachments [67]. Rahman et al. observed that excessive program entities mentioned in the bug report may deteriorate the quality of IR-based bug localization and proposed a query reformulation technique, BLIZZARD [13]. Le et al. suggests that bug localization tools can be ineffective for some bug reports and builds a model that can automatically predict the effectiveness of a IR-based bug localization tool [68]. Kim et al. approach the problem similarly, by building a two-phase classifier that first determines whether the bug report has sufficient information, and, only if it does, recommends a set of code elements [69].

To address different types of bug reports, this work propose to use a mechanism allowing to adapt prediction to the content of a bug report.

3.5 Scarcity of training data in software engineering

Small datasets, with only a few thousands or even hundreds of labelled samples, are common for many software engineering tasks [70]. While those datasets are appropriate to use with most traditional machine learning algorithms, such as SVMs or Random Forests, an increasing number of approaches towards code search [71, 26, 72, 27], defect prediction [73, 74, 75, 76, 77], and bug localization [30, 58, 59, 78] leverages deep learning models. However, given that the success of deep learning stems largely from the combination of large datasets and supervised learning, the availability of training data is one of the key factors limiting deep learning performance in the software engineering domain [70, 27].

To mitigate this problem, the prior body of work concentrated on custom deep learning architectures that enable incorporating data from multiple software projects when training the model. For instance, Huo et al. [59] proposed using neural networks to learn a unified feature space for bug reports and source code that better reflects the correlation between the two types of documents. In their later work [79], this

approach was revised to extract and train on features that are transferable between different projects (i.e., cross-project). Zhu et al. [80] proposed another cross-project bug localization technique that utilizes adversarial transfer learning.

More recently, the emergence of transformers made a significant step towards training with small datasets. Transformer is a large deep learning model pre-trained on a massive amount of unlabelled data in a semi-supervised manner, and later on, it is fine tuned on a much smaller labelled dataset. To date, researchers have pre-trained a few transformer-based models on large-scale software engineering data, collected from GitHub or Stack Overflow [81, 82]. While fine tuning such models enable their application for problems where data is scarce, Gururangan et al. [83] noted that including more in-domain and task-specific data during pre-training results in a significantly higher effectiveness of the model. To this end, Lin et al. [26] introduced an extra step called intermediate pre-training, which uses a large corpus of data that is closely related to the downstream task (i.e., code search) to continue pre-training the model. Most recently, Prenner and Robbes [70] observed that domain-specific pre-training and data augmentation can improve the performance on app review and sentiment classification tasks.

To address the problem of training data scarcity in the context of bug localization, this work explores data augmentation strategies to generate synthetic bug reports with the goal of increasing the size of the training set, and subsequently improving the retrieval performance of the underlying bug localization model.

CHAPTER 4

ONLINE ADAPTABLE BUG LOCALIZATION FOR RAPIDLY EVOLVING SOFTWARE

The vast majority of prior work on IR-based bug localization has relied on files or methods taken from snapshots of the source code, typically the most recent release of the software. However, most of these approaches lack ability to update an index when new code elements are added or the existing ones are modified. The later scenario is particularly challenging as it requires not only to re-introduce the modified code element into the index, but also remove the old information. One of the advantages of leveraging changesets as a primary data unit is that they allow for continuous updates to the bug localization model, such that the model is built incrementally as changesets arrive, and captures all information about the source code at any given point in time [25]. Just-in-Time Information Retrieval-Based Bug Localization (JIT-IRBL) is a new perspective on bug localization that uses changesets to avoid costly retraining as the software changes, which has significant computational cost, even on the latest hardware [84].

This work proposes a novel model for JIT-IRBL, JINGO, based on an online variant of the Latent Dirichlet Allocation (LDA) topic modeling technique [37]. The architecture of JINGO consists of two main components: two unsupervised Online LDA models and a supervised translation module. To capture the evolution of software artifacts and bug reports over time, we use two parallel Online LDA models, one that tracks changes in the source code repo and another that tracks bugs reported in the issue tracker. The models 1) naturally represent development activity, including

frequently changed and co-changed program elements; 2) effectively handle streaming data, including the ability to deemphasize older information; and 3) raise the level of abstraction into topics, which provide a higher-level structure for detecting related artifacts. Translating between the probabilistic topic spaces maintained by the two models is performed using a translation matrix, primarily constructed from fixed bug reports and their corresponding changesets.

4.1 Online Bug Localization

In simple terms, Online Bug Localization allows for building an up-to-date model for retrieving program elements based on the most current version of the source code, i.e., HEAD in each developer's local copy of the software repo, rather than on prior versions. We deem that most of the time, the current version represents the source code that the developer cares about, rather than past snapshots reflecting a recent release of the software. To confirm the previous statement, we conducted a small survey on Reddit, asking the question *"When working on finding the location of a user-reported bug, what version of the software do you use most of the time?"*. The question was posted in the r/AskProgrammers, r/SoftwareEngineering and r/AskProgramming subreddits. More than two thirds of the respondents, 21/31, indicated that they use *"The most recent version of the software from version control"*, while 10/31 indicated they use *"The version of the software in use by the reporting user"*. One of the respondents further stated that *"I start with trying to reproduce the bug with the most recent version. If I can't, then I rollback to the version it was reported on to try to recreate it. I do quick and dirty until quick and dirty doesn't cut it."* Another one noted that *"I work with desktop applications, which are obfuscated and optimized, so debugging the released version can be difficult and time consuming"*. Yet another responded indicated a continuous deployment style of software engineering, i.e., *"I do web projects*

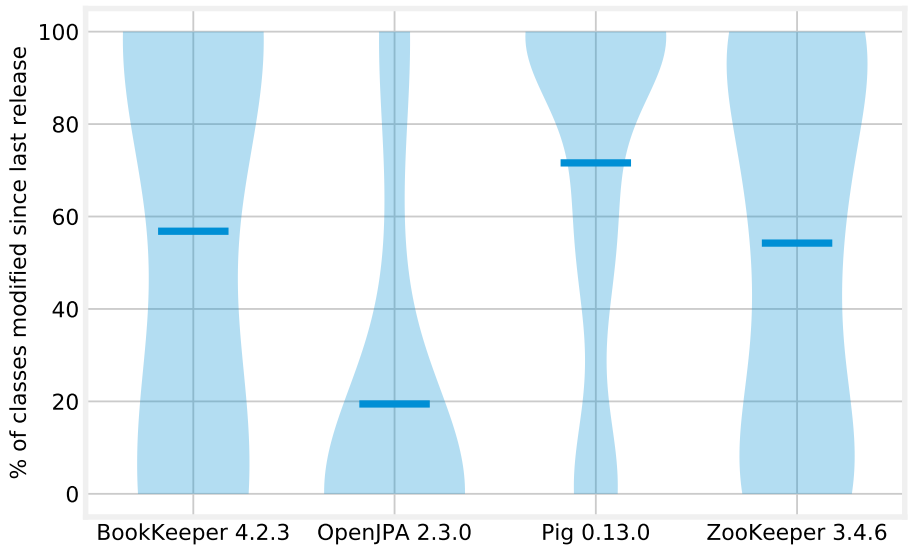


Fig. 4.: Percentage of classes that are part of a bug fix, which were modified in the time between the affected software release and the bug fix. Such classes would be inaccurately represented by a model trained on the last software release.

so I use the version in the repo, in my node_modules folder. It doesn't matter whats latest, because my app is using the version is was compiled with.". Overall, we note that developers prefer to start bug localization process by using the current version of the source code.

Further, in many cases there is evidence that the current and release versions differ in important ways. Figure 4 examines how using a release version of a software can impact bugs reported in four different popular open source software projects: BookKeeper, OpenJPA, Pig, and ZooKeeper. More specifically, the Figure shows the percentage of classes that had to be modified to fix a reported bug and that were also changed since the affected software release. In other words, the figure shows the percentage of classes that would be incorrectly represented by a bug localization model built from the prior release of the software and not a JIT-IRBL system. In three out of the four projects, BookKeeper, Pig, and ZooKeeper, the majority of

classes pertinent to the bug reports are modified between the prior software release and the time of the bug fix.

4.2 Topic modeling with Latent Dirichlet Allocation

Latent Dirichlet Allocation. LDA is a Bayesian probabilistic model designed to discover latent topics in large document corpora [37]. Based on a collection of documents, LDA infers a latent document-topic distribution and a corresponding topic-word (or topic-term) distribution, such that each document is represented as a mixture of topics, and each topic is described by a mixture of words (or terms). More specifically, LDA assumes that documents are generated according to the latent topics distributions with the following generative process:

1. Choose $\theta \sim Dir(\alpha)$
2. Choose $\phi \sim Dir(\beta)$
3. For each word w in document d:
   (a) draw a topic assignment $z_k \sim \text{Multinomial}(\theta_d)$
   (b) choose a word $w \sim \text{Multinomial}(\phi_z)$

where α is the Dirichlet prior to the document-topic distribution, β is the Dirichlet prior to the topic-word distribution, θ is $N \times K$ document-topic distribution matrix, with θ_d representing a topic distribution in a document d, and ϕ is $K \times M$ topic-word distribution matrix, with ϕ_z corresponding to topic-word distribution for topic z.

Inferring the document-topic and topic-word distributions is equivalent to training LDA and produces an interpretable model that can be applied to previously unobserved documents to extract their lower-dimensional representation.

LDA is configurable via two main hyperparameters, α and β, affecting the smoothness of document-topic and topic-word distributions. Hyperparameter α influences the document-topic distribution, such that increasing the α value causes a document to express more topics, whereas lowering α makes a document to be represented by fewer topics. Conversely, hyperparameter β influences the topic-word distribution. Raising β causes words to relate to multiple topics, while lowering β produces more specific topics with words rarely repeating between topics.

Online LDA. Updating LDA with a new document requires retraining the entire model from the beginning. As this process introduces significant time delay and computational cost, ordinary LDA is inappropriate for use in streaming environments where new documents, such as bug reports or changesets, arrive continuously. Therefore, to model the dynamics of modern software development, we use a recently devised variant of LDA, Online LDA [85], which allows to incorporate new documents through a update procedure, without the need for complete model retraining. Online LDA introduces the hyperparameter κ, which influences how quickly older information in the document stream is forgotten. Increasing κ causes the pace of forgetting to rise, thus older documents have smaller impact on the current topic distributions.

LDA on Changesets. IR models like LDA can be trained and applied on separate, but closely related datasets, which reflect a similar vocabulary of terms. In this study, we follow the methodology proposed by Corley et al. [25], that is to train an LDA model on changesets, and infer topic distributions of program elements (e.g., classes) from a snapshot of the same repository to compare them to the topic distributions of bug reports in order to perform bug localization. Such a model would retrieve program elements to the developer. Alternatively, the LDA model can both train on changesets and retrieve changesets to developers. Wen et al. argued that presenting the developer with a changeset may provide contextual clues that

are not available when retrieving program elements, however, most bug localization techniques focus on retrieving program elements [24]. The model requires no change to alternate between these two configurations, as these operations are performed on the already trained LDA model.

4.3 JINGO Model

Performing online bug localization requires the ability to operate in an environment where incoming changes are immediately integrated into a model, which is able to detect both simple (i.e., near exact terms) and high-level similarities between the code and bug reports. JINGO, a novel adaptable **J**ust-**IN**-time bu**G** l**O**calization technique based on changesets, separately models the streams of bug reports and changesets with individual Online LDA models [85], obtaining two independent topic spaces, one for bug reports and one for changesets. To translate between the two topic spaces, JINGO constructs a translation matrix based on the history of previously fixed bug reports, which captures a mapping between high-level concepts expressed in bug reports and their corresponding fixed program elements.

The architecture of JINGO, depicted in Figure 5, allows for dynamically adapting to the three different types of bug reports described in Chapter 2.4. To this end, for a newly arriving bug report, JINGO uses its changesets model and bug reports model to infer two topic distributions respectively. The first distribution is directed towards CR and ST bug reports that share code references or common concepts with the code base. The second distribution targets NL bug reports through the multiplication via the translation matrix. The key idea behind the translation matrix is to utilize the bug fixing history in the project to capture the correlation between topics occurring in bug reports and in their relevant code entities. In other words, multiplying topic distribution of a bug report by the translation matrix results in a topic distribution

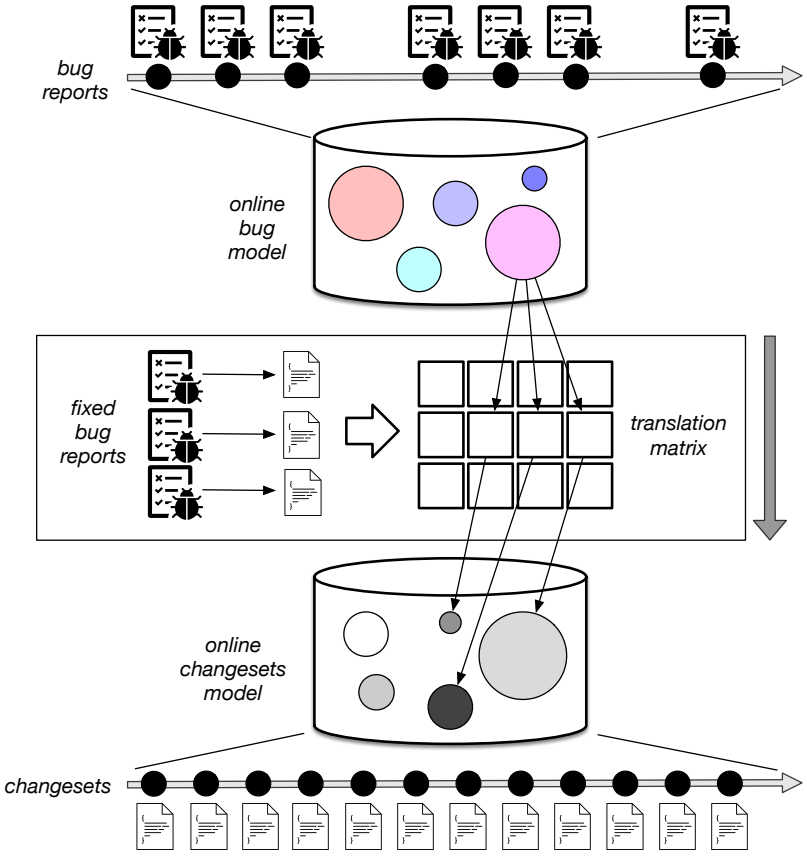


Fig. 5.: The structure of the JIT-IRBL model.

of relevant code entities in the changeset topic space. Given that a bug report often include varying content, and hence, it is unlikely to be of only one type , we use a soft mechanism, based on the ratio of code tokens to all tokens in a bug report, to combine the two distributions. Finally, this combined distribution reflects the topics in the relevant code entities, and is used to select those elements in the code base.

4.3.1 Structure of the JINGO Model

JINGO is characterized by two parallel Online LDA topic models, one for changesets and the other for bug reports, and a matrix that translates from the bug reports to the changesets topic space, as shown in Figure 5.

Changeset Model. To build the changeset topic model, we use all changesets as they are committed into the source code repository. For every changeset, we use the output of `git diff` command, which includes basic changeset information (e.g. commit SHA, author, date) and a list of changed code hunks, across all of the project's source code files, represented through added, modified or removed lines, each accompanied by 3 lines of context. We filter out the metadata and `git diff` boilerplate formatting, such as e.g., +++, obtaining a set of file names and code modifications for each changeset[86]. Following a recommendation of Eddy et al. [34], we decided to give more weight to file names by repeating them 10 times to emphasize their importance to the Online LDA. Finally, we follow the standard procedure to prepare source code for an IR model, including steps such as tokenization (using camel case and underscore), stemming using a Porter stemmer, and removal of standard programming language keywords, (e.g. if, for). In addition, we preserve the unsplit tokens into the corpus.

Bug Report Model. We train the bug reports topic model with new bugs as they are reported in an issue tracking system, e.g., JIRA. For each report, we first retrieve its summary and description. The summary is commonly a single sentence, while the description provides more details about the bug. Finally, before updating the model, for each bug report we perform a preprocessing procedure common for natural language text, including tokenization, stemming and English stop word removal. Similarly as when building changeset corpora, we also preserve unsplit camel case tokens to ease locating relevant files when explicit code references are included in the bug report.

Translation Matrix. A translation matrix - **T** - allows us to map from the bug report space to the changeset space, by simply multiplying a topic distribution inferred with the bug report model by the translation matrix, resulting in a projection into

the changeset model's space of topics. The use of a translation matrix was inspired by TM-LDA, a model based on LDA that intends to predict the expected future topics for a stream of documents [87].

To create the **T** matrix we leverage previously fixed bug reports and their corresponding changesets. The corpus of previously fixed bug reports provides an additional source of information that is leveraged by many approaches to bug localization, e.g. [54, 13, 88, 89, 90]. However, the number of fixed bug reports depends on the size of the project and it is often very limited, thus using solely bug fixing history may not provide enough data to train the **T** matrix. To solve this cold-start problem, when building the **T** matrix we also include pairs of commit logs and changesets, since commit logs have been observed to contain substantial level of information describing in natural language the purpose or the functionality of the modified code [24].

We train the translation matrix using the following set of steps. First, for a set of fixed bug report - changeset pairs, and if necessary, commit log - changesets pairs, we infer a topic distribution for each fixed bug report using the bug report model and store it in matrix **B**, where the rows of the matrix correspond to the distribution inferred for each of the fixed bug reports and the number of columns in **B** corresponds to the number of topics in the bug report LDA model. Second, an analogous procedure is performed for the matching changesets. For these, the topic distribution is inferred by the changeset model and added to another matrix - **A**, with rows containing topic distribution of changesets and column corresponding to the number of topic in the changeset model. In this way, each fixed bug report - changeset pair creates one corresponding row in matrices **B** and **A** respectively. Finally, training the translation matrix **T** reduces to solving the following equation.

$$\mathbf{T} = \min_{T} \|\mathbf{BT} - \mathbf{A}\|^2$$

To solve the equation for the unknown **T** we perform least square minimization. Note that **T** is of size, number of topics in bug report model by number of topics in changeset model. Therefore, it requires *at least* as many rows in **A** and **B**, i.e., fixed bug reports, to compute. On the other hand, providing more than the required minimum amount of data to train the **T** matrix is desirable, as it is likely to increase the quality of mapping between topic spaces. We introduce a parameter specifying the minimum amount of data required to train the **T** matrix, ω, expressed as a multiplying factor of the (maximum) number of topics required in the topic models. For instance, $\omega = 1.5$ and 50 LDA topics, would indicate that 75 fixed bug reports (or commit logs) are required to build the **T** matrix.

Once we determine the translation matrix, mapping between the bug reports to the changeset topic space is simple: we multiply the bug-related topic distribution for a new bug report by **T** matrix to get the equivalent distribution in changeset space. The computational cost of updating **T** over time is not large, and it is proportional to the number of fixed bug reports used. The cost can also be controlled by using a window based approach.

4.3.2 Using JINGO for Prediction

Using a trained JINGO model, we follow the workflow in Figure 6 to perform bug localization for a newly arriving bug report. To start, we preprocess the new bug report to construct a query, using the same procedure as when building the bug reports corpus. We use the query to infer two topic distributions, one using the changeset model – *changeset-related topic distribution*, and another using the bug report model – *bug-related topic distribution*. To map the *bug-related topic distribution* from the bug report model to the changeset model, we multiply it by the **T** matrix, obtaining a *co-occurrence topic distribution*. Note that, if the **T** matrix is not trained due

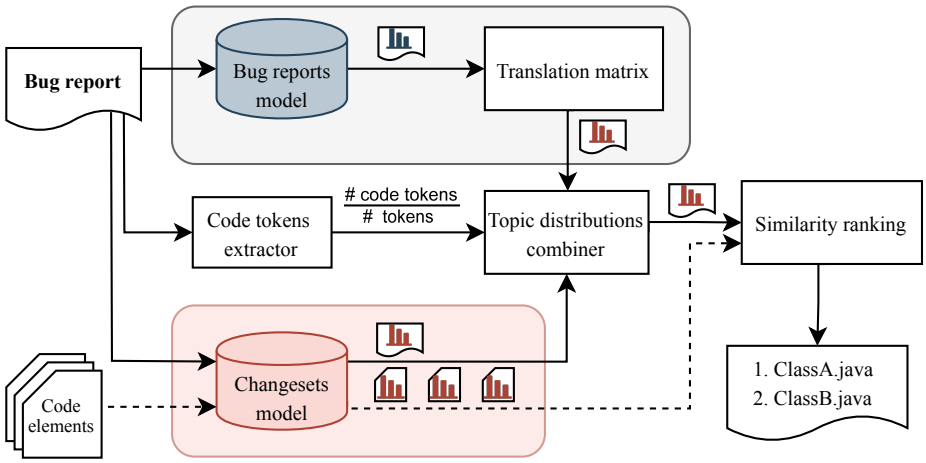


Fig. 6.: Bug localization for a new bug report, where topic distributions in the bug report space ( ) are translated and combined with others in the changeset space( ) to produce a combined topic distribution; cosine distance between the combined distribution and distributions for each of the code elements is used to produce the final ranking.

to lack of data, this step is skipped and the final prediction is based solely on the *changeset-related topic distribution*.

Bug reports typically include various content and are likely to be related to the code base in more than one way via, e.g., code element names, shared tokens or bug fixing history. Hence, we decide to combine the *changeset-related topic distribution* and the *co-occurrence topic distribution* to better reflect bug report's characteristic. To this end, we adopt a weighting strategy based on the number of code tokens that appear in the bug report. A token is considered to be a code token if it is in a camel case format or it corresponds to one of class names in the source code base. We prioritize the importance of either of the distributions based on the intuition that the more code tokens are present in the bug report, the stronger is the similarity between the bug report and the source code base. If this is the case, we weigh the

changeset-related topic distribution more strongly. On the other hand, if code tokens are rare and the bug report predominantly consists of natural language text, then the *co-occurrence topic distribution* becomes more significant, as it leverages topics co-occurrences patterns that are not directly connected to bug's textual content. We use λ to refer to the ratio of code tokens to the total number of tokens in a bug report. To account for the fact that natural language is more verbose in general and therefore the typical number of code tokens is much lower than the number of natural language tokens, we introduce an amplifying factor γ, which increases the importance of program element terms. We compute the *combined topic distribution* using the following equation:

$$dist_{combined} = norm(dist_{changesets} * \lambda * \gamma + dist_{co-occurence} * (1 - \lambda))$$

Next, we use the changeset model to infer topic distributions for all documents in the most recent snapshot of the source code. As noted by researchers, bugs often pertain to small part of the code [88, 91], thus inferring at the granularity of large source code files, e.g., classes, may negatively impact the performance of IR-based techniques. To solve this issue, we infer topic distributions of methods for each class and make a pairwise comparison against the combined topic distribution ($dist_{combined}$) of the query. Finally, each class is represented by a method that minimizes the cosine distance to the query and the classes are ranked according to increasing cosine distance to create a recommendation list.

4.4 Evaluation setup

4.4.1 Datasets

To evaluate the performance of JINGO we use Bench4BL [84], a collection of bug reports and corresponding lists of fixed source code files extracted from 51 open-source projects. Each bug report includes information such as summary, description, creation date and list of fixed files. Bug reports are stored according to the version in which they were reported. The structure of the Bench4BL dataset allows for immediate use for evaluating release-based bug localization techniques, however it is not immediately suitable for a JIT approach. The key missing component in the dataset are the explicit connections between bugs and fixing changesets, which are necessary to perform JIT evaluation. To this end, we adapted Bench4BL's code to retrieve the required data in a way that maintains complete consistency with the original dataset. More specifically, following Bench4BL's prior approach, to link a bug to a changeset we searched for an explicit mention of the bug identifier in the commit message. If more than one changeset was related to a bug, we selected the latest changeset, which is Bench4BL's existing assumption. Due to difficulties when finding clearly discernible links between bugs and changesets, we excluded 5 projects from Bench4BL: JBMeta, ENTESB, ZXing, WLFY and SOCIALLI. Our final evaluation Bench4BL dataset is shown in Table 3 and consists of 46 projects and 2,125 bug reports.

One of the key advantages of JINGO is the ability to perform quick model updates as the new data arrives, making it suitable for repositories exhibiting high code churn. However, most of the Bench4BL's projects display limited commit traffic, hence we curate an additional dataset by selecting open-source projects hosted on GitHub. To locate repositories of interest on GitHub, we perform a repository search query for all

active Java repositories with at least one commit in 2019, and size of at least 10MB. Next, we sort the repositories in descending order of their code churn, calculated as the average number of commits per day, and select the top 10 repositories that use an explicit label to mark a reported issue as a bug (e.g., *bug*, *kind:bug*). To build a goldset that connects a bug report to its fixing changesets and to retrieve a list of modified files, for each project we manually investigate 20 randomly selected bug reports to identify the project's convention for linking an issue to a changeset. In general, we note that developers tend to use keywords, such as "fixes", "closes" or "resolves", or a project name followed by an issue number. Hence, to link an issue to its implementing commit, we test commit messages and pull requests against two types of regex, `keyword_#XXX` and `project_name-#XXX`, where `XXX` denotes an issue number. In the case of identifying multiple changesets for a bug report, we follow Bench4BL's approach. Our final dataset of high code churn repositories, HCC-Repo, contains 10 projects and the total of 874 bug reports.

Table 3.: Evaluation datasets for JINGO

Group	Project	Commits	Evaluated version	Previously fixed bug	Fixed bugs in version
Hyperparameter tuning projects					
[25]	BookKeeper	574	4.3.0	223	102
	OpenJPA	4,616	2.3.0	1039	100
	Pig	2,584	0.14.0	1063	155
	ZooKeeper	1,245	3.5.0	368	235
	Total	9,019	–	2,693	592
Evaluation projects - BENCH4BL					
COMMONS	Codec	1,387	1.5	13	11
	Collections	2,837	4.0	4	49
	Compress	1,033	1.4	41	12
	Configuration	2,743	1.7	66	31
	Crypto	548	1.0.0	1	8
	CSV	1,085	1.3	10	5
	IO	1,850	2.0	23	25
	Lang	5,231	3.5	217	40
	Math	5,795	3.0	74	39
	Weaver	420	1.3	0	2

APACHE	Camel	3,7986	2.15.0	1,298	147
	Hbase	16,015	2.0.0	446	418
	Hive	10,096	2.1.0	547	221
SPRING	AMQP	1,254	1.5.0	78	12
	Android	504	1.0.0	0	6
	Batch	4,991	2.1.0	460	25
	Batch Admin	444	1.2.0	4	5
	Data Commons	1,356	1.13.0	140	20
	Data GemFire	1,123	1.4.0	47	25
	Data JPA	875	1.11.0	131	18
	Data MongoDB	1,879	1.5.0	139	28
	Data Neo4j	461	4.0.0	6	36
	Data Redis	1,421	1.8.0	38	16
	Data Rest	883	2.6.0	116	22
	Framework	14,635	3.0.0	0	13
	Hadoop	1,473	2.1.0	18	18
	LDAP	979	1.3.0	11	12
	Mobile	324	1.1.0	2	8
	Roo	6,305	1.1.0	201	134
	Security	5,959	3.1.0	305	67
	Security OAuth	1,149	2.0.0	112	9
	Shell	275	1.1.0	3	2
	Social	1,721	1.1.0	11	6
	Social FB	1,265	1.1.0	1	7
	Social Twitter	1,090	1.1.0	2	4
	Webflow	2,438	2.0.0	24	24
	Web Service	2,042	1.5.0	50	23
WILDFLY	Arquillian	606	1.1.0	0	1
	Core	11,379	3.0.0	125	120
	Elytron	2,415	1.1.0	2	20
	Maven Plugin	481	1.1.0	0	5
	Swarm	3,152	2016.10	55	9
ECLIPSE	org.aspectj	7,721	286	–	286
	platform.swt	24,585	98	–	89
	jdt.core	22,459	94	–	88
	pde.ui	12,722	60	–	57
Evaluation projects - HCC-Repo					
HCC-Repo	DataHelix	6,153	–	107	54
	eXist	18,612	–	286	86
	Flink	8,579	–	424	100
	Hazelcast	8,728	–	425	100
	Magarena	24,883	–	203	72
	MegaMek	18,899	–	163	100
	Micronaut	9,469	–	302	100
	OpenDJ SDK	7,868	–	358	100
	ShardinSphere	27,014	–	383	100
	WooCommerce4A.	10,658	–	102	62
	Total	358,526	–	6,799	2,999

4.4.2 Hyperparameter Optimization

JINGO relies on probabilistic models with several impactful parameters, hence prior to the evaluation study we perform hyperparameter optimization using a sepa-

rate dataset released as part of Corley et al.'s study [25]. To optimize the performance of JINGO we perform hyperparameter tuning using grid search. We use a set of four open source projects, BookKeeper, OpenJPA, Pig and ZooKeeper, published by Corley et al. [25], which are not part of our evaluation set. The key parameters to optimize are shown in Table 4 and include the Online LDA priors for the two streaming topic models, and the set of parameters introduced by the bug localization technique. Since some parameters rely solely on the streams of bug reports and changesets, while others influence the combination of these models for bug localization, we divided our hyperparameter optimization in two steps. First, we optimized the two Online LDA models independently using two metrics, perplexity and coherence [92, 93], and, second, we optimized the bug localization parameters based on the MRR metric.

Several researchers have highlighted the importance of hyperparameter tuning of topic models for software engineering applications [94, 95]. Therefore, our first grid search focused on choosing appropriate LDA hyperparameters for the two topic models. To decide on parameters values to investigate for the changeset model, we followed results reported in previous research [25]. In the case of the bug report model we used characteristics of the bug reports corpora, such as e.g., the number of documents and unique words, in relation to similar changeset corpora, to choose the set of parameters to explore. For the decay factor we used three values that correspond to the minimum, mean and maximum possible value for that parameter. Note that, we did not optimize priors α and β explicitly, relying instead on the automated estimation approach implemented in the *gensim* topic modeling library. We computed coherence and perplexity after training a model with 25%, 50% and 75% of the respective corpus stream, in order to avoid bias towards longer streams (or larger corpora), averaging the values of the metrics to assess the model's performance. At the conclusion of the first grid search we selected the top parameter combinations

Table 4.: Hyperparameters and their corresponding values used during grid search; selected, optimal values are in bold.

Component	**Parameter**	**Value**
Changeset model	# topics – k	$\{75, \mathbf{100}, 150, 200\}$
	decay factor – κ	$\{0.5, \mathbf{0.75}, 1.0\}$
Bug report model	# topics – k	$\{10, 25, \mathbf{50}, 100\}$
	decay factor – κ	$\{0.5, 0.75, \mathbf{1.0}\}$
Bug localization	fixed bug reports factor – ω	$\{1, \mathbf{1.5}, 2.0\}$
	model combining factor – γ	$\{1, 3, \mathbf{5}, 7\}$

for each model to use in the second grid search.

In the second phase, we searched over the parameters related to JINGO namely ω and λ. The first parameter influences the number of fixed bug reports the model needs to observe before building the translation matrix. The value of ω is a multiplying factor of the minimum number of observed fixed bug reports, where the minimum number is equal to the larger value of number of topics between models. In the case of λ, we selected a set of values based on similar experiments by Wen et al. [24]. Values marked with bold in Table 4 represent the set of final optimal values for all the parameters that we used during evaluation.

4.4.3 Experiment setup

To evaluate the performance of the proposed bug localization we simulate the development history of a specific software project, continuously updating the model with a time series of bug reports and changesets. At each timestep, we also update the translation matrix with each fixed bug report and its corresponding changeset. Therefore, when evaluating for a specific newly arriving bug report, the changeset model contains all changesets that occurred before the time of the bug fixing commit, while the bug report model includes all bug reports reported before the commit

timestamp.

To evaluate the statistical significance of the difference in performance between JINGO and the baseline, we compute the Wilcoxon signed-rank test with Holm correction and effect size using the Cliff's delta δ. The values of δ ranges from +1 to -1, where -1 implies that all values in the first group are larger than values in the second group, and +1 represents the opposite situation. The effect size was intepreted using the following criteria: (1) small effect = $|\delta| > 0.147$; (2) medium effect = $|\delta| > 0.33$; and (3) large effect = $|\delta| > 0.474$ [96]. Note that in the case of small projects with number of bug reports equal or less than 10, we report the evaluation metric but did not conduct statistical testing.

4.4.4 Research Questions

RQ1: How accurate is JINGO in locating source code files relevant to a bug report? To answer RQ1, we use the proposed approach to identify buggy files in Bench4BL and high code churn datasets, and measure the effectiveness of JINGO with respect to the previously defined metrics. We compare the performance of JINGO against the JIT feature location technique proposed by Corley et al. [25]. Corley et al.'s technique has several similarities to JINGO, as it uses a single Online LDA model trained on changesets to locate program elements relevant to a given bug report. The main difference is in that JINGO models bug reports via a separate topic model and, through the usage translation matrix, incorporates information about previously fixed bug reports. Since Corley's approach is based solely on Online LDA trained with changesets, we set its hyperparameters to values identified to perform best for the changeset Online LDA part of our model (as described in Section 5.2).

RQ2: What is JINGO's time overhead required to update the model? The key advantage of using an online model is the ability to update the model with

new data once it arrives. However, a key question remains: how rapid is an update procedure when compared to a full model rebuild? To answer this question, we collect the execution logs for all studied projects and, based on the recorded timestamps, we compute the time required to *build* and to *update* the model. Specifically, *build time* is the time required to build the model from scratch to the target version of the project we run the evaluation on, and *update time* reflects the time needed to update the model with one changeset (i.e., a single commit).

RQ3: How does JINGO compare to static (i.e., non JIT) bug localization techniques in terms of time overhead and accuracy? Most bug localization approaches proposed thus far use a static, snapshot-based model of the software. Although online models in general, and JINGO in particular, focus on updating a model as the software changes, we still need to contrast its bug localization accuracy to state of the art static models. In this research question, we compare the accuracy of JINGO in retrieving relevant results to state of the art static approaches based on the Vector Space Model (VSM). We also quantify the time overhead to rebuild the model for such techniques and contrast it to JINGO.

RQ4: Can JINGO adapt to different types of content in bug reports? Bug reports have diverse characteristics and can embody different level of details, that, when leveraged, can increase the effectiveness of a bug localization technique. The aim of RQ3 is to investigate how well the proposed model captures different types of bug reports. To this end, first, we examine bug reports in our corpus of almost 3,000 with respect to the number of code tokens relative to the number of overall tokens. Next, we group the bug reports based on the ratio of the tokens to bins of size 0.02, which we experimentally established to be the smallest interval ensuring good visualization level, and depict MAP values comparing JINGO against Corley's et al.'s approach results. The idea is that bug reports with high proportion of code tokens are more

likely to belong to the CR category, while those with a low proportion or no code tokens are likely to belong to the NL category.

In addition, we randomly sampled a set of 322 bug reports from our corpus (95% confidence level with a 5% margin error to the target bug report population). The sample spanned 40 different projects. Subsequently, one of the authors manually categorized each of the bug reports into one of the three groups (i.e., CR, ST, and NL). We report MAP and MRR scores contrasted to Corley's et al.'s approach, however, the results for the remaining metrics are analogous.

4.5 Results

Table 5.: Evaluation results for JINGO compared to Corley et al. [25]. The per-project higher value of a metric is highlighted by light gray background (n.nnn). Per-group of projects we used dark gray background (n.nnn) to highlight the higher value. Statistically significant increase in MRR and MAP values (p-value < 0.05) is marked with bold type with a superscript indicating the effect size: s – small, m – medium, l – large. Projects marked with † had less than 10 bug reports, thus statistical testing was not conducted.

G.	Project	MRR		MAP		P@1		P@3		P@5	
		JIN.	[25]	JIN.	[25]	JIN.	[25]	JIN.	[25]	JIN.	[25]
COMMONS	Codec	0.572	0.679	0.536	0.621	0.364	0.545	0.818	0.818	0.818	0.818
	Collections	0.629	0.591	0.490	0.478	0.490	0.429	0.694	0.714	0.796	0.776
	Compress	0.418	0.214	0.255	0.133	0.250	0.083	0.500	0.167	0.583	0.333
	Configuration	**0.685**^{s}	0.512	**0.482**^{s}	0.379	0.548	0.387	0.774	0.548	0.903	0.677
	Crypto†	0.654	0.453	0.514	0.406	0.571	0.286	0.714	0.429	0.714	0.571
	CSV†	0.647	0.717	0.638	0.598	0.600	0.600	0.600	0.800	0.600	1.000
	IO	0.694	0.627	0.674	0.565	0.560	0.520	0.800	0.680	0.840	0.720
	Lang	**0.798**^{l}	0.449	**0.719**^{l}	0.372	0.700	0.300	0.875	0.500	0.925	0.650
	Math	0.584	0.613	0.474	0.453	0.462	0.487	0.641	0.615	0.744	0.744
	Weaver†	0.563	0.536	0.549	0.490	0.500	0.500	0.500	0.500	0.500	0.500
	Group avg.	***0.624***^{s}	*0.539*	***0.533***^{s}	*0.449*	*0.504*	*0.414*	*0.692*	*0.577*	*0.742*	*0.679*
APACHE	Camel	0.258	0.272	0.181	0.192	0.163	0.170	0.279	0.293	0.374	0.374
	HBase	0.393	0.355	0.275	0.244	0.278	0.246	0.428	0.395	0.524	0.462
	Hive	**0.248**^{s}	0.208	**0.176**^{s}	0.128	0.137	0.133	0.280	0.223	0.365	0.303
	Group average	*0.299*	*0.278*	***0.211***^{s}	*0.188*	*0.193*	*0.183*	*0.329*	*0.303*	*0.421*	*0.380*

SPRING	AMQP	**0.561**$^{\mathbf{m}}$	0.221	**0.381**$^{\mathbf{m}}$	0.172	0.417	0.083	0.667	0.250	0.750	0.333
	Android†	0.239	0.379	0.243	0.195	0.167	0.167	0.167	0.667	0.167	0.667
	Batch	0.463	0.621	0.288	0.377	0.320	0.480	0.480	0.760	0.600	0.800
	Batch Admin†	0.729	0.126	0.542	0.118	0.600	0.000	0.800	0.200	0.800	0.200
	Data Commons	**0.569**$^{\mathbf{m}}$	0.343	**0.483**$^{\mathbf{m}}$	0.285	0.400	0.200	0.750	0.350	0.800	0.600
	Data GemFire	0.645	0.532	0.357	0.315	0.520	0.360	0.680	0.640	0.840	0.720
	Data JPA	0.403	0.410	0.312	0.328	0.278	0.333	0.500	0.444	0.556	0.500
	Data MongoDB	0.531	0.418	0.367	0.303	0.357	0.286	0.643	0.500	0.786	0.571
	Data Neo4j	0.242	0.262	0.179	0.148	0.056	0.167	0.389	0.222	0.444	0.333
	Data Redis	0.586	0.505	0.393	0.271	0.500	0.375	0.625	0.625	0.625	0.688
	Data REST	0.449	0.504	0.362	0.314	0.273	0.318	0.500	0.636	0.773	0.773
	Framework	0.080	0.048	0.060	0.038	0.000	0.000	0.077	0.000	0.154	0.000
	Hadoop	0.394	0.434	0.300	0.308	0.222	0.333	0.444	0.444	0.667	0.444
	LDAP	0.427	0.258	0.351	0.163	0.250	0.167	0.500	0.333	0.667	0.333
	Mobile	0.668	0.543	0.634	0.510	0.500	0.375	0.750	0.625	0.875	0.750
	Roo	**0.221**$^{\mathbf{s}}$	0.184	**0.191**$^{\mathbf{s}}$	0.141	0.119	0.104	0.246	0.187	0.306	0.246
	Security	0.469	0.410	**0.356**$^{\mathbf{s}}$	0.311	0.328	0.313	0.552	0.448	0.627	0.507
	Security OAuth†	0.353	0.261	0.291	0.256	0.222	0.111	0.444	0.222	0.444	0.444
	Shell†	0.556	0.667	0.436	0.440	0.500	0.500	0.500	1.000	0.500	1.000
	Social†	0.554	0.571	0.483	0.474	0.500	0.500	0.500	0.500	0.667	0.667
	Social FB†	0.545	0.694	0.467	0.592	0.429	0.571	0.571	0.714	0.571	0.857
	Social Twitter†	0.204	0.773	0.235	0.511	0.000	0.750	0.250	0.750	0.250	0.750
	Webflow	0.282	0.453	0.185	**0.338**$^{\mathbf{s}}$	0.125	0.333	0.375	0.500	0.500	0.542
	Web Service	0.515	0.330	0.379	0.237	0.348	0.217	0.609	0.348	0.696	0.478
	Group average	***0.445***$^{\mathbf{s}}$	*0.414*	***0.345***$^{\mathbf{s}}$	*0.298*	*0.310*	*0.294*	*0.501*	*0.474*	*0.586*	*0.550*
WILDFLY	Arquillian†	1.000	0.200	0.538	0.132	1.000	0.000	1.000	0.000	1.000	1.000
	Core	0.183	0.208	0.139	0.142	0.075	0.133	0.250	0.208	0.308	0.267
	Elytron	0.381	0.281	0.355	0.262	0.250	0.150	0.400	0.350	0.600	0.350
	Maven Plugin†	0.510	0.451	0.459	0.357	0.400	0.200	0.600	0.600	0.600	0.800
	Swarm†	0.341	0.203	0.318	0.164	0.222	0.111	0.444	0.222	0.556	0.333
	Group average	*0.483*	*0.269*	*0.362*	*0.211*	*0.389*	*0.119*	*0.539*	*0.276*	*0.613*	*0.550*
ECLIPSE	jdt.core	**0.258**$^{\mathbf{l}}$	0.066	**0.124**$^{\mathbf{l}}$	0.019	0.148	0.034	0.295	0.068	0.375	0.091
	pde.ui	**0.192**$^{\mathbf{s}}$	0.149	**0.130**$^{\mathbf{s}}$	0.101	0.105	0.105	0.193	0.105	0.316	0.193
	platform.swt	**0.254**$^{\mathbf{m}}$	0.090	**0.210**$^{\mathbf{m}}$	0.065	0.135	0.034	0.315	0.079	0.393	0.112
	org.aspectj	0.041	**0.113**$^{\mathbf{s}}$	0.022	**0.041**$^{\mathbf{s}}$	0.021	0.073	0.031	0.119	0.042	0.143
	Group average	***0.186***$^{\mathbf{s}}$	*0.105*	***0.122***$^{\mathbf{s}}$	*0.056*	*0.102*	*0.062*	*0.209*	*0.093*	*0.282*	*0.135*
HCC-Repo	DataHelix	0.150	0.196	0.109	0.121	0.094	0.151	0.208	0.264	0.283	0.321
	eXist	**0.123**$^{\mathbf{s}}$	0.100	0.049	**0.057**$^{\mathbf{s}}$	0.250	0.250	0.321	0.286	0.357	0.310
	Flink	0.269	0.308	0.178	0.195	0.172	0.222	0.273	0.323	0.364	0.364
	Hazelcast	0.386	0.324	0.235	0.219	0.280	0.204	0.440	0.398	0.550	0.459
	Magarena	0.111	0.104	0.085	0.073	0.028	0.042	0.141	0.127	0.183	0.127
	MegaMek	**0.236**$^{\mathbf{m}}$	0.106	**0.152**$^{\mathbf{m}}$	0.076	0.182	0.111	0.313	0.141	0.404	0.182
	Micronaut	**0.208**$^{\mathbf{s}}$	0.158	**0.148**$^{\mathbf{s}}$	0.120	0.131	0.091	0.222	0.162	0.253	0.232
	OpenDJ SDK	0.225	0.200	0.152	0.133	0.155	0.113	0.216	0.216	0.299	0.289
	ShardingSphere	**0.172**$^{\mathbf{s}}$	0.131	**0.128**$^{\mathbf{s}}$	0.101	0.234	0.202	0.287	0.287	0.351	0.330
	WooCommerce4A.	**0.375**$^{\mathbf{m}}$	0.140	**0.194**$^{\mathbf{m}}$	0.093	0.295	0.082	0.443	0.197	0.541	0.279
	Group average	**0.225**$^{\mathbf{s}}$	0.177	**0.143**$^{\mathbf{s}}$	0.119	0.182	0.147	0.286	0.240	0.358	0.289
	Total	**0.415**$^{\mathbf{s}}$	0.352	**0.321**$^{\mathbf{s}}$	0.262	0.308	0.251	0.470	0.396	0.545	0.488

4.5.1 RQ1: Retrieval accuracy

Table 5 shows the performance of JINGO alongside Corley et al.[25] baseline with respect to 5 metrics: MRR, MAP, Top@1, Top@3, and Top@5. Statistically significant improvements in JINGO are marked with bold type with effect size (small, medium, large) noted in the superscript. Overall, JINGO demonstrates higher performance in locating buggy files across all of the above metrics with statistically significant increase of 6.3% and 5.9% for MRR and MAP respectively (p-value < 0.05). At a finer scale, we observe an improvement in the average results across all groups of software projects, with statistically significant difference in MRR and MAP values for the COMMONS, SPRING, ECLIPSE and HCC-Repo groups and MAP value for the APACHE group.

Specifically, JINGO achieves better performance in terms of MRR values for 37 out of 56 projects with a statistically significant improvement in 14 out of 43 projects that has more than 10 bug reports. In the ECLIPSE group, MRR results for all but one project were improved with statistical significance, increasing MRR score obtained by the baseline by 19.2%, 4.3% and 16.4% for *jdt.core*, *pde.ui* and *platform.swt*. In the case of *org.aspectj*, the baseline outperformed JINGO by 7.2%. We note the highest variance of increase and decrease of MRR values for the SPRING group, with JINGO outperforming the baseline in 13 out of 24 projects. Finally, in the HCC-Repo group, JINGO improved MRR scores for 8 out of 10 projects, including a statistically significant improvement observed for 5 projects.

Improvement in MAP values is achieved by JINGO for 42 out of 56 projects with statistically significant difference noted for 14 projects. Similarly as for the MRR metric, the highest MAP increase of 8.4% is observed for the COMMONS group with JINGO outperforming the baseline for all but one project. We also observe

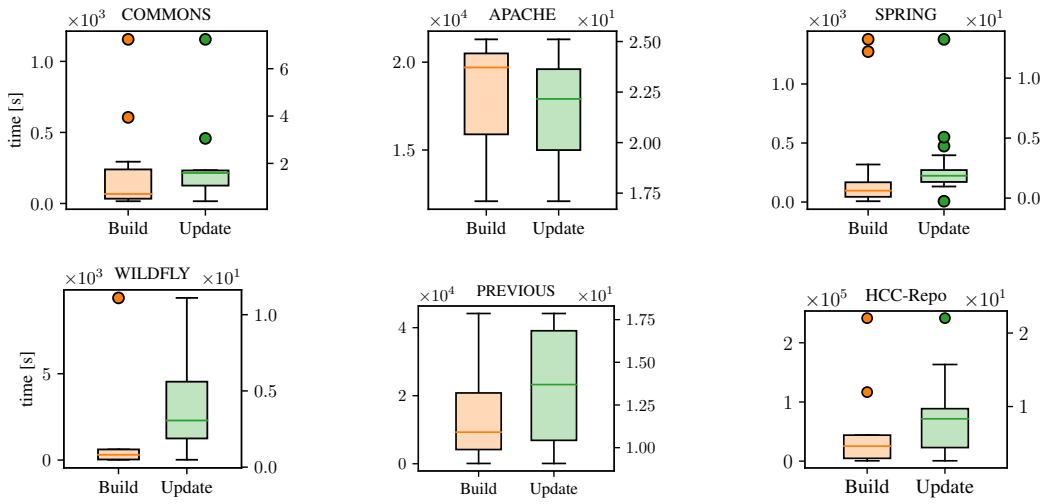


Fig. 7.: Average time in seconds required to *build* and to *update* JINGO. Note that *bulid* and *update* time are illustrated with two independent y axes.

the ECLIPSE projects achieved significantly higher results when using JINGO with a small effect size. For the SPRING group, the proposed approach improves MAP results in 17 out of 24 projects. The improvement is statistically significant for 2 projects with medium (*AMQP*, *Data Commons*) and 2 with small (*Roo*, *Security*) effect sizes. In the HCC-Repo group, we note that JINGO outperforms baseline for 7 out of 10 projects with a significant improvement in 4 projects.

On average, JINGO outperforms the baseline for Top@1, Top@3 and Top@5 recommendation by 5.7%, 7.4% and 5.7% respectively. For 4 project groups, APACHE, SPRING, ECLIPSE and HCC-Repo, we observe that as we consider longer recommendation list the difference between the techniques is growing. However, for the COMMONS and WILDFLY groups, the difference between JINGO and Corley et al.'s approach reduces at Top@5 and Top@3 respectively.

4.5.2 RQ2: Time overhead to update the model

Given the fact that JINGO is an online model, it is updated with each newly arriving data instance, keeping the model up-to-date with minimal time overhead. To investigate the performance benefit of using an online model over a model that requires re-building, in Figure 7 we show the time required to build and to update the model, averaged per each group of projects. Across all groups, we observe that the update time is significantly lower than build time, with the average speedup of 100 for the groups with smaller projects (SPRING and COMMONS), and up to about 1000 in the case of large projects (APACHE, ECLIPSE and HCC-Repo). This indicates that with the growing size of a repository, the cost of re-training the model becomes even more prohibitive. As an example, consider the results obtained for APACHE projects, with the average build time close to $20,000s = 5.5h$ and update time of about $22.5s$. With new changesets being committed to a repository multiple times during a day, a model that relies only on re-building is outdated every couple of hours and consumes computational resources for a significant amount of time. On the other hand, utilizing an online model with an update procedure significantly reduces the time overhead, hence allowing to incorporate new information as it arrives.

4.5.3 RQ3: JINGO compared to static bug localization

In order to compare JINGO's bug localization accuracy to that of the state of the art static models, we select two recent techniques based on the Vector Space Model (VSM), BLiA [97] and BRTracer [91]. We used source code for both of these techniques that was shared as part of Bench4BL [84]. When considering all types of bug reports, simpler models like VSM have been reported to outperform LDA on bug localization [98]. Table 6 shows the average MAP and MRR for JINGO, BLiA and

Table 6.: Comparison between JINGO and two VSM techniques based on average (per bug report) accuracy and time overhead measures.

	Accuracy		***Time overhead***	
Technique	**MRR**	**MAP**	**Build Time [s]**	**Update Time [s]**
JINGO	0.323	0.241	2964.955	4.786
BLiA	0.371	0.314	101.315	101.315*
BRTracer	0.471	0.367	91.245	91.245*

BRTracer on the Bench4BL set of repositories; we compute the average per bug report in order to account for some projects having more instances than others. Alongside the accuracy measures, Table 6 shows the average time overhead to construct the model and to update it with a single changeset, also averaged across all the projects in our dataset. BLiA and BRTracer do not have an update mechanism so the update time can be assumed to be equivalent to (re-)building the model.

BLiA and BRTracer outperforms JINGO in retrieval accuracy with the improvement in MRR of 4.8% and 14.8% respectively. On the other hand, JINGO offers significantly more time-efficient update procedure that is about 20 times faster than performing a full model rebuild for the VSM-based baselines. Although JINGO needs more time to initially construct the model when compared to BLiA and BRTracer, note that the build time depends on the number of documents. While BLiA and BRTracer use source code files (e.g., java class files), JINGO leverages changesets which are significantly more numerous compared to the number of classes, hence it is expected that JINGO requires more time to complete the initial build.

4.5.4 RQ4: Different types of content in bug reports

One of the key goals of JINGO is to address the different types of bug reports. Figure 8 contrasts the MAP values of JINGO and Corley's et al.'s approach when

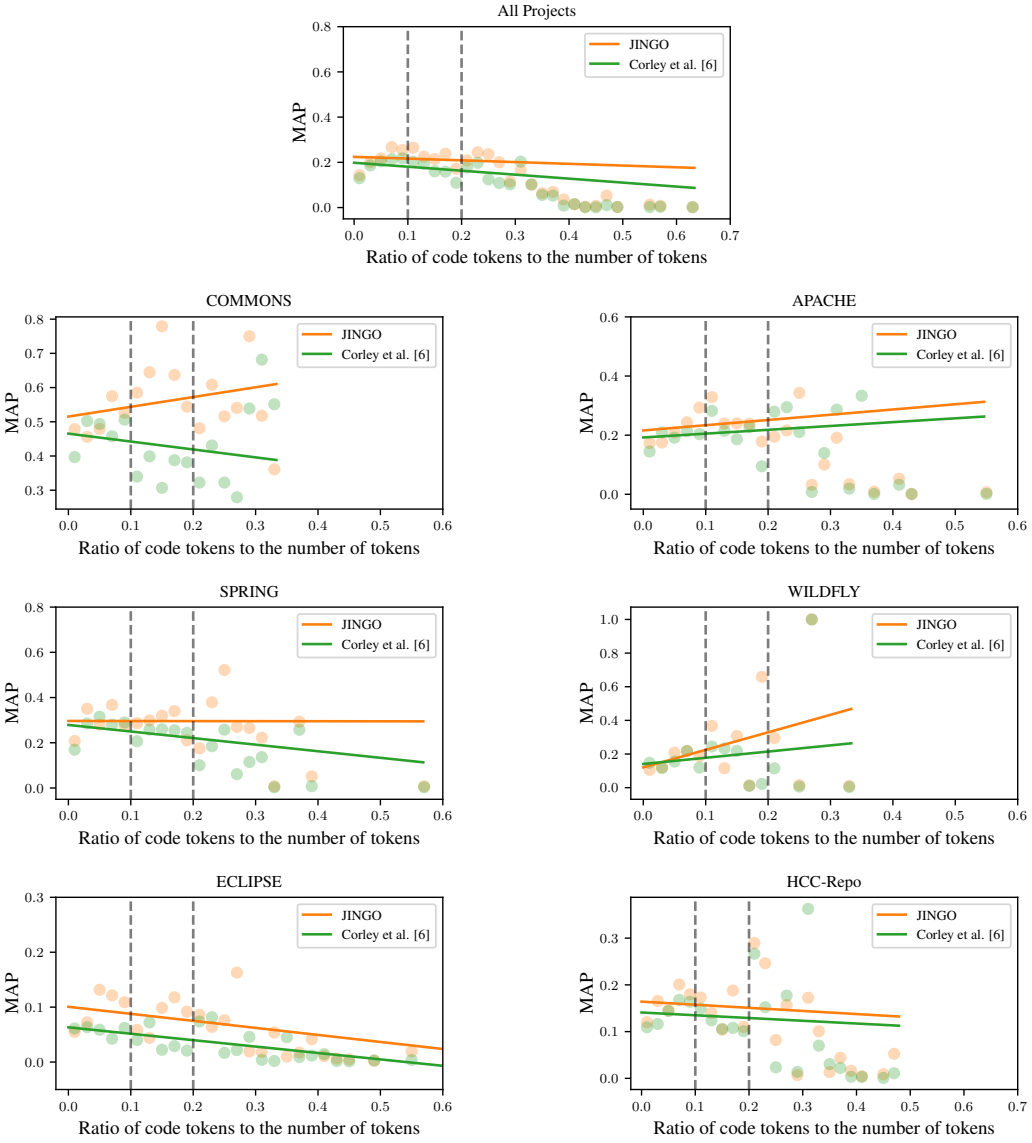


Fig. 8.: Performance of JINGO and Corley et al.'s [25] technique for bug reports containing different ratio of code tokens to the total number of tokens (x-axis). Performance is measured with MAP (y-axis) with horizontal lines depicting trend lines.

locating relevant files for bug reports with varying ratios of code tokens to the total number of tokens in the bug report. To ease readability, bug reports were grouped in bins of size 0.02 according to the ratio of code tokens. Each bug report bin corresponds

to one dot, with MAP value equal to the average of the MAP values of the bug reports in the bin. The best fit lines in Figure 8 further illustrate how the MAP results vary with different ratios of code tokens.

Across all bug reports, as shown in the top left of Figure 8, JINGO's best fit line is above the baseline, indicating that the technique is in general more effective. Moreover, JINGO provides more stable performance for different types of bug reports as its trend line is more flat when compared to the baseline's trend line. In other words, JINGO is less affected by the presence of code tokens in a bug report.

However, depending on the project group, we note different characteristics and scale of improvement. For instance, the COMMONS group exhibits the highest improvement in MAP values with increasing difference between JINGO and the baseline's performance as the ratio of code tokens rises. We observe similar characteristics for the APACHE, SPRING, and WILDFLY groups, however the gap between JINGO and the baseline effectiveness is smaller when compared to results for the COMMONS group. The ECLIPSE group presents significantly different characteristics for JINGO. Unlike for the other groups, the MAP trend line is decreasing with increases in the ratio of code tokens. We noticed that the bug reports in the ECLIPSE projects often included code snippets from the users' projects, which had little to no connection to the code base, that could misguide the JINGO model. Yet another interesting characteristic can be noted for the HCC-Repo group, where JINGO trend line is consistently above the baseline's trend line. Given the fact that the projects in HCC-Repo group have long and rich development history which includes numerous previously fixed bug reports (see Table 8), this result indicates that JINGO can effectively leverage historical data to boost the performance for all types of bug reports.

In Table 7 we show the results for the manually annotated set of bug reports, again observing improvements in JINGO over Corley et al.'s JIT-IRBL technique.

Table 7.: Performance on different bug report types from a manually annotated set ($N = 322$).

Bug Report Type	Num.	MRR JINGO	MRR [25]	MAP JINGO	MAP [25]
Code References (CR)	165	0.455	0.365	0.376	0.266
Shared Terms (ST)	100	0.257	0.272	0.180	0.177
Natural Language (NL)	57	0.226	0.178	0.143	0.091

We also observe that the techniques overall do best with CR bug reports, followed with ST, and struggle the most with NL. JINGO provides observable improvements in the CR and NL categories, but performs on par with the other technique on ST bug reports.

4.5.5 Discussion

In this section, we detail a few salient observations resulting from evaluating the four research questions. The evaluation of RQ1 indicates that JINGO presents an improvement over a prior technique that proposed an updatable model of the software based on changesets (i.e., JIT-IRBL). More specifically, according to RQ4, the primary reason for this improvement is two fold, (1) JINGO performs better on bug reports with high number of code terms, including exact references to the methods and classes of interest, (i.e., code references); and (2) JINGO shows improvements on bug reports with high level of abstraction that do not mention any of the related code references. We believe the first reason is due to heuristics JINGO uses, such as higher weighting program element names, while the second reason is due to the two-level hierarchical architecture of JINGO. However, in the evaluation of RQ3, we observe that the accuracy of JINGO is not at the level of static (i.e., non JIT) approaches, in particular, those based on the vector space model.

JINGO achieves the goal of performing fast updates for newly arriving data, according to RQ2. The static VSM-based approaches we contrasted with in RQ3 are significantly faster to build than JINGO, but are 20x slower in model updating, as these models are not designed to be updatable, hence they have to be rebuilt.

According to our evaluation of RQ4, the most abstract (Natural Language category) bug reports still perform poorly in absolute terms, i.e., MRR=0.226 and MAP=0.143, producing weaker results than the two other categories of bug reports we identified, despite the heavy emphasis of JINGO on capturing abstract semantics via its two-layer architecture. Clearly, more work is needed to improve how bug localization techniques perform for this category of bug reports. Future research efforts should identify such bug reports explicitly as they are significantly fewer than the other categories (roughly 1/3 or 1/2 of the other categories in our randomly collected sample), while localizing them arguably provides the greatest value to end users.

4.5.6 Threats to Validity

The results of the study suffer from several threats to their validity. A key threat to the internal validity of our study are the specific parameter choices we used to build our model. Probabilistic models like ours are particularly sensitive to such parameters [94]. While, to mitigate this threat we employed extensive hyperparameter optimization using a separate dataset, it is clear that this threat can still be impacting our study.

Leveraging changesets for bug localization pose another threat due to possible noise that can be introduced by tangled, split, or refactoring changesets [50, 51]. However, as long as such noisy changesets are in the minority relative to ones that reflect semantically related modifications, probabilistic techniques like LDA are likely to still produce a reasonable representation that can model how source code evolves

over time [52].

Another threat is in potential biases affecting our evaluation datasets, such as incorrect ground truth, and misclassified or already localized bug reports [64]. The first two biases have a potential to negatively affect the performance of JINGO as they introduce noise in the translation matrix, while the last bias can spuriously increase the results by having localization hints present. To mitigate the first threat, we followed experimental procedures used by other researchers, aiming in most cases to err on the side of caution by adopting choices that produce low false positives when identifying source code files related to a bug [84]. To mitigate the risk of using issues misclassified as bug reports, before building HCC-Repo dataset one of the authors manually inspected each project to ensure the quality of issue labeling, and identified labels referring to actual bugs. Although our efforts cannot completely remove those two biases, as observed by Kochhar et al. [64] they are typically not significant, hence, considering the size of our dataset, they should not have a significant effect. As for the already localized bug reports, we did not exclude them since one of our goals was to observe how bug localization performance changes for different types of bug reports. However, to present a complete picture, we included results for a manually annotated subset of bug reports with and without localization hints (Table 7).

A key threat to external validity is that we applied the bug localization technique only on a limited number of bugs, which primarily reflect popular open source Java projects. A mitigating factor is the evaluation with the large number of projects curated by the Bench4BL benchmark [84]. Additionally, this benchmark has also been applied to prior bug localization studies. Another threat to external validity is in the chosen evaluation metrics, which may not directly correspond to user satisfaction with our bug localization technique [12], impacting the generalizability and validity of the reported results. We mitigate this threat by evaluating our approach with high-

quality datasets and well-known metrics, which continue to be used by academia and industry to measure the performance of IR techniques.

4.6 Conclusions

In this chapter, we propose a novel technique toward the just-in-time bug localization problem, which becomes increasingly relevant as software size grows rapidly. JINGO is based on two Online LDAs that separately model changesets and bug reports, and uses the translation matrix, trained on previously observed pairs of bug reports and fixing changesets, to translate between the probabilistic spaces of the two LDA models. By adopting online variant of LDA model, JINGO can be updated with newly arriving data (i.e. bug reports or changesets), while using separate models for bug reports and changesets allows it to adapt the prediction to the content of a bug report.

The experimental evaluation was conducted on 56 software project, with a total of 2,999 bug reports. The results indicate that the proposed approach outperforms a previously reported topic-based baseline, especially when considering different types of bug reports. Given that JINGO incorporates historical data about bug fixes, it is more robust for bug reports that lack localization hints. On the other hand, the retrieval accuracy of JINGO is still superseded by VSM-based approaches, however, they are also significantly slower as they do not support an update procedure and have to be rebuilt periodically.

Overall, these results indicate the importance of leveraging historical data for improved retrieval accuracy in the case of bug reports containing only a high-level description that is challenging to map to source code files without prior knowledge. As a future work, JINGO could possibly be improved by leveraging noise filtering techniques to improve the quality of data used to construct the translation matrix. On

the other hand, the main strength of JINGO, building an abstract data representation based on topic distributions, is also its main drawback for bug reports that include localization hints, since single terms are not explicitly captured by the model. To address that, future work could investigate how to preserve fine-grain data, such as single terms, when building a high-level data representation.

CHAPTER 5

FAST CHANGESET-BASED BUG LOCALIZATION WITH BERT

As industry is increasingly attempting to use bug localization to aid developers in their daily work, specific requirements of the problem for modern use are coming to the forefront [11]. One key characteristic found beneficial in modern software projects is bug-inducing changeset retrieval. A bug-inducing changeset is one where the bug was initially introduced into the repository. Retrieving such changesets leads to faster bug repair, as they contain related parts of the code that were changed together, which makes fixing the bug easier. However, retrieving bug-inducing changesets with high accuracy is more challenging than retrieving source code elements, as changesets are typically much more numerous than software classes or files. In recent years, numerous popular natural language processing tasks (e.g., question answering, machine translation) have all observed improved performance when using neural network architectures based on transformers. These transformer-based models are typically applied via transfer learning, by first pre-training them on a very large corpus and then fine tuning them to the specific task using a much smaller dataset. Transformer-based models pre-trained on large software engineering corpora (e.g., StackOverflow, GitHub) are now becoming available [28], with the potential to improve software engineering tasks like bug localization. However, given the fact that these models consist of many neural layers and require heavy computation, measuring relatedness between the bug report and the software artifact quickly becomes expensive, especially when the search space consist of changesets.

In this chapter, we apply a BERT-based approach the problem of changeset-based bug localization with the goal of improved retrieval quality, especially on bug reports where straightforward textual similarity would not suffice. We describe an architecture for IR that leverages BERT without compromising retrieval speed and response time. In addition, we examine a number of design decisions that can be beneficial in leveraging BERT-like models for bug localization, including how best to encode changesets and their unique structure.

5.1 Industrial requirements for bug localization

Applying a bug localization tool in an industrial environment is a challenging task given that such a tool needs to address needs and requirements that are often unknown or not considered by researchers while building a tool. In this section, we list and discuss the specific constraints of the bug localization problem that we aim to address, which are based on a recent survey of industry practitioners and the problem requirements observed at a large software enterprise [11, 53]. Our focus is a bug localization technique that: 1) focuses on retrieving changesets; 2) aims to capture semantics and can be applied to bug reports that do not share terms with the relevant parts of the code base; and 3) quickly retrieves results for a newly created bug report.

Requirement#1: Localizing changesets [11]. Over the years, a large body of research has been dedicated to locating source code files (or classes) relevant to a bug report [54, 29, 91, 56, 31, 25]. However, recent studies have pointed out that bug localization at the level of source code files still requires significant effort by software developers in order to locate relevant code within large files [24, 11, 53]. Adjusting for this finding, researchers shifted their efforts towards more fine grained code elements, such as file segments [91] and methods [88, 99, 100], which introduce new sets of

challenges such as difficulty in selecting optimal segment size and large methods that still require effort to examine. More recently, there has been a growing interest in changeset retrieval [25, 24, 26, 61] for bug localization as changesets have several unique properties that make them convenient to developers aiming to fix a bug. First, they inherently capture lines of code that are related to each other within the context of a modification, while the changeset log summarize the goal of the modification. Second, when locating changesets, we can retrieve not only the modified portion of the code, but identify a software developer that committed the modification in the first place, therefore easing the bug triaging process. Finally, changesets allow for straightforward context-aware division into a set of hunks, i.e., a set of changes in one area of the file. Hunks are usually convenient to read for developers and allow for easy detection of changes with no semantic value (e.g., changes only in whitespace).

Requirement#2: Leveraging semantics of input documents [11, 28]. Understanding the meaning behind the source code can be essential to discern whether the code is relevant to a bug report. However, while bug reports are expressed in natural language, source code is highly structured and typically use a fixed, repetitive vocabulary, that can easily obscure the actual functionality it implements. Moreover, given that multiple developers are involved in maintaining the code base, it becomes affected with unique coding and naming patterns, which pose a significant challenge to traditional IR systems based solely on token similarity [27]. To address that, there has been a growing interest in building semantically rich document representations [27, 26, 15, 79, 101, 11, 102]. Transformer-based models in general, and BERT in particular, are currently one of the most exciting deep learning techniques achieving broad improvements across a variety of text-based tasks. The main strength of BERT-like models is in building a token representation based on bidirectional capture of contextual information encoded in the preceding and succeeding tokens, which leads to richer

semantics that is more likely to detect related pairs of bug report and changesets that do not share terms. Prior generations of word embeddings, e.g., word2vec [38] and GloVe [39], which have been frequently applied on software engineering tasks [103], do not use word context at inference time, i.e., each token maps to a unique vector regardless of the surrounding text.

Requirement#3: Fast retrieval in a large search space [104]. Retrieving bug-inducing changesets requires computing similarity between a bug report of interest and all changesets committed to a repository up to the present point in time. Given that modern software evolves rapidly, resulting in large source code repositories with numerous commits [105, 106], it is impractical to compute pair-wise similarity due to the large search space. This is especially the case if computing the similarity measure itself is expensive. Though deep learning models provide state-of-the-art accuracy, they typically require more computational resources than token-based techniques, which emphasizes the need for a bug localization technique to limit the search space in order to improve performance without compromising accuracy.

In order to address the above problem constraints, this work investigates the use of a BERT model towards bug localization with changesets as a primary data granularity, which is also preferred by practitioners' (Req#1)[107]. We specifically selected BERT as it is the state of the art in semantics modeling and extracting contextual information, therefore addressing (Req#2). Finally, to ensure that our approach is applicable to large, industry scale repositories (§2.3), we introduce Fast Bug Localization BERT (FBL-BERT), which reduces the search space, such that only promising candidate changesets are considered for neural re-ranking with BERT. In addition, FBL-BERT encodes a bug report and a changeset separately, allowing to compute changeset representations offline and reduce the computational effort per

bug report at retrieval time.

5.2 Approach

In this section, we first introduce BERT and how it can be used to enhance bug localization, followed by FBL-BERT, which is aimed to significantly improve BERT's retrieval speed. Last, we outline different choices for how to encode changesets with BERT in order to better preserve their semantics and provide quality matches with bug reports.

5.2.1 BERT for bug localization

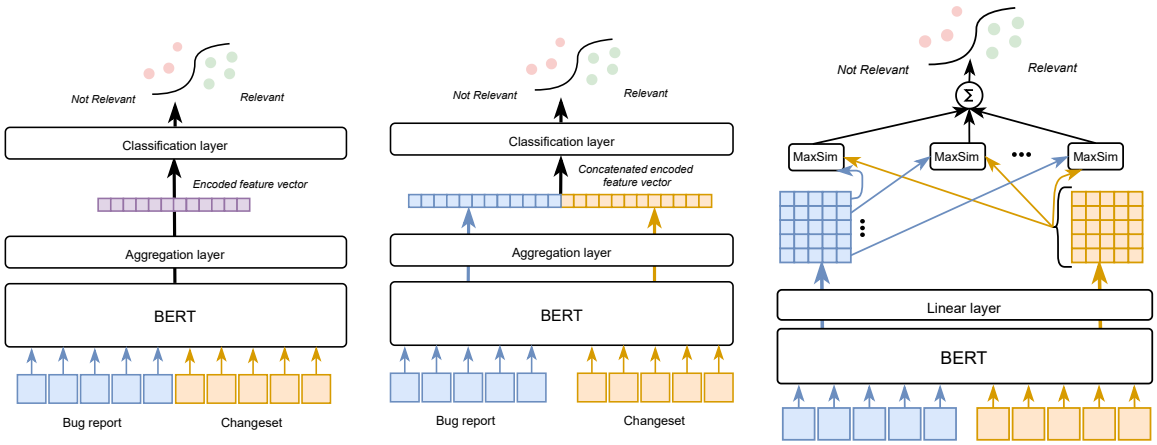
The architecture of BERT consists of multiple layers of transformer-encoders, which are an abstraction aimed at modeling sequential data that utilizes self-attention; the notion of attention is to weight specific terms in the sequence differently, i.e., encoding a stronger relationship from each term in the sequence to the remaining most semantically relevant terms. As pointed out by Mills et al. [10], retrieval techniques for bug localization can be significantly improved with intelligent query construction, i.e., by carefully choosing which parts of the bug report to use for comparison. Therefore, leveraging a model that uses attention to emphasize certain word relationships has the potential to significantly improve upon prior state-of-the-art bug localization techniques.

Using a BERT model for bug localization (or other similar purposes) involves three essential steps: (1) pre-training the model with a large corpus of general software engineering-related data, (2) fine tuning the BERT model for bug localization, and finally, after BERT has been completely trained, (3) retrieving relevant bug-inducing changesets for a newly reported bug.

During pre-training, BERT uses massive corpora of relevant text to build a lan-

guage model for a specific domain, e.g., software development. Given that this step requires a significant amount of data and computational resources, a common choice is to re-use a pre-trained BERT model, when available. In the fine tuning step, BERT updates the general data representation with respect to a specific downstream task (e.g., bug localization) given a much smaller, task-specific dataset. More precisely, fine tuning a BERT model occurs by adding an additional layer (e.g., a classification layer) to the pre-trained BERT model. This task-specific layer takes the output of BERT as input and represents the part of the model that is primarily trained during fine tuning, though BERT's internal weights are also updated in the process. In most scenarios, fine tuning can be completed faster and with much less computational resources than pre-training. Since our goal is locating bug-inducing changesets, a natural choice for a task-specific dataset consists of bug reports and their inducing changesets. A key design choice at this stage is how to connect BERT with the additional task-specific neural network layer. Given an input document, BERT encodes each word in the document with a vector, i.e., for each input document, the output of the BERT model is an embedding matrix of size $|d|$ by v_{len}, where $|d|$ represents the number of words in the document and v_{len} the length of a BERT vector; typically $v_{len} = 728$. The most common approach when retrieving BERT-encoded documents is to aggregate the embedding matrix across words through average or summation, which produces a single vector as output. Using such an aggregate representation of a document allows for faster processing and easier comparison between pairs of documents. However, as pointed by Sachdev et al.[108], this simple aggregation strategy leads to a dissipative data representation that has the potential to negatively affect retrieval performance. In the next section, we present an alternative strategy that takes advantage of the full matrix to encode input data.

In the simplest changeset retrieval scenario, presented in Fig. 9a, each newly



(a) Single BERT [26, 109, 110]

(b) Siamese BERT [26, 111]

(c) FBL-BERT [112]

Fig. 9.: BERT-based architectures for changesets retrieval.

arriving bug report is concatenated with every changeset in the project history. Subsequently, they are processed by BERT, producing an embedding matrix, which is transformed to a vector by an aggregation layer. Finally, the vector is passed into a classification layer that produces a relevancy score between a bug report and a changeset. Changesets are ordered based on their scores to produce a ranked result set. This type of BERT architecture for information retrieval is often referred to as Single BERT [26, 109, 110]. In an alternative retrieval architecture, called Siamese BERT [26, 111] and depicted in Fig. 9b, the bug report and the changeset are processed in parallel, first through BERT and then through an aggregation layer. As a result, bug reports and changesets are transformed into independent vectors that are subsequently concatenated and fed into the classification layer to produce a relevance score. The advantage of Siamese BERT over Single BERT is that Siamese BERT enables pre-computing changeset representations offline since changesets are not required to be concatenated with a bug report for retrieval. However, Siamese BERT still requires comparing a bug report to each changeset, which incurs significant

retrieval delay in the case of large number of changesets.

5.2.2 Fast Bug Localization BERT

The FBL-BERT architecture, based on ColBERT by Khattab et al. [112], eschews aggregation of the embedding matrix, and instead builds a relevance score by leveraging the whole matrix, resulting in a more complete, fine grained comparison. More specifically, a bug report br and a changeset c are separately processed by BERT creating embedding matrices E_{br} and E_c, respectively. To compute the relevance score between E_{br} and E_c, for each word embedding in the bug report $v_{br} \in E_{br}$, we find the maximum cosine similarity across word embeddings of the changeset $v_c \in E_c$, and combine the maximum cosine similarities via summation as illustrated in Fig. 9c. As a result, the model learns how to associate words from a bug report with tokens in a changeset, taking into account the context in which they appear. To account for the two different types of data we process, i.e., bug reports and changesets, we modify ColBERT by increasing the numbers of BERT encoder layers taken to the linear layer. More specifically, while ColBERT uses the output of the last BERT encoder, we take the output of the last 4 encoders (as recommended by [40]). This modification is dictated by prior studies observing that that different layers of BERT encode different granularity of semantic information [40, 113, 114]. Note that the linear layer in FBL-BERT is not equivalent to the aggregation layer discussed before, but is used to reduce the size of word embeddings produced by BERT, retaining all word embeddings in a compressed form for faster downstream processing.

There are several benefits that make the FBL-BERT architecture particularly applicable to our problem. First, the model purposely avoids joint document encoding, as in Single BERT, delaying interaction between a bug report and a changeset to facilitate off-line encoding of changesets. Moreover, by using computationally cheap,

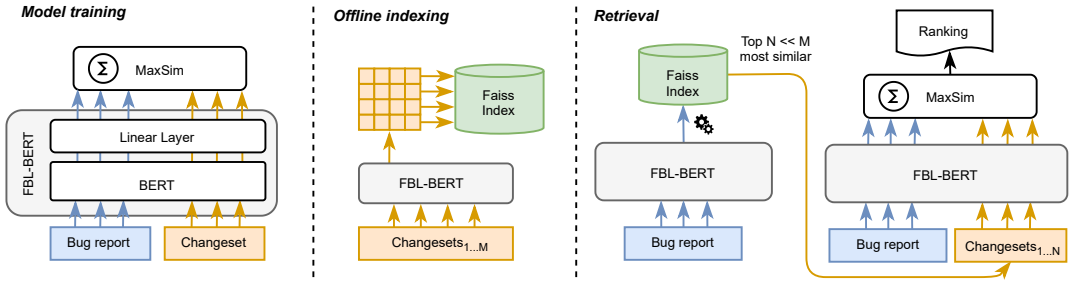


Fig. 10.: FBL-BERT for changeset-based bug localization pipeline.

yet efficient, maximum similarity summation as a scoring operator instead of a more complex strategy, such as the classification layer in Siamese BERT, the processing time for a query is reduced. Finally, given that the relevance score computation is isolated and relies solely on maximum similarity, it is possible to utilize efficient vector similarity algorithms to reduce the search space of all M changesets by identifying top-N changesets, $N << M$, that are similar to a new bug report, and subsequently re-rank only the top-N subset.

To clarify how FBL-BERT operates for changeset-based bug localization, consider the pipeline depicted in Fig. 10. First, as shown in the Model Training section of Fig. 10, the FBL-BERT model is fine tuned on a project-specific dataset consisting of bug reports and bug-inducing changesets. In the next step (Offline Indexing), *all changesets* in the project repository are encoded via FBL-BERT and stored in an index supporting efficient vector-similarity search. For this purpose, we use an IVFPQ (InVerted File with Product Quantization) index, implemented in the Faiss library [115]. The IVFPQ index uses the k-means algorithm to partition the embedding space into P (e.g., $P = 300$) partitions, and subsequently assigns each word embedding to its nearest cluster. To facilitate efficient search, when a query is issued, the query is first compared against the partitions' centroids to locate the nearest partitions, and then the search continues to the instance-level only within those. Note

that the Faiss index contains *word* embeddings across *all* changesets. After completion of this step, the retrieval system is ready to be deployed. When a new bug report arrives, it is first encoded via FBL-BERT producing an embedding matrix. Next, for each word embedding in the embedding matrix, we query the Faiss index to identify the N' most similar embeddings across all changesets embeddings stored in the Faiss index. Since among N' most similar embeddings some may point to the same changeset, in the end we obtain a total of N unique candidate changesets. Finally, we use FBL-BERT to re-rank the candidate changesets and produce the final ranking.

5.2.3 Changesets encoding strategies

Software evolution over time is recorded in a repository as a time-ordered sequence of changesets. Each changeset consists of a log message, providing a short rationale explaining the goal of the modification, and a set of source code changes. Depending on the version control system and `diff` algorithm used in the software project, the representation of source code changes can vary. In this paper, we focus on the format that is the output of the `git diff` command, in which added lines of code are annotated with `+`, removed lines with `-`, and all modified lines are surrounded by 3 lines of contextual, unchanged lines. While there exist more advanced tree-based code differencing algorithms (e.g., GumTreeDiff [116]), providing detailed code-change information to a machine learning model may affect the model negatively [117], hence we opt for a text-based approach. Changesets can encapsulate code changes across one or multiple source code files, and modifications to each file can be divided into *hunks* - groups of modified (added or removed) lines surrounded by unchanged (context) lines. Given this specific formatting, we explore how best to utilize changesets' properties to construct BERT input from two perspectives: (1) encoding characteristics of code modifications, such as additions or removals; and (2)

levels of granularity in a changeset.

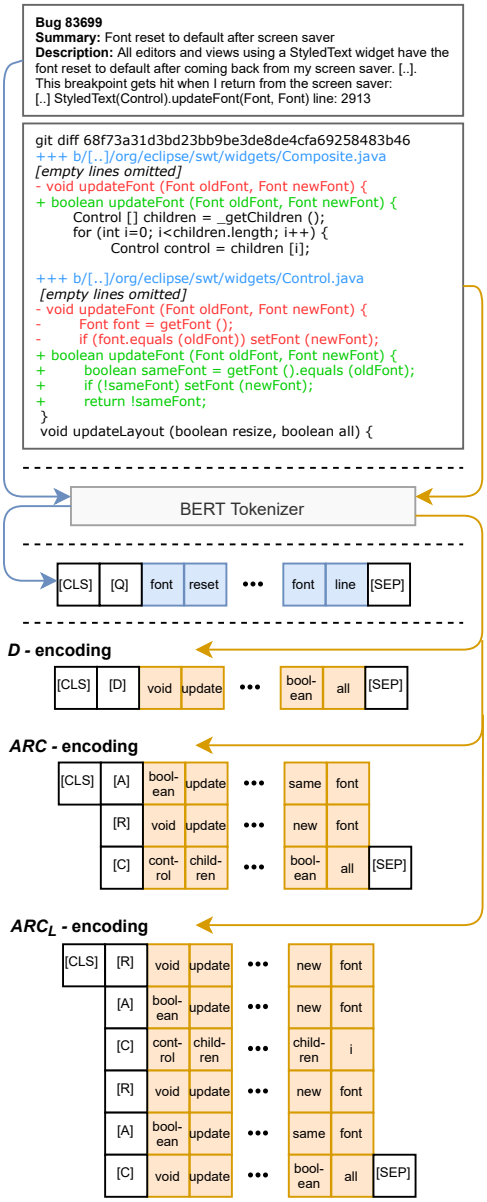


Fig. 11.: Changeset encoding strategies.

Input provided to BERT models is required to follow certain rules. First, a document (e.g., a changeset or a bug report) needs to be tokenized and each token replaced by its unique token id. Pre-trained BERT models supply their own BERT tokenizers,

that are optimized towards the corpus on which the model is pre-trained. BERT tokenizers are trained using the WordPiece algorithm [118]. The main advantage of BERT tokenizers is in avoiding out-of-vocabulary words by dividing unknown words to their largest subwords present in the vocabulary, which is likely to be beneficial in our setting, as software projects can have very specific vocabularies unlikely to be observed elsewhere [28]. Secondly, BERT uses a pre-defined set of special tokens. In general, due to how BERT is trained (more details in [40]), the model requires that each token sequence starts with special classification token `[CLS]` and ends with separator token `[SEP]`, while other special tokens, such as padding `[PAD]` are used if and when necessary. Special tokens can convey information about the structure of data allowing BERT to differentiate between parts of the input, hence we explore how special tokens can be best utilized to encode changesets. To this end, we propose the following encoding strategies, depicted in Fig. 11.

D: A changeset is considered a single document that is feed into the model. To inform the model that a changeset sequence begins, we define and pre-append the special token `[D]` at the beginning of the code sequence. Since this strategy does not utilize specific characteristics of a code change, it serves as a baseline to compare against other strategies.

ARC: In this encoding, a changeset is split into lines, and the lines are subsequently grouped based on whether they are added, removed or provide context, as indicated by their initial character: `+` for added, `-` for removed, and an empty space for context lines. The lines in each group are concatenated to create a sequence to which we pre-append a special token: `[A]` for the sequence of added lines, `[R]` for the sequence of removed lines and `[C]` for the sequence of context lines. Finally, all the sequences are concatenated together to create an input for the model. By grouping different

parts of changesets based on their characteristics, we aim to investigate whether any particular type of modification is more beneficial than the other. With the ARC strategy the model is given an opportunity to learn how to combine information of different types and, if necessary, decide to disregard a portion of it if it poorly affects performance.

$\mathbf{ARC_L}$: Similarly as in ARC, a changeset is divided into lines, however ARC_L encoding does not group the lines. Instead, it preserves the ordering of lines within a changeset, such that special tokens `[A]`, `[R]`, or `[C]` are pre-appended wherever type of modification changes. While this strategy results in more accurate data representation, compared to ARC, ARC_L is also more challenging for the model, since the special tokens occur multiple times and in several places.

Given that a bug report and a changeset are encoded separately, the model has to differentiate between these two types of documents. To this end, when encoding a bug report, we define a special token `[Q]` that is pre-appended to the query, i.e., the bug report.

Another dimension in choosing how to best encode changesets is related to their granularity, i.e., using entire changesets or separating a changeset to a file- or hunk-level. Leveraging hunks as the primary data dimension in an IR model brings several advantages. First, bugs have been observed to be typically caused by small pieces of code [88, 91], thus the inherent fine granularity of hunks makes them less susceptible to noise when compared to whole source code files [24]. Second, dividing changesets into hunks alleviates issues caused by tangled commits [119]. Given the fact that hunks are typically small and concentrate on an enclosed portion of the code, BERT is not affected by long-range token dependencies, which is a problem typically affecting source code [28]. Finally, shorter input documents are less likely to exceed the

maximum sequence length accepted by BERT, while longer documents have to be truncated, which may negatively affect the results. However, despite easily accessible smaller data granularity within a changeset, to date, most of the efforts are focused on leveraging entire changesets [26, 5, 11].

5.3 Experimental evaluation

5.3.1 Research questions

RQ1: *How effective is FBL-BERT when compared to (1) state-of-the-art techniques based on the VSM, and (2) related BERT-based architectures?*

The main opportunity in using FBL-BERT is in incorporating additional context and semantics when retrieving bug-inducing changesets, which should provide improvements in accuracy over the state-of-the-art, especially for bug reports that provide high level bug descriptions and lack explicit localization hints. Researchers have identified that a non-trivial amount of bug reports already contain localization hints, i.e., they mention the class or method names relevant to fixing the bug, and some recent approaches for bug localization argue that only bug reports that lack extensive localization hints should be considered in evaluation [30]. We follow the methodology proposed by Kochhar et al. [64] to categorize bug reports into 3 groups based on the completeness of localization hints they provide and evaluate the performance for each bug report group separately. We also investigate how the runtime performance of FBL-BERT, which utilizes fine grained matching, compares to other BERT-based architectures that rely on embedding aggregation and perform retrieval across the entire search space. As baselines, we use (1) Locus [24], a state-of-the-art approach based on VSM that locates bug-inducing changesets, and (2) TBERT-Single and TBERT-Siamese [26] approaches that utilize aggregated BERT-based representations that

have recently been proposed for software engineering.

RQ2: *Which changeset encoding strategy is the most profitable? Are there advantages to using hunks, changeset-files or entire changesets as the primary data dimension?*

In this RQ, we first investigate whether encoding information about the type of modification in each line of a changeset can increase the performance of the FBL-BERT model. We evaluate two alternatives to encode changesets semantics, ARC, ARC_{L}, and a baseline approach, D, which disregards change-related information. Second, we investigate how granularity of the input data affects the model performance and what are the benefits and challenges of leveraging changesets, changeset-files, or hunks in our model. To answer this RQ, we fine tune FBL-BERT separately for each of the encoding strategies and with each input data granularity, resulting in 9 evaluation configurations per software project, measuring the model's performance in retrieving relevant changesets.

5.3.2 Dataset and baselines

To answer the RQs, we leverage the dataset of bugs and their inducing changesets collected and manually validated by Wen et al. [24]; manually validated datasets remove the error that can be introduced by the SZZ algorithm that maps the bug fixing to the inducing commit [120]. This dataset includes 6 software projects, namely AspectJ, JDT, PDE, SWT, Tomcat and ZXing (descriptive statistics are presented in Table 8). To create a training set for each project, we selected the first half of project's pairs of bug reports and bug-inducing changesets, ordered by bug opening date, as a training set, and left the remaining half as a test set. For each pair in the training sets, we also create a negative sample by randomly choosing a code change which does not belong to the inducing changeset, essentially forming triplets of bug

Table 8.: Evaluation datasets for FBL-BERT.

	#Bugs	#Changesets	#Changeset-files	#Hunks
AspectJ	200	2,939	14,030	23,446
JDT	94	13,860	58,619	150,630
PDE	60	9,419	42,303	100,373
SWT	90	10,206	25,666	69,833
Tomcat	193	10,034	30,866	72,134
ZXing	20	843	2,846	6,165

report, bug-inducing changeset, not bug-inducing changeset. We experimented with choosing negative samples by selecting a syntactically similar changeset that was not bug-inducing but we did not observe a significant change in retrieval accuracy. As this type of generating negative samples incurred substantial computational cost to gather, we opted to use random sampling. Finally, for each project we obtained a balanced training set with equal number of positive and negative examples. Note that although training sets do not include all available code changes, during bug localization the model performs retrieval across *all* code changes available for a specific project. (as explained in Section 5.2.1). To study the impact of different changeset data granularity on the BERT-based models, we created a separate dataset for each type of granularity, i.e., changesets, changeset-files and hunks. To this end, for changeset-file and hunk granularity, we divide the bug-inducing changeset to file- or hunk-level code changes, such that one bug report creates multiple pairs with files or hunks from its respective inducing changeset.

We compare the performance of the proposed model with Locus [24], which is an unsupervised model that utilizes hunk-level granularity and the VSM to locate relevant changesets based on the maximum similarity score obtained between a bug report, a hunk, and a log message. Note that FBL-BERT does not use log messages

as our goal is to explore mapping from natural language in a bug report to code changes. While well written log messages can have a positive impact on the results by boosting the scores for some changesets, not all relevant code changes are accompanied by logs of good quality [44, 121]. As a second set of baselines, we employ TBERT architectures for software artifacts retrieval recently proposed by Lin et al. [26]. Out of the three architectures investigated by Lin et al., we selected TBERT-Single and TBERT-Siamese as our baselines, rejecting TBERT-Twin, since its performance in terms of accuracy and time was significantly surpassed by the two others. In general, both of these architectures are fairly similar to those presented in Fig. 9 with an exception of using more advanced embedding aggregation operators [26].

5.3.3 Experiment setup

The experiments were conducted on a server with Dual 12-core 3.2GHz Intel Xeon and utilized 1 NVIDIA Tesla V100 with 32GB RAM memory running on CUDA version 10.1. To implement our model, we used PyTorch v.1.7.1, HuggingFace library v.4.3.2, and Faiss v.1.6.5 with GPU support. Since pre-training is a computationally expensive task and requires a huge dataset, we decided to use an available pre-trained BERT model, BERTOverflow [81]. BERTOverflow is trained on StackOverflow data, hence it contains a mixture of code snippets and natural language descriptions, which is logical for the bug localization task that operates on both code and natural language. We fine tuned our BERT model and TBERT baselines for 4 epochs with batches of size 16 and a learning rate of 3E-06 [40]. Based on the average number of tokens in bug reports, hunks, changeset-files and changesets across the evaluation projects, we set the maximum length limit to 256, 256, 512, and 512 respectively. All input documents are truncated or padded to their respective length limit. For the Faiss index, we set the number of partitions to 320 and retrieved a total of 1000

Table 10.: Mean Reciprocal Rank (MRR) of changeset-based BL techniques for different types of bug reports.

Technique	Granularity	Bug report type				
		BL_{NL} *n=151*	BL_{PL} *n=75*	BL_{FL} *n=105*	BL_{NL+PL} *n=226*	All BRs *n=331*
Locus	Hunks	0.235	0.302	**0.452**	0.258	0.319
TBERT-Single	Changesets	0.119	0.213	0.136	0.150	0.146
	Change. files	0.274	0.469	0.299	0.339	0.326
	Hunks	0.268	0.429	0.273	0.321	0.306
TBERT-Siamese	Changesets	0.125	0.256	0.080	0.168	0.140
	Change. files	0.263	0.424	0.200	0.316	0.279
	Hunks	0.236	0.333	0.171	0.269	0.238
FBL-BERT	Changesets	0.076	0.114	0.113	0.089	0.096
	Change. files	**0.303**	0.441	0.294	0.349	0.331
	Hunks	0.290	**0.509**	0.338	**0.363**	**0.355**

changesets for re-ranking with FBL-BERT [112]. In the case of Locus, we set the model parameters to $\lambda = 5$ and $\beta_2 = 0.2$, indicated by the authors to provide the highest performance.

5.4 Results

5.4.1 RQ1: Retrieval performance

Retrieval accuracy. Table 10 contrasts the retrieval performance of the FBL-BERT model against the baseline approaches for three different types of bug reports: not localized, partially localized, or fully localized. If a bug report has no mentions of relevant classes, it is classified as not localized (BR_{NL}); when some of the relevant classes appear in the report, the bug is categorized as partially localized (BR_{PL}); and if all relevant class names are provided, the bug report is fully localized (BR_{FL}) [64]. Note that in the case of FBL-BERT, we use the results of the model trained with ARC_L encoding since, on average, it provides the best performance across the evalu-

ation projects, as shown in Section 5.4.2.

FBL-BERT outperforms Locus for BR_{NL} and BR_{PL} by 5.5% and 20.6% respectively, while in the case of BR_{FL}, Locus surpasses our approach by 11.4%. Given that Locus relies on more direct term matching between a bug report and a changeset, it makes intuitive sense that such a model performs best when localization hints are present in a bug report, and struggles in their absence (as indicated by lower MRR values for BR_{NL} and BR_{PL}). On the other hand, FBL-BERT utilizes higher-level association between bug reports and bug-introducing changesets, which can result in exact matches getting less emphasis. Interestingly, the highest improvement in retrieval accuracy is observed for BR_{PL} indicating that the model can effectively retrieve changesets based on partial clues by associating them with patterns learned from historical data.

The performance of both TBERT models and FBL-BERT improves when the models are trained and evaluated on hunks or changeset-files. Compared to leveraging changesets, across all bug reports FBL-BERT improves between 23.5%–25.9%, while the retrieval accuracy of TBERT-Single and TBERT-Siamese increases by 16%–18% and 9.8%–13.9% respectively. While this results indicate that leveraging fine grained data affects retrieval performance positively, it is important to note that the poor performance observed for changesets can be partially attributed to the input size limit of the BERT model (i.e., 512 tokens), which is more often exceed by changesets than hunks or changeset-files. More specifically, in our dataset truncation affects about 8% of hunks and 25% of changeset-files compared to 45% of changesets.

In general, FBL-BERT outperforms TBERT-Single and TBERT-Siamese by 4.9% and 7.6% respectively across all types of bug reports. Comparing the results of FBL-BERT trained on hunks to TBERT models trained on changeset-files, given that changeset-files provide on average the best performance for TBERT models, we note

varying difference in retrieval accuracy depending on the bug report type. In the case of BR_{NL}, FBL-BERT improves MRR score by only about 2% over TBERT models. For BR_{PL}, FBL-BERT improves by 4% and 8.5% over TBERT-Single and TBERT-Siamese, while for BR_{FL} the improvement is equal to 3.9% and 13.8% respectively. The larger gap in retrieval accuracy for BR_{PL} and BR_{FL} between FBL-BERT and TBERT models indicates the importance of token-level embedding matching, i.e., while TBERT uses aggregated embedding to represent and compare documents, the token-level embedding matching performed by FBL-BERT allows this model to better recognize the key code names presented in the bug report, which, in turn, translates to higher retrieval accuracy.

Retrieval time. One of the key desirable characteristics of FBL-BERT is to perform efficient retrieval across a large corpus. This would allow it to leverage fine grained data, such as changesets-files or hunks which were observed to provide the best retrieval accuracy, while maintaining reasonable retrieval delay. In Fig. 12, we compare the average retrieval time per bug report with respect to the increasing number of documents in the search space, i.e., changesets, changesets-files and hunks. In general, FBL-BERT retrieves relevant documents faster than both TBERT models with the retrieval time gap increasing as the search space grows. More specifically, TBERT-Single is the slowest model and requires about 50s to perform retrieval over a small number of documents (e.g., ZXing), and nearly 1000s(!) for a large project (e.g., JDT). TBERT-Siamese is significantly faster than TBERT-Single, and up to the search space of about 15K documents, it performs on-pair with FBL-BERT. However, after that point, retrieval time for TBERT-Siamese rises steadily to reach about 70s for the largest search space, while in the case of FBL-BERT the retrieval time is still just above 1s. By comparing the performance of FBL-BERT against TBERT models, it becomes evident that plain BERT-based models can quickly hit a retrieval delay

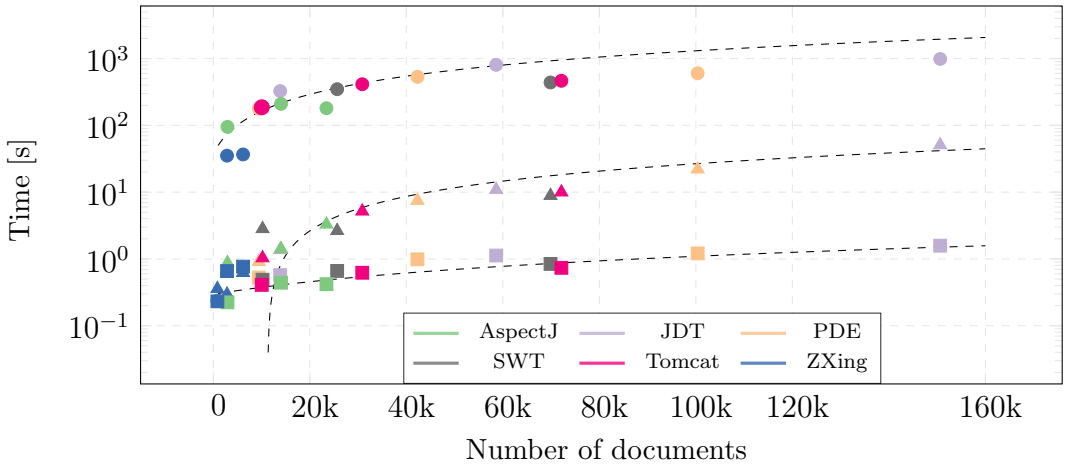


Fig. 12.: Average retrieval time per a bug report with different sizes of search space (• TBERT-Single, ▲ TBERT-Siamese, ■ FBL-BERT).

wall which makes them impractical to use. On the other hand, FBL-BERT scales up with respect to the search space size allowing to leverage fine grained data to increase retrieval accuracy without sacrificing model responsiveness.

Note that the observed speed improvement is the result of both FBL-BERT and FAISS. More specifically, the training objective of FBL-BERT (i.e., finding most similar embedding vectors) *enables* using vector similarity search (e.g., FAISS). As a consequence, FAISS can be used to retrieve the K best candidates ($K << N$, where N is #documents) with similar word-level embedding representations that are then re-ranked by FBL-BERT. By re-ranking only K documents, the search space becomes significantly reduced, hence decreasing the retrieval time. On the other hand, typical BERT-based pipelines (e.g., TBERT) concatenate bug reports and changesets, and use neural network layers to estimate a relevancy score. This approach precludes pruning the search space via FAISS, therefore, during retrieval a bug report has to be compared to all N documents, which in turn increases retrieval delay.

Error analysis. To gain more insight into factors that negatively affect the retrieval

accuracy of FBL-BERT, we manually analyzed the bug reports for which the model struggles the most. More specifically, we selected all bug reports where the bug-inducing hunk was ranked 50 or worse by FBL-BERT. This resulted in 20 bug reports ($BR_{NL} = 8$, $BR_{PL} = 3$, $BR_{FL} = 9$) that the authors independently analyzed, contrasting the retrieved hunks to the true bug-inducing hunks in order to devise a set of common issues causing low retrieval accuracy. The authors also examined the most similar terms (and their weights) for both the retrieved and gold set hunks, focusing specifically on the sources of largest differences between the two. Finally, the authors discussed their independent observations and agreed on three common error categories: *stack trace/code snippets*, *comments*, and *code tokens splitting*, where a single bug report can belong to more than one error category. We discuss each of these, in turn.

In 11 out of 20 bug reports, the difficulty to retrieve the correct hunk was caused by the presence of a code snippet or a stack trace in the bug report. Since code snippets and stack traces typically consists of multiple class names or code tokens, they have a potential to introduce noise through unrelated code names, which, in turn, can lead the model astray [122]. For 7 out of 20 bug reports, we noted that the model was misguided by source code comments present in the top-1 retrieved hunk. Since source code comments are formulated in natural language, a highly-contextual model like BERT tends to emphasize their similarity with the bug report as it is also expressed in natural language. For both of the above error categories, we believe that the wholesale removal of the problematic text (i.e, comments from code and code snippets and stack traces from bug reports) would negatively affect the model as it removes both relevant and irrelevant information. Hence, researchers should explore strategies to treat this data separately, perhaps by encoding their content within BERT with special tokens akin to the ARC and ARC_{L} strategies we discuss in this

paper.

Finally, for 5 of the bug reports, FBL-BERT failed due to spurious matches in code tokens that were split into sub-tokens during preprocessing. One of the previously observed strengths of BERT is in using the WordPiece algorithm to avoid the out-of-vocabulary problem by splitting unseen tokens into the largest sub-tokens that are part of the BERT vocabulary [123]. Since source code identifier names are typically project-specific words, they do not occur in the pre-trained vocabulary, hence they are often split by WordPiece (e.g., ManagerServlet → manager, ##servlet). The sub-tokens can then spuriously match other terms, including sub-tokens from other split identifiers, but not the whole, unsplit term. Researchers in the biomedical domain recognized the same issue affecting medical terms and proposed domain-specific BERT adaptations [124, 125, 126].

Table 11.: Retrieval performance for different configurations of FBL-BERT.

	#Bugs	MRR	MAP	P@1	P@3	P@5	MRR	MAP	P@1	P@3	P@5	MRR	MAP	P@1	P@3	P@5
Changesets				D					ARC					ARC_L		
AspectJ	104	0.053	0.032	0.029	0.024	0.037	0.107	0.061	0.058	0.080	0.083	0.070	0.042	0.029	0.045	0.044
JDT	47	0.097	0.014	0.043	0.028	0.021	0.118	0.160	0.064	0.043	0.030	0.118	0.016	0.064	0.035	0.026
PDE	60	0.091	0.012	0.067	0.022	0.020	0.099	0.019	0.033	0.033	0.031	0.103	0.013	0.067	0.033	0.027
SWT	43	0.067	0.015	0.023	0.027	0.026	0.033	0.006	0.023	0.008	0.005	0.018	0.007	0.0	0.0	0.0
Tomcat	97	0.135	0.048	0.052	0.070	0.074	0.132	0.051	0.062	0.072	0.071	0.141	0.055	0.062	0.077	0.088
ZXing	10	0.127	0.034	0.100	0.033	0.020	0.141	0.034	0.100	0.033	0.040	0.155	0.061	0.100	0.133	0.120
All projects	*331*	*0.091*	*0.030*	*0.042*	*0.039*	*0.042*	*0.107*	*0.040*	*0.054*	*0.057*	*0.056*	*0.096*	*0.036*	*0.045*	*0.049*	*0.049*
Changeset-files				D					ARC					ARC_L		
AspectJ	104	0.173	0.083	0.154	0.085	0.100	0.165	0.079	0.144	0.085	0.085	0.176	0.085	0.154	0.095	0.097
JDT	47	0.403	0.060	0.319	0.184	0.128	0.355	0.060	0.255	0.149	0.126	0.368	0.055	0.277	0.149	0.109
PDE	30	0.259	0.087	0.167	0.128	0.101	0.236	0.069	0.133	0.117	0.094	0.260	0.079	0.167	0.128	0.151
SWT	43	0.552	0.129	0.535	0.217	0.164	0.538	0.127	0.535	0.209	0.159	0.555	0.131	0.535	0.233	0.173
Tomcat	97	0.424	0.099	0.361	0.175	0.147	0.421	0.116	0.351	0.191	0.155	0.463	0.114	0.381	0.222	0.183
ZXing	10	0.199	0.157	0.100	0.133	0.140	0.212	0.163	0.100	0.133	0.220	0.200	0.159	0.100	0.133	0.120
All projects	*331*	*0.348*	*0.097*	*0.293*	*0.162*	*0.138*	*0.325*	*0.095*	*0.269*	*0.149*	*0.128*	*0.331*	*0.092*	*0.281*	*0.145*	*0.127*
Hunks				D					ARC					ARC_L		
AspectJ	104	0.175	0.084	0.163	0.091	0.093	0.176	0.082	0.163	0.093	0.083	0.183	0.093	0.173	0.111	0.099
JDT	47	0.362	0.059	0.255	0.135	0.122	0.322	0.049	0.213	0.149	0.109	0.429	0.062	0.319	0.195	0.167
PDE	30	0.249	0.088	0.167	0.122	0.141	0.288	0.093	0.200	0.144	0.127	0.200	0.068	0.133	0.078	0.087
SWT	43	0.510	0.117	0.465	0.225	0.196	0.519	0.142	0.442	0.240	0.201	0.526	0.131	0.488	0.217	0.164
Tomcat	97	0.426	0.135	0.289	0.211	0.191	0.441	0.140	0.351	0.211	0.211	0.482	0.129	0.412	0.216	0.182
ZXing	10	0.334	0.225	0.200	0.283	0.370	0.306	0.193	0.200	0.283	0.270	0.328	0.210	0.200	0.233	0.240
All projects	*331*	*0.330*	*0.101*	*0.254*	*0.159*	*0.152*	*0.334*	*0.105*	*0.272*	*0.162*	*0.144*	*0.355*	*0.107*	*0.296*	*0.171*	*0.149*

5.4.2 RQ2: Changeset encoding strategy

Table 11 shows retrieval performance of FBL-BERT trained and evaluated with different changeset encoding strategies and input data granularities. For each project, the three best performing configurations are highlighted, such that dark green marks a configuration with the highest retrieval performance, while green and yellow correspond to the second and third best configurations. Overall, we notice that using entire changesets as the granularity of input results in, by far, the worst performance across all of the investigated configurations for all evaluation projects. We can attribute this result to: (1) truncation of changesets due to input length limitation of the BERT model; and (2) tangled changes within a single changeset [51], which are likely to affect the model by introducing noise via unrelated code modifications. On the other hand, while the model based on hunks or changeset-files is not free of these problems, the finer data granularity allows it to partially overcome them. For instance, in case of tangled changes, dividing the entire changeset into hunks or changeset-files creates multiple new data points, which limits the noise introduced by instances that are poorly related to the bug. The difference in retrieval accuracy across all the metrics between using hunks and changeset-files as the input data is minor and differs from 1% to 12.2% per project. This results is indicative of the observation that leveraging hunks and changeset-files perform similarly and are both resilient to the problems affecting changesets.

Examining the results for different changest encoding strategies, we observe that ARC_L performs universally best across hunks and changeset-files. Interestingly, at the level of changeset-files, the baseline encoding D, which does not encode modification type, does surprisingly well and outperforms ARC encoding. We attribute this result to the specifics of ARC encoding, which groups lines based on the per-

formed modification, hence in the case of larger documents the grouping may affect the semantics of the documents. On the other hand, ARC encoding for hunks is less likely to be susceptible to that problem since hunks are typically much shorter. Analyzing the results for different projects, we observe that ARC_L performs best for AspectJ, JDT and Tomcat, with an improvement in MRR scores of 0.7%, 6.7% and 4.1% over their second best configurations respectively, while ARC is the most beneficial strategy for the PDE project. In the case of SWT, we observe the highest retrieval accuracy with ARC_L, while ZXing performs best with D encoding; however, both of these observations are likely negligible given the low difference between ARC_L and other encodings for SWT, and the relatively fewer bug reports in the ZXing project. Overall, we conclude that leveraging changesets semantics via encoding modification with either ARC and ARC_L increases retrieval accuracy over the D configuration which does not provide the model with additional information about the change. However, based on these results, the difference between ARC and ARC_L is not significant enough to clearly indicate which strategy is superior on average.

5.4.3 Threats to validity

The conclusions of this paper suffer from several threats to validity. A key threat to the internal validity of our study are the specific parameter choices we used to build our FBL-BERT model. A mitigating factor is that all parameters were either studied by us or were reported in other prior reputable papers as recommended or optimal [40, 112]. Another threat is our automated separation of bug reports based on localization hints into, not localized, partially localized, and fully localized, which may result in mistaken categorization, even though we used a well-known and frequently followed procedure [64].

Leveraging changesets for bug localization poses another threat due to possible

noise that can be introduced by SZZ [127], which could result in poor quality mapping between bug reports and bug-inducing changesets. However, the dataset was validated manually [24, 9], and therefore such mistakes, if they still exist, should not significantly affect our conclusions. Errors due to tangled changes [50, 51] are still possible in the dataset as such changes are difficult to remove manually. We believe tangled commits to have affected our final presented results (as discussed in RQ2), however, since tangled commits are a part of software development removing them completely may arguably result in unrealistic evaluation.

A threat to external validity, which concerns the ability to generalize our evaluation results, is that we applied the bug localization technique only on a limited number of bugs collected from a selection of popular open source Java projects. A mitigating factor is that the projects have a variety of purposes and development styles and the benchmark we used has also been applied to prior changeset-based bug localization studies [29, 91, 24]. Another threat to external validity is in the chosen evaluation metrics, which may not directly gauge user satisfaction with our bug localization technique [12], impacting the validity of the reported results. The threat is mitigated by the fact that the selected metrics are well-known and widely accepted as best available to measure and compare the performance of IR techniques.

5.5 Conclusion

In this chapter, we propose an approach for automatically retrieving bug-inducing changesets for a newly reported bug. The approach uses the popular BERT model to more accurately match the semantics in the bug report text to the inducing changeset. More specifically, we describe the FBL-BERT model, based on the prior work by Khattab et al. [112], which speeds up the retrieval of results while performing fine grained matching across all embeddings in the two documents.

The results show a significant improvement in retrieval accuracy for bug reports that lack localization hints or have only partial hints. We note that using a whole changeset as the primary data granularity negatively affect the accuracy of the model, while utilizing hunk- or file-level modifications brings significant improvement. By leveraging efficient retrieval architecture, the technique scales up with respect to the size of the search space (i.e. number of hunks), hence it is viable to be used in practice.

One of the key aspects of future work is to acquire more data from large scale software projects to extend the scope of the evaluation. Given that our technique require pairs of bug reports and bug-inducing changesets, the main challenge is in ensuring the quality of the data mined from the project history, since the quality of automated approaches designed for that purpose has been recently debated [120]. Another emerging research direction is in utilizing transfer learning properties inherent to BERT-based models to investigate the effectiveness of the proposed approach toward small to medium size software projects.

CHAPTER 6

DATA AUGMENTATION FOR IMPROVED DEEP LEARNING-BASED BUG LOCALIZATION

Deep Learning (DL) architectures has fueled outstanding improvements across multiple tasks in Natural Language Processing (NLP) and encouraged their application to various problems in the software engineering domain, such as bug localization. However, the fundamental weakness of DL approaches is that they require large amount of labelled data to train the model. At the same time, maintaining the quality of the labelled data is crucial to achieve the best performance. While manual labelling is typically a preferred approach to ensure high data quality, it is a slow and time-consuming process [128], often intractable considering the amount of data required to train a DL model. On the other hand, automated mining for labels is far more likely to meet the demand for data quantity, however at the cost of introducing noise in the form of both false positives and false negatives [129, 130]. Hence, collecting large amount of good quality labelled data can pose a significant challenge for many important software engineering problems and tasks, in particular those that require single project data (i.e., within project) [131]. A recent approach to address this problem is to use transfer learning, i.e., pre-training a model with unsupervised learning on a large, general corpus, followed by fine-tuning via supervised learning towards the downstream task. However, this strategy still requires a non-trivial dataset for fine-tuning and, as observed by Gururangan et al., it leads to suboptimal performance compared to when a model is pre-trained and fine tuned on in-domain data [83].

In the case of bug localization, the training data consists of pairs of bug reports and their introducing changesets, which are difficult to obtain at scale for a couple of key reasons. First, matching a bug report to bug-introducing changesets is challenging as developers rarely mark culprit code changes explicitly [11], while approaches that find the bug-inducing changesets automatically are prone to introducing noise [127, 120]. Second, the number of positive examples is bounded by the number of fixed bug reports, which are limited even for large and actively maintained projects. Relatively smaller software projects with, e.g., dozens of fixed bug reports, would be very difficult to use. In the end, the main question remains open: how to enable the potential of DL techniques for bug localization, given the paucity of project-specific data.

In the NLP domain, this question has been answered with some success by Data Augmentation (DA) techniques, which, in general, can be described as strategies to artificially increase the number and diversity of training examples based on the currently available data [132]. DA aims to create high quality synthetic data by applying transformations to the available data, while maintaining label invariance. As a result, the size of the original dataset increases, which in turn enables training a DL model for low resource domains and tasks.

In this chapter, we explore data augmentation for bug reports with the goal of producing a large number of high quality, realistic, synthetic bug reports, which can be subsequently used to increase the size of the training set for a bug localization DL model. To this end, we propose two sets of DA operators that independently target natural language text and code-related data (e.g., code tokens, stack traces and code snippets) in each bug report. More specifically, natural language text is augmented using token- and paragraph-level transformations (e.g., synonym inserts), while the code-related data is augmented with code tokens from its respective bug-inducing changesets in order to strengthen the connection between a bug report and different

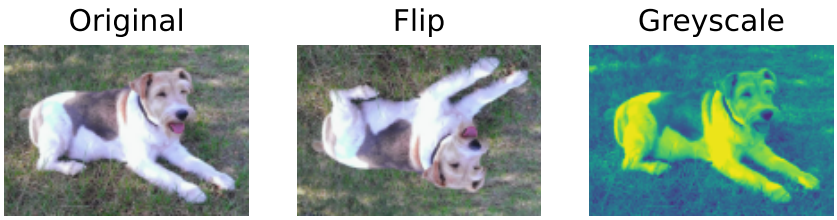


Fig. 13.: Data augmentation transformations in computer vision domain.

portions of its introducing changeset. At the same time, by leveraging the augmented bug reports we plan to achieve another important goal, i.e., balancing the augmented dataset toward parts of the source code underrepresented in the original training set.

6.1 Background

Data Augmentation (DA) aims to generate high quality, synthetic training examples from the already available data. To this end, DA transforms the existing training examples to create new data points that expand the original training set with the ultimate goal of improving the performance of the downstream DL model. DA was initially very successfully adopted in the computer vision domain, where simple image transformations, such as rotation or color manipulation, allow for the creation of an abundance of new data points to train the model on, thus leading to significant improvements in image classification tasks [133, 134].

Although the general concept of DA is easy to grasp, i.e., generating new data from the existing training set, the question of *how to* is very domain- or task-specific. Given a dataset, performing DA requires answering the following questions: (1) what are the data transformations that can create synthetic training examples, and (2) how the quality of those examples can be assured. Depending on the problem, the answers to those questions vary. For instance, in image classification there exist multiple well-established methods to produce augmented data, examples of which are depicted in

Figure 13. The key property of these transformations is that they are label invariant (i.e., a rotated or flipped image of dog, is still an image of a dog), which is fundamental to generating good quality labelled training examples.

Designing similar data transformations for textual data is significantly more challenging given the discrete nature of language. More specifically, compared to image transformations, augmenting textual data is more prone to inadvertently distorting the text's semantics, which in turn, can affect the original label. In other words, the transformation may not be label invariant. To prevent this from happening, researchers in NLP proposed numerous DA strategies ranging from simple rule-based techniques to transformations leveraging advanced DL models [135, 132].

Rule-based techniques rely on data transformations that typically extend and re-arrange the existing data. For instance, Wei and Zou. [136] proposed Easy Data Augmentation (EDA), which encompasses four token-level operators: random insert, random swap, random delete, and synonym replacement. The operators are applied only to a small portion of words (e.g., 1%) to minimize the risk of affecting the label. EDA improves the performance of text classification tasks, with the highest boost observed for the smallest datasets (i.e., 500 samples). Sahin et al. [137] introduced a sentence-level augmenter that uses a dependency tree to maintain the invariance of the sentence while swapping or deleting child nodes. Guided by in-domain knowledge, Yan et al. [138] proposed to randomly delete sentences in lengthy legal documents as many of them are irrelevant to the understanding of the case. The strength of rule-based techniques lies in the ease of use and interpretability of the augmentation operators, while the strict augmentation rules help to enforce label invariance.

This question of label preserving data transformations can be further extended to the overall question of quality and diversity of newly generated training examples. In other words, a synthetic example should be different enough from the original exam-

ple to add "new value" to the training set, and similar enough to preserve the original label. Given that rule-based DA techniques follow a rigid and fixed set of rules, they have limited ability to produce diverse output. To overcome that, another branch of DA methods use external models. For instance, Guo et al. [139] uses embeddings to convert text to a numeric vector, and subsequently mixes pairs of numeric vectors to create synthetic examples. Backtranslation [140, 141] is another popular method used to paraphrase the original data by translating a sentence to another language and back [142, 143, 144]. More recently, researchers explored generative data augmentation with the help of pre-trained GPT-2 models [145, 146]. While these methods are able to extrapolate further from the existing data, and, hence lead to more diverse augmentation, they require a good quality pre-trained models and are computationally expensive. Moreover, the recent work of Kovatchev et al. [147] points out that for domain-specific tasks, DA using models pre-trained on out-of-domain data, can be in fact outperformed by carefully designed rule-based operators that take into account a dictionary specific to the task. Interestingly, the authors reported to achieve the best performance when combining multiple, simple data transformations to generate a new example. While combining data transformations, known as stacking augmentations [135], is often used in the computer vision domain, it has only recently been effectively applied to textual data [147, 148].

Although the primary goal of DA is to increase the size of the training set, augmenting the data brings a few other benefits over the original datasets. First, using augmented data helps in preventing overfitting to the training data [149, 141]. Secondly, the augmented data can be used to fix the data distribution in case of data imbalance [150, 151, 152].

6.2 Data augmentation in bug localization

With the increasing complexity of DL-based methods for bug localization [30, 58, 153], the problem of data scarcity comes to the forefront. More specifically, while the more advanced models have the potential to bridge the lexical gap between a bug report and source code [154, 88], in order to fulfill that promise, they require large amount of bug reports to learn the semantics of the project and subsequently associate it with bug-inducing changesets. Insufficient amount of training examples may lead to model overfitting, memorizing high-frequency patterns or structures instead of generalizing the knowledge [135]. DA can help to address the data scarcity problem in bug localization by focusing on the following goals.

1. Increasing the number of bug reports. Training a DL model for bug localization requires a substantial dataset consisting of bug reports and bug-inducing changesets. The main challenge of constructing such dataset is that it is project-specific. Most software projects typically have few bug reports with a clear indication of the changesets that caused them [11]. Moreover, the total number of bug reports in a project is an upper bound on the number of positive training instances that are available. Note that while we can create numerous negative instances (i.e., a bug report and a non-bug-inducing changeset), the benefit to the DL model is limited as the bug report remains the same in each instance. Moreover, out of all the reported bugs, some are closed with *Won't fix* or *Not a Bug* status [64, 155], hence they do not have corresponding changesets and cannot be used for training. To empirically verify the scale of data scarcity problem in bug localization data, we examined Bench4BL [84], a large bug localization dataset. Bench4BL includes 10K bug reports and their fixes coming from 51 popular and actively developed open source software projects, which equals to roughly 200 bug reports per project. Considering that the

projects in the Bench4BL dataset are typically large and well-established (e.g., long running Apache Software Foundation projects like Camel and Hive), 200 bug reports is a discouragingly low number when it comes to ability to train an effective DL model.

2. Maintaining label invariance of bug reports. In NLP, data augmentation is primarily evaluated on classification tasks, such as sentiment analysis or topic classification, in which rarely a single word can be representative of the overall result (i.e., a sentiment or a topic). Data in software engineering is a mix of natural language and code-related segments. In case of bug localization this mix typically affects bug reports which often contain not only natural language description but also mentions of relevant program elements, stack traces or code snippets [154]. Applying off-the-shelf data augmentation transformations to bug localization data may cause more harm than good as it does not differentiate between NL and code, which both bring useful information, but in different forms and quantities. Table 12 shows examples of textual augmentation performed on the summary of bug report #55996 from the `Tomcat` project using two augmentation operators proposed by Wei et al. [136]. Random Swap exchanges two randomly selected words, while Synonym Replacement substitutes a randomly selected word with its synonym. To find synonyms, we use BERTOverflow [81], a BERT model pre-trained on the StackOverflow corpus. Given the randomness of data augmentation operations, we see different versions of augmented bug report summary. While *Random Swap 1* swaps two words without affecting the semantics, *Random Swap 2* exchanges words that can easily indicate the relevant code component, if a project contains *AsyncContext* and *AsyncConnector* classes. Similarly, in the case of *Synonym Replacement 1* changing *context* to *session* affects the semantics less than replacing *Async* with *TCP* which are different concepts. This toy-example shows how easily off-the-shelf data augmentation can

Table 12.: Examples of textual data augmentation with EDA [136].

	Bug report summary	Valid
Original	Async connector does not timeout with HTTP NIO context.	–
Random Swap 1	Async connector does **timeout** **not** with HTTP NIO context.	✓
Random Swap 2	Async **context** does not timeout with HTTP NIO **connector**.	✗
Synonym Replacement 1	Async connector does not timeout with HTTP NIO **session**.	✓
Synonym Replacement 2	**TCP** connector does not timeout with HTTP NIO context.	✗

introduce noise that affects the original label, especially when handling data that contains key software engineering-related phrases. Hence augmentation of software engineering data in general, and bug reports in particular, requires additional steps to ensure the invariance of the newly generated data points.

3. Diversifying the training data. The goal of data diversification in DA is to ensure that augmented data introduces "new quality" to a training set, such as previously unobserved motifs, patterns or expressions, leading a DL model to learn the meaning behind the data instead of memorizing certain forms [156, 135]. In the case of bug localization, the training dataset depicts how natural language describing a bug connects to source code concepts in the bug-inducing changeset. Commonly, the natural language in bug reports consists of Observed Behavior (OB), Expected Behavior (EB), or Steps to Reproduce (S2R) [66]. Given that OB, EB and S2R have been recognized by developers as useful information when fixing a bug [154], augmentation for bug localization data should focus on introducing diversity into those through, e.g., paraphrasing their sentences. The second important component of diversification of bug localization training set are the connections between bug reports and source code. While it is true that bugs are not evenly distributed in

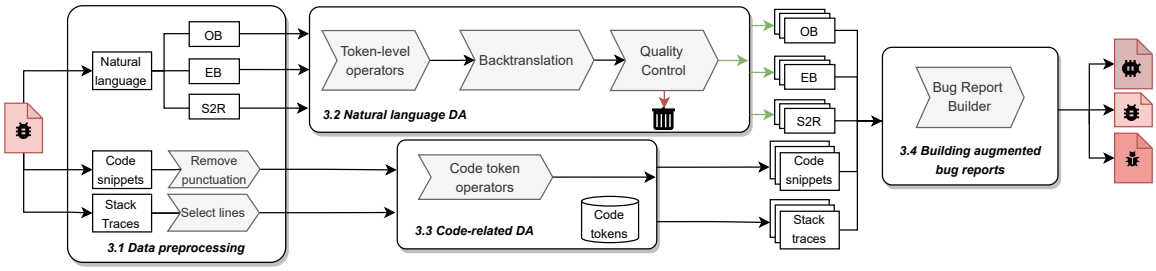


Fig. 14.: Augmentation pipeline for a single bug report.

the source code base, the over-representation of one source code component (e.g., class, package) in the training set, may lead to the model blaming that particular component for every bug. To account for that, while augmenting training set for bug localization, additional steps can be taken to mitigate that risk, through, e.g., creating more augmented bug reports for those source code components that occur less often in the training set. In summary, diversification of training data should focus on: (1) modifying the natural language content of a bug report, and (2) diversifying how a bug report connects to the source code.

6.3 Approach

To create augmented bug reports that introduce diversity and preserve invariance (i.e., the augmented bug report still matches the same changeset as the original bug report), we propose a set of custom DA operators. Bug reports describe software failure using various types of information, such as natural language, code snippets or stack traces, which may have different impact on matching a bug report to its inducing changeset, hence we decided to separately augment natural language and code-related information to ensure invariance of the newly created data points and avoid introducing noise. Figure 14 illustrates the workflow of our data augmentation process which starts with data extraction and preprocessing, followed by augmentation with

the proposed operators, and construction of augmented bug reports combining the newly generated data.

6.3.1 Data preprocessing

As a first step, we use *infozilla* [157], a tool that extracts stack traces and code snippets from unstructured bug report content, leaving the remaining text broadly categorized as natural language. To bring out further structure from the natural language data, we extract Observed Behavior (OB), Expected Behavior (EB), and Steps to Reproduce (S2R) using the BEE tool [158].

Stack traces are a valuable source of localization hints, however, due to their length they tend to introduce noise through multiple mentions of classes not necessarily related to a particular bug report [159, 13]. To mitigate the noise in stack traces, we reduce their size by selecting the lines that are most likely to contain relevant information. For instance, for Java stack traces this leads to three groups: 1) top lines, which include the exception name and where the exception originated; 2) middle lines, which occur after the Java standard library traces and are most likely last lines of the application code closest to the bug; and 3) bottom lines, which can be useful for exceptions thrown from threads. Sampling from these three groups creates a generic recipe that shortens the stack trace, captures different software designs, and preserves important information. Hence, for each stack trace, we decided to keep top 1 line, first 3 lines that refer to the application code, and bottom 1 line. Heuristic approaches such as this one have been reported to perform reasonably well even on unstructured runtime data (e.g., raw crash logs with multiple stack traces, possibly from different programming languages) [160].

For preprocessing code snippets, we decided to filter out punctuation for two reasons. First, in a recently published study, Paltenghi et al. [161] compared the

reasoning of developers and neural models, and observed that the models pay more attention to syntactic tokens (e.g., dots, periods, brackets), while developers focus more on strings or keywords. Given that developers perform better, DL models should mimic developers and put less attention to syntactic tokens. The second reason for filtering punctuation is pragmatic – reducing the number of tokens to prevent exceeding the input limit size of the DL models. Following preprocessing, each bug report is represented as a collection of OB, EB, S2R, stack traces, and code snippets.

6.3.2 Natural language DA operators

This group of operators is applied to OB, EB, and S2R due to their primarily natural language content. We propose to use two types of operators: token-level and paragraph-level. Inspired by a simple yet effective technique called *Easy Data Augmentation* (EDA) [53], we propose to use 4 token-level operators.

- **Dictionary Replace** - randomly selects a word from a pre-defined in-domain dictionary and replaces the word with its substitute.
- **Dictionary Insert** - works similarly to Dictionary Replace, however instead of replacing the word, this operator inserts the substitute at a random position in the text.
- **Random Swap** - randomly selects two words and swaps them.
- **Random Delete** - removes a randomly selected word.

To build the in-domain dictionary for augmenting OB, EB and S2R, we use keywords from language patterns devised by Chaparro et al. [162]. The patterns specify combinations of different parts of speech with certain keywords that have to occur to classify

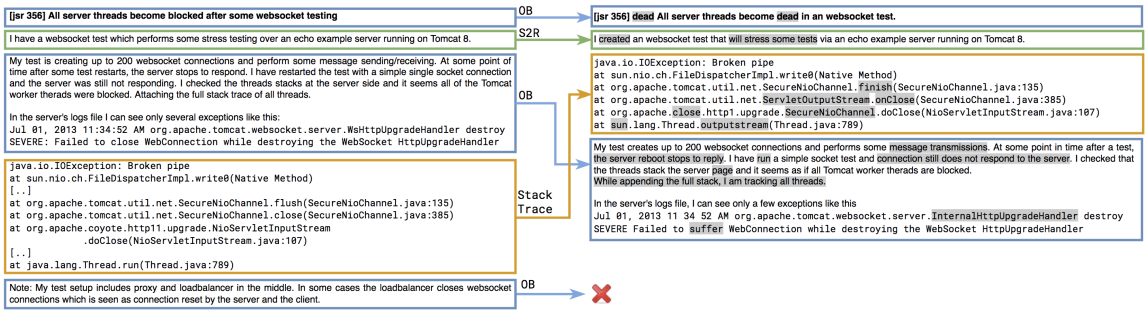


Fig. 15.: An example of augmented bug report for `Tomcat` #55171. Token-level modifications are marked with grey color.

a sentence or a paragraph as OB, EB or S2R. For instance, one of the most popular OB patterns is NEG_VERB defined as: *(subject/noun phrase) ([adjective/adverb]) [negative verb] ([complement])*, where the negative verbs are defined as: *affect, break, block, close, etc.* The in-domain dictionary contains all keywords identified by Chaparo et al. and maps each keyword to its substitutes, e.g., *affect* $\rightarrow$ {*break, block, close, ...*}. Domain knowledge guided operators have been recently shown to lead to better performance compared to more advanced but general approaches (e.g., embeddings) [147].

As a paragraph-level operator, we use **Backtranslation** to translate paragraphs of OB, EB or S2R from English to German and back to English [163]. Backtranslation is a popular data augmentation operation that allows to paraphrase the original text.

Finally, let us describe how those operators are applied together to generate augmented data. For each bug report and for each OB, EB, S2R, we apply all token level operators n times, where $n = \lambda * \#tokens$. The value of λ is set to 0.1 for insert, replace, and swap operations, and 0.05 for delete operation as these parameters have been empirically shown to produce best results [53]. Next, the Backtranslation operator is applied to paraphrase the modified text. Given the randomness of the

augmentation, the quality of the augmented sample may vary. As a final step, we employ quality control that consists of two steps. First, we check if OB, EB and/or S2R can be still identified in the augmented paragraph using the BEE tool. For instance, if the original paragraph contained OB and EB, then the augmented version must contain OB and EB be to considered a valid paragraph. We also disallow changing the pattern (e.g., from OB to EB). Second, we ensure that no code tokens are lost during the augmentation by comparing the number of code tokens between the augmented and the original paragraph.

6.3.3 Code-related DA operators

In the context of this paper, code-related data refers to stack traces, code snippets, and code tokens present in natural language text. To augment code-related data, we propose 3 code token operators that are more strict versions of the natural language operators to minimize the risk of distorting the context.

- **Code Token Replace** - randomly selects a code token and replaces it with its substitute.
- **Code Token Insert** - randomly selects a code token, and insert a substitute of that code token at a random position that is at most 3 positions away from the selected code token.
- **Code Token Swap** - swaps two randomly selected code tokens, such that (1) for stack traces code tokens can be swapped only between consecutive stack lines; (2) for code snippets, a swap operations must be performed within the surrounding 3 tokens.

We decided against including a code token deletion operator as removing code tokens is more likely to disturb the invariance of augmented samples.

To find substitutes for a code token, first, for each bug report we build a dictionary of code names using class and method names that occur in its corresponding bug-inducing changesets. Next, we use the Levenshtein distance to measure the distance between the selected code token and all other tokens in the dictionary. A substitute is selected randomly from the 20 code tokens that have the lowest distance from the selected code token.

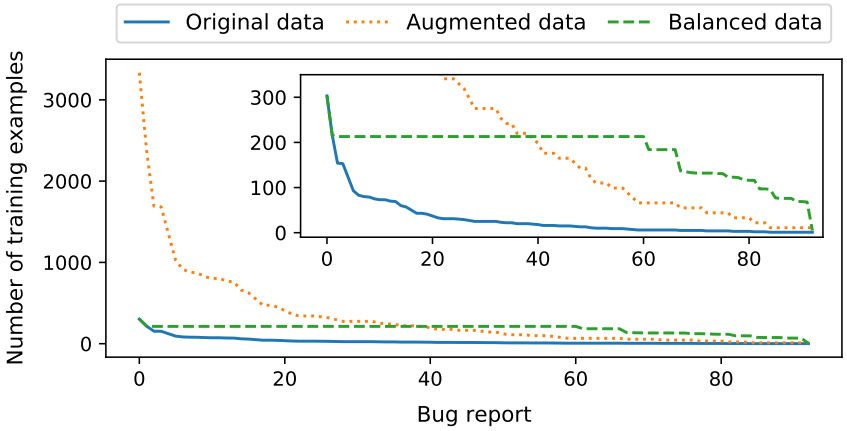
6.3.4 Building augmented bug reports

After augmentation, each bug report is decomposed into a collection of the original and augmented samples, i.e., natural language data (OB, EB, S2R), and code-related data (stack traces and code snippets). The remaining question is how to build a synthetic bug report out of all the available samples. Recent work in neural machine translation has shown that concatenating augmented samples introduces structural diversity that prevents a DL model from learning to focus only on one part of the input, thus leading to a significant improvement in the model's performance [164, 156]. We propose to use a similar approach to build augmented bug reports. More specifically, first we recreate the original structure of a bug report by concatenating augmented samples. Next, samples are reordered and at most 1 sample can be dropped to achieve further structural diversity. While dropping parts of bug reports may seem counterintuitive, DA strategies that remove tokens or sentences has been observed to have a positive impact on large pre-trained DL models [165, 166]. Figure 15 shows bug report #55171 from the `Tomcat` project and its augmented version. Each part of the bug report has been augmented separately using all of its respective DA operators. When constructing the augmented bug report, the second OB and the stack trace have been swapped, while the third OB has been dropped, creating the final augmented version of bug report #55171.

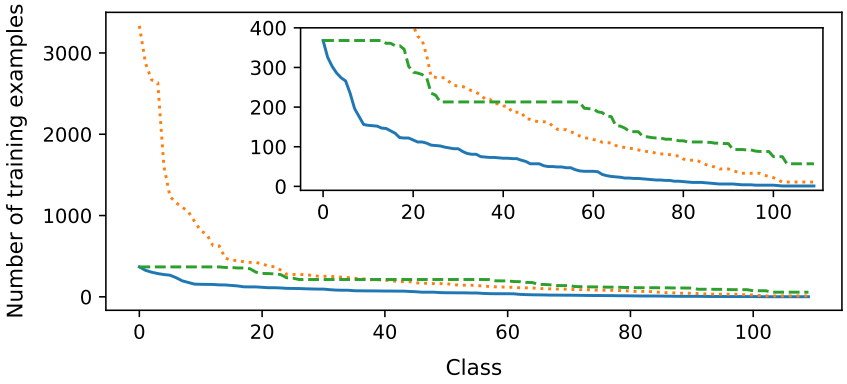
6.3.5 Ensuring a balanced augmented dataset

To increase the size of the bug localization training set with data augmentation, our approach is to focus on augmenting bug reports, increasing the number of pairs of bug reports and bug-inducing hunks. Recent studies show that using hunks, a set of consecutive line modifications that capture changes in one area of the file, produces improved retrieval results than using entire changesets [159, 24]. For instance, given a bug report with n introducing hunks, data augmentation by a factor of 10 creates 10 new bug reports for each hunk, which leads to $10n$ new training examples. However, there is one major drawback to this DA approach. Bugs affect different parts of source code base with varying frequency [167]. In other words, parts of the source code (i.e., specific files or classes) are related to multiple bug reports and therefore their hunks can also be overrepresented in the original dataset. This data imbalance, created by the uneven distribution of bug reports and hunks in the training set, can be exasperated by DA, with strong downstream effects on the DL model and its prediction.

To provide further evidence, we empirically checked the dataset published by Wen et al. [24], which we use in our study. Figure 16 shows the distribution of bug reports (Fig. 16a) and class occurrences (Fig. 16b) for the `Tomcat` project considering three different choices of training datasets: the original unaugmented dataset, a 10x augmented dataset, and an artificially balanced dataset. Within the plots, there are zoomed-in versions to increase readability at the smaller scale. In the plot for the original training set (i.e., the blue line), we observe that 11 out of 97 bug reports cover over 50% (1432 out of 2812) of the training examples, while 39 bug reports occur less than 10 times. Similarly, out of the unique 110 classes that introduced a bug, the top 10 classes with most frequently occurring hunks cover 34.5% (2586 out of



(a) Number of times a bug report occurs in the training set.



(b) Number of times a class occurs in the training set.

Fig. 16.: Data imbalance in bug localization training set.

7478) of all training data. This imbalance in the training data can have two potential consequences for supervised training of a DL model. First, the model is more likely to learn the structure and semantics of bug reports that have a large number of bug inducing hunks, while neglecting less frequent bug reports. Secondly, classes that occur the most in the training set are more likely to be selected as bug-inducing by the trained model since they were often seen during training as bug-inducing. The issue of data imbalance has also been recognized in defect prediction datasets [168].

How augmentation exacerbates the problem of uneven data distribution can be

observed in Figure 16, where the orange dotted line depicts the data distributions in a dataset that was augmented by a factor of 10. The majority bug reports and classes become even more dominant in the augmented dataset, making the data imbalance problem more severe than in the original dataset. To mitigate this problem, we propose a data balancing strategy that deliberately chooses samples to augment in order to smooth out the distributions of bug reports with respect to the source code. There are two main concerns that a data balancing strategy has to consider: (1) increasing the number of training examples for infrequent bug reports, and (2) ensuring that the number of examples with a given class does not dominate the dataset. To illustrate the need for these strategies, consider a bug report B_1 with 20 hunks from different classes, and a bug report B_2 with one hunk from class C. If the balancing strategy is focused only on the distribution of bug reports, then it creates 20 augmented examples for B_2, every time using the hunk from class C, hence C is likely to be overrepresented in the training set. To address this, we introduce two augmentation factors α and ω. While α influences the number of times each bug report is augmented, ω restricts how many times each class can be repeated in the augmented dataset.

Algorithm 1: Data balancing with augmented bug reports

Input : D_{train} – training dataset;
α – augmentation factor;
ω – balancing factor
Output: D_{bl} – balanced training dataset

```
1  max_br ← α× max. # of bug reports in D_train
2  max_cl ← ω× max. # of classes in D_train
3  D_bl ← D_train
4  for br, H_br in D_train do
5      while (count of br in D_bal) < max_br do
6          br_a ← augment br
7          h_a ← select hunk h_a from H_br, where
8                    (count of class for h_a in D_bal) < max_cl
9          Add br_a, h_a to D_bl
10     end while
11 end for
12 return D_bl
```

The sequence of steps for the proposed data balancing augmentation strategy is presented in Algorithm 1. In lines 1-2, we compute a limit for bug reports max_{br} and classes max_{cl} based on factors α and ω and the maximum number of times a unique bug report and class is present in the original training dataset. Line 3 copies the existing data instances into the balanced dataset D_{bl}. For each bug report that occurs below the max_{br} limit, the algorithm augments the bug report (line 6), and selects a bug-inducing hunk from a class that occurs less than max_{cl} times in D_{bl} (lines 7-8), creating a new training example. The algorithm continues to add new examples for a bug report until (1) the max_{br} limit is reached, or (2) bug-inducing hunks from all the classes have reached max_{cl}. The result of this balancing strategy is depicted in the green line in Figure 16, using values of $\alpha = 0.7$ and $\omega = 1.0$. Compared to the augmented dataset, the data distribution of the balanced dataset is obviously smoother, with a much more even representation of the source code.

6.4 Evaluation setup

6.4.1 Research Questions

RQ1: *Can DA improve the retrieval performance of DL-based bug localization?*

By generating synthetic bug reports with DA, we aim to increase the size of the training set with the goal of improving retrieval performance of the downstream DL-based model. To understand the impact of the proposed DA approach on DL-based bug localization, we identify three recent transformer-based models to perform this task, which includes FBL-BERT, a technique introduced in Chapter 5. We evaluate the performance of these bug localization approaches, with and without DA, using a standard bug localization dataset and metrics commonly used to measure information retrieval performance. As augmentation necessarily introduces significantly higher

data quantity, we add baselines to the evaluation that aim to differentiate the quantity vs. the quality of the augmented dataset.

RQ2: *Which of the proposed DA operators contribute the most to retrieval performance?*

The Data Augmentation approach in RQ1 relies on augmentation operators that perform specific types of transformations (e.g., insert, remove). In RQ2, we aim to understand what is the impact of these augmentation operators on the retrieval performance during bug localization. To answer RQ2, we perform ablation studies training each DL model with augmented datasets created using all but one augmentation operator type.

6.4.2 Dataset and models

To evaluate our data augmentation approach, we use a dataset published by Wen et al. [24] that contains data from 6 open source software projects: AspectJ, JDT, PDE, SWT, Tomcat, and ZXing. Given that *infozilla* requires new lines to extract code snippets and stack traces, and new lines were removed from all bug reports in Wen et al.'s dataset, we located and re-scraped the bug reports (with new lines) from Bugzilla for all projects. For ZXing, the bug reports in the GitHub issue tracker did not match those collected by Wen et al., likely because the project was moved, and therefore ZXing was excluded from the evaluation set. To create a training set for each project, we ordered the bug reports by opening dates and selected the first half for training, while the remaining bug reports constitute the test set. Each positive training example corresponds to a pair of a bug report and one of its inducing hunks (extracted from the inducing changeset). Each bug report includes the bug summary and description, while each hunk contains a log message and source code changes. The

Table 13.: Evaluation datasets for DA.

	Training		Testing	
Project	# bugs	D_{ori}	# bugs	# hunks
AspectJ	100	2212	100	23446
SWT	45	9982	45	69833
Tomcat	96	5624	97	72134
PDE	30	3856	30	100373
JDT	47	18230	47	150630

dataset of Wen et al. was constructed using SZZ [169], which identifies a changeset as bug-inducing if it shares *any* file modifications with bug fixing changeset. While a bug-inducing changeset may include modifications of multiple files, only a few of those may be relevant to a bug (as indicated by bug fixing changeset). Hence, to improve the quality of the positive training examples, we only include bug inducing hunks that refer to classes that also occurs in the bug fixing commit. For each positive example, we create a negative example by randomly selecting a hunk from a class which does not belong to the inducing changeset. After completing this step, for each project we obtain one training dataset, D_{ori}, which serves as our baseline dataset. The descriptive statistics of training and testing datasets used in this study are shown in Table 13. Note that the last column, *# hunks*, denotes the number of *all* hunks that are examined by the model during retrieval.

To evaluate the impact of the proposed data augmentation and balancing strategies on the retrieval performance, we train and evaluate three BERT-based [40] code retrieval architectures.

TBERT-Single [26, 109, 110] is the most straightforward approach for information retrieval with BERT. The model concatenates a bug report and a hunk, and processes it through BERT and a pooling layer to obtain a fused vector representation, which is subsequently passed to the classification head to obtain a relevancy score. While

this model typically provides high retrieval accuracy, it also incurs significant retrieval delay, since a bug report needs to be compared with *all* hunks available in a project.

TBERT-Siamese [26, 111] processes a bug report and a hunk sequentially through BERT and a pooling layer, creating two features vectors, that are subsequently concatenated and passed to the classification layer to produce the relevancy score. The key difference between TBERT-Single and TBERT-Siamese is in the opportunity to perform offline encoding of feature vectors for hunks, hence reducing the retrieval delay.

FBL-BERT [159, 112] is our proposed BERT-based architecture that enables rapid retrieval across a large collection of documents (i.e., hunks). Unlike TBERTs, which flattens the embedding matrix to a vector to make a prediction, FBL-BERT leverages the full embedding matrix and calculates relevancy score between a bug report and a hunk as a sum of maximum vector similarities between word embeddings of the bug report and hunk. This, in turn, allows to use efficient vector similarity search algorithms to find the most similar hunks and only re-rank those with FBL-BERT, hence significantly reducing the retrieval time per bug report. Given that FBL-BERT leverages fine-grained token-to-token embeddings matching, the model is more likely to better utilize relevant keywords if they occur in the bug report.

6.4.3 Experiment setup

We performed the experiments on a server with Dual 12-core 3.2GHz Intel Xeon and 1 NVIDIA Tesla V100 with 32GB RAM memory running CUDA v.11.4. The models are implemented with PyTorch v.1.7.1, HuggingFace library v.4.3.2, and Faiss v.1.6.5 with GPU support. We opted for using BERTOverflow [81] as our pre-trained base BERT model, since, similarly to our data, StackOverflow data is also a mixture of code and natural language. All models are fine tuned for 4 epochs, using a batch

size of 16 and Adam optimizer with learning rate set to 3e-6 [40]. Based on the average number of tokens in bug reports and hunks in our dataset, we set the input size limit to 256 and 512 tokens for bug reports and hunks respectively. All input documents are padded or truncated with respect to their input size limit.

6.5 Results

6.5.1 RQ1: Retrieval accuracy on augmented dataset

Setup. To evaluate the impact of DA on DL-based models, we compare the retrieval accuracy when training on the original, unaugmented dataset, D_{ori}, to training with augmented and balanced data. More specifically, for each project we construct the five augmented datasets shown in Table 14. D_{aug} is an augmented, but unbalanced, dataset that contains 10 additional examples for each pair of bug report and hunk, while D_{bl_i}, $i = 1, 2, 3, 4$, are balanced datasets with different choices for $\alpha = \{0.7, 0.85, 1.0, 1.3\}$ and $\omega = \{1.0, 2.0, 2.0, 2.0\}$ respectively. Given that augmentation increases the number of positive examples, the number of negative examples grows proportionally as well (i.e., for each positive example, there is one negative example). To ensure that the difference in performance is in fact the result of DA, and not the higher number of data instances, we created an additional baseline, D_{rep}, that *repeats* positive examples without augmentation 10 times, and, correspondingly, also adds 10 new negative examples. In effect, the only difference between D_{rep} and D_{aug} is the fact that D_{aug} uses augmented bug reports while D_{rep} repeats the positives examples.

All models are evaluated on the same test set. Since TBERT-Single requires significantly more time than the other models (e.g., TBERT-Single takes more than 24h to run on the JDT project), we only evaluate it on one of the balanced datasets -

Table 14.: Datasets for RQ1.

	AspectJ	**SWT**	**Tomcat**	**PDE**	**JDT**
Not augmented datasets					
D_{ori}	2.2k	9.9k	5.6k	3.9k	18.2k
D_{rep}	22.1k	99.8k	56.2k	38.6k	182.3k
Augmented datasets					
D_{aug}	24.3k	109.8k	61.9k	42.4k	200.5k
D_{bl1}	22.4k	66.9k	33.8k	25.1k	112.9k
D_{bl2}	29.8k	90.5k	46.8k	30.7k	142.3k
D_{bl3}	31.5k	95.1k	49.0k	32.5k	150.5k
D_{bl4}	44.2k	130.9k	65.6k	46.7k	216.4k

D_{bl_1} - as it exhibits the best performance for TBERT-Siamese, which uses a relatively similar DL architecture to TBERT-Single.

Results. Table 15 shows the retrieval performance of FBL-BERT, TBERT-Siamese and TBERT-Single trained on four dataset: D_{ori}, D_{rep}, D_{aug} and D_{bl*}, where D_{bl*} denotes the average best performing balanced dataset for the given model. In general, we observe that the models improve across all the metrics compared to D_{ori}, with the lowest improvement noted for D_{rep}, followed by D_{aug}, and with the highest improvement recorded for D_{bl*}.

Depending on the model the scale of the improvement varies. While the MRR score for FBL-BERT increases from 0.264 for D_{ori} to 0.367 for D_{bl*}, about half of the improvement can be attributed to the dataset size as indicated by the results for D_{rep} with the MRR score of 0.307. Moreover, we also observe that D_{aug} improves the score from 0.307 for D_{rep} to 0.353, indicating that using an augmented dataset makes a difference not only through data quantity. The improvement between D_{aug} and D_{bl*} is marginal and equal to 0.014, indicating that even the best balancing configuration has a small effect on FBL-BERT in general.

Training with a balanced dataset has a bigger impact on TBERT-Single and

Table 15.: Retrieval performance for different training datasets.

	MRR	**MAP**	**P@1**	**P@3**	**P@5**
Dataset	*FBL-BERT*				
D_{ori}	0.264	0.109	0.163	0.153	0.145
D_{rep}	0.307	0.129	0.213	0.179	0.176
D_{aug}	0.353	0.146	0.247	0.202	0.197
D_{bl2}	0.367	0.147	0.267	0.198	0.206
	TBERT-Siamese				
D_{ori}	0.180	0.062	0.144	0.076	0.069
D_{rep}	0.201	0.086	0.110	0.093	0.093
D_{aug}	0.236	0.103	0.157	0.124	0.119
D_{bl1}	0.328	0.107	0.247	0.150	0.146
	TBERT-Single				
D_{ori}	0.273	0.120	0.162	0.145	0.149
D_{rep}	0.271	0.140	0.152	0.136	0.176
D_{aug}	0.333	0.144	0.217	0.188	0.194
D_{bl1}	0.368	0.149	0.269	0.192	0.182

TBERT-Siamese with an improvement of 0.035 and 0.092 in MRR scores respectively when compared to D_{aug}. Moreover, data balancing is the key contributor to the improvement in TBERT-Siamese. Finally, comparing the results of D_{rep} and D_{aug} for both TBERT models and FBL-BERT, we observe that D_{aug} increases the retrieval accuracy across all metrics, indicating that the proposed bug reports augmentation approach is effective.

Table 16 shows the retrieval accuracy for FBL-BERT and TBERT-Siamese when the models are trained on different balanced datasets, with the values of α, ω and the dataset size provided on the right side of the table. In the case of FBL-BERT, D_{bl2} and D_{bl3} provide on average the best performance, improving MRR and MAP by 16.8% and 14.8% compared to D_{bl1} and D_{bl4}. However, as noted before, the improvement over the imbalanced dataset D_{aug} is marginal. In case of TBERT-Siamese, the smallest balanced dataset, D_{bl1}, produces the highest MRR score of 0.328 which outperforms other balanced dataset by at least 49%. To better understand how

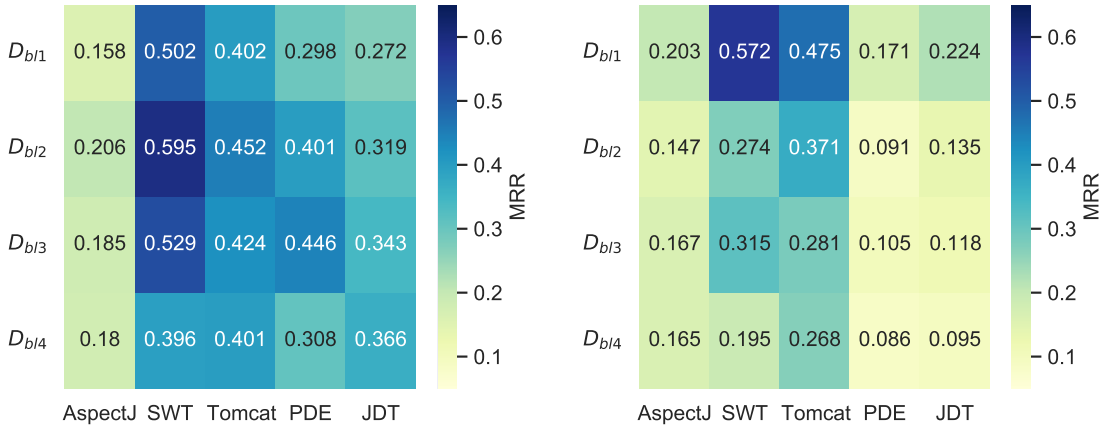
Table 16.: Retrieval performance with different balanced training datasets.

	MRR	**MAP**	**P@1**	**P@3**	**P@5**	α	ω	$\#D$
Dataset	*FBL-BERT*							
D_{bl1}	0.314	0.128	0.210	0.183	0.177	0.70	1.0	260k
D_{bl2}	0.367	0.147	0.267	0.198	0.206	0.85	2.0	340k
D_{bl3}	0.357	0.155	0.260	0.204	0.215	1.00	2.0	360k
D_{bl4}	0.315	0.142	0.217	0.176	0.179	1.30	2.0	500k
	TBERT-Siamese							
D_{bl1}	0.328	0.107	0.247	0.150	0.146	0.70	1.0	260k
D_{bl2}	0.220	0.080	0.140	0.116	0.111	0.85	2.0	340k
D_{bl3}	0.215	0.081	0.130	0.105	0.117	1.00	2.0	360k
D_{bl4}	0.182	0.068	0.107	0.091	0.086	1.30	2.0	500k

different balanced datasets affect the models' performance, in Figure 17 we show MRR scores across all evaluation projects ordered by the project sizes, i.e., the number of hunks in a project (see Table 8 for details). Interestingly, for FBL-BERT we observe that the larger the project, the more improvement when training with the bigger training dataset. More specifically, while for AspectJ, SWT, and Tomcat the maximum MRR scores are obtained with D_{bl2}, PDE performs best with D_{bl3}, and JDT with D_{bl4}. On the other hand, TBERT-Siamese consistently achieves the highest MRR scores for all projects with D_{bl1}, while other datasets, albeit larger, do not bring improvement. In summary, the results indicate that different model architectures may have different needs in terms of training dataset size to achieve their optimal performance. Some models benefit from more augmented samples, especially for larger projects.

6.5.2 RQ2: Impact of data augmentation operators

Setup. To better understand the influence of the proposed data augmentation operators on the downstream model effectiveness, we perform ablation studies on training datasets created using all but one augmentation operator type. To this end,



(a) FBL-BERT (b) TBERT-Siamese

Fig. 17.: MRR scores for evaluation projects trained on different balanced datasets.

we create 5 types of augmented training datasets: No Backtranslation, No Insert, No Delete, No Replace and No Swap operator. Note that we consider, e.g., both Code Token Swap and Random Swap as operators of Swap type. To balance the datasets, we use α and ω values from RQ1 that resulted in the best performance for the models, i.e., for FBL-BERT $\alpha = 0.85$, $\omega = 2.0$, while for TBERT models $\alpha = 0.7$, and $\omega = 1.0$.

Results. Figure 18 shows the MRR scores for datasets augmented with 4 out of 5 operator types as well as MRR scores of D_{ori} and D_{aug} as horizontal lines for reference. We note that most of the operators contribute towards the final performance, with an exception of Swap operator for FBL-BERT. The lack of impact for Swap operator can be attributed to the model architecture. Given that FBL-BERT leverages all of the tokens in a bug report separately, swapping the token positions does not preclude them from being matched. On the other hand, excluding Random Insert affects FBL-BERT the most, indicating that inserted tokens are valuable to the model and improve its effectiveness when matching token embeddings. The Delete opera-

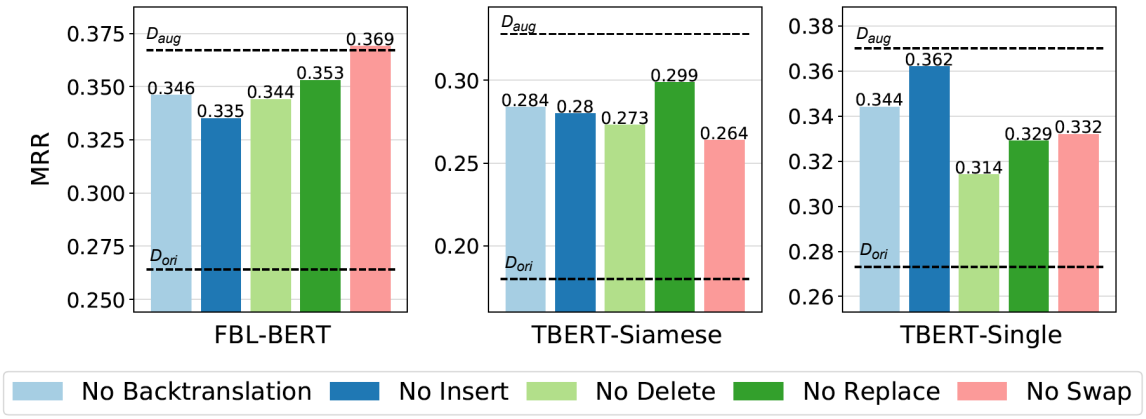


Fig. 18.: MRR scores when trained with augmented data using different DA operators.

tor is the most prominent contributor to the performance of both TBERT models. When the Delete operator is disallowed during augmentation, the MRR score of augmented datasets drops by 0.054 and 0.055 for TBERT-Single and TBERT-Siamese respectively, indicating that the variance caused by removing tokens randomly has a positive impact.

6.5.3 Threats to validity

There are several validity threads of our findings. A threat to internal validity of the study are the parameter choices for DL-based bug localization models, particularly in the context of (1) training procedure; (2) BERT-base selection; and (3) parameters inherent to each model. To mitigate that threat, during training we follow recommendations of BERT authors [40], while for each model we use parameters identified as optimal by the previous studies [26, 159]. While in our study, we use BERTOverflow as our BERT base model, other choices exist (e.g., CodeBERT [82]), and more models are underway, hence we leave the evaluation of different BERT base models in the context of bug localization to future work.

Another internal threat is in our choices of augmentation operators, their pa-

rameters (e.g., λ), and how they are applied together (e.g., stacking operators). This threat is mitigated by following best known practices from the NLP augmentation literature that focus on token-level operators and in-domain tasks [136, 147]. While we explore some parameter choices for data balancing in the paper (e.g., α and ω), there are also additional parameters related to bug report building process as well as other possible augmentation operators that may provide more improvement.

Furthermore, the randomness of augmentation operators pose a certain threat to the internal validity. To mitigate that, we ensure to set an initial value on the system's pseudo-random number generator when building an augmented dataset as well as when training a DL model.

A threat to the external validity is that we evaluated the data augmentation technique only for bug localization on a limited number of bugs collected from a selection of open source Java projects. This threat is mitigated by the fact that the dataset has been used in several prior bug localization studies [29, 91, 24]. Another mitigating factor is that the projects reflect a variety of purposes, development styles and histories.

Limitations in the chosen evaluation metrics pose a threat to conclusion validity as they may not directly measure user satisfaction with the retrieved change hunks [12]. The threat is mitigated by the fact that the selected metrics are well-known and widely accepted as best available to measure and compare the performance of IR techniques.

6.6 Conclusion

DL models toward bug localization excel at bridging the lexical gap between natural language describing a bug report and programming language that defines the source code. However, training an effective DL model requires large amount of

project-specific labelled data (i.e., pairs of bug reports and bug-inducing changesets), that is typically difficult to obtain in sufficient quantity for a single project. To relax the requirement on data quantity, and enable using DL model when training data is scarce, this chapter proposes to use data augmentation (DA) to create new, realistically looking bug reports that can be used to significantly increase the size of the training set. To augment bug reports, we propose DA operators that independently augment the natural language and code-related content of a bug report. To build a new training dataset using augmented bug reports, we propose a data balancing strategy that selectively augments bug reports to add more training examples for underrepresented parts of the source code.

The results indicate that the proposed data augmentation improves retrieval accuracy across all studied DL models increasing MRR score by 39% to 82% compared to the original, unaugmented dataset. Moreover, when augmented datasets are compared against training sets expanded by data repetition, we observe that they improve MRR scores by 20% to 36%. All of the proposed DA operators contribute to the final performance, with token deletion bringing most consistent impact for different DL models.

That being said, the proposed approach requires more experiments to strengthen our observations and recommendations. As our future work, we plan to (1) extend our evaluation datasets with new software projects written in Java, Python and Javascript; (2) conduct experiments with more heavily augmented data, i.e., by using DA operators on a larger number of tokens; (3) add Code Token Delete operator given good performance of Random Delete for natural language; and (4) experiment with different configurations for the bug report builder.

CHAPTER 7

CONCLUSION

This thesis proposes two context-aware bug localization models to address the lexical gap between natural language used to describe a bug and programming language that defines software functionalities. To learn how natural language maps to source code, the models leverage historical data about bugs and their related source code components. Besides retrieval accuracy, the thesis examines a set of practical considerations pertaining to applicability of the proposed bug localization models with respect to diverse characteristics of bug reports, retrieval delay, and availability of training data.

In order to bridge the lexical gap, this thesis proposes two bug localization models. JINGO uses two Online LDA models, one for bug reports and one for changesets, and translates between the two topic spaces using a translation matrix constructed from historical data. By leveraging LDA, JINGO creates a context-aware data representation at the *document*-level, while the translation matrix captures the mapping of natural language topics to source code topics. JINGO is evaluated on a task of locating relevant *source code files*, and shows the improvement over the baseline, especially for bug reports with no or a limited number of code tokens.

The second proposed approach is FBL-BERT, an efficient deep learning architecture that leverages BERT model to build a context-aware data representation at the *word*-level, hence it allows for more fine-grained context matching. FBL-BERT is evaluated on a task of locating relevant *changesets*, which is a significantly more challenging task compared to source code retrieval given that changesets are typically more numerous. The results show that FBL-BERT outperforms other deep learning

architectures due to its token-to-token embedding matching, and improves upon state of the art VSM-based baseline.

One of the main focus of this thesis are bug reports that have little to no code references and other localization hints. To address that, both models leverage historical data to learn the latent connection between bug reports and source code. The results indicate that JINGO and FBL-BERT improve upon their respective baselines when retrieving source code components for bug reports expressed primarily in the natural language.

Deploying a bug localization tool in a continuously evolving source code environment requires that such a tool is able to efficiently update its state based on observed changes, and, at the same time, can quickly locate relevant source code for a specific bug report. Given that a regular topic model requires full re-training when new data arrives, JINGO uses an Online LDA that supports adding new documents and updating topic distributions, hence the model is effectively kept up to date with respect to the current state of a software project. In the case of FBL-BERT, the main concern is the retrieval delay given that the FBL-BERT uses computationally heavy deep learning architecture, while retrieval typically requires comparing a bug report with every document in a search space. To this end, this thesis leverages late interaction BERT architecture that combines offline document representation with fast approximate search to rank only relevant candidates. Compared to state of the art BERT-base baselines, FBL-BERT significantly reduces retrieval delay and scales up with respect to the search space size.

While historical data is essential for the proposed models in order to learn the connection between natural language and source code, bug localization datasets are typically small. This, in turn, is likely to negatively affect the performance of the bug localization models, particularly FBL-BERT as it uses a data-demanding deep

learning model. This thesis addresses the data availability problem by introducing a data augmentation strategy to produce synthetic bug reports, and subsequently, use them to increase training dataset size. To ensure the quality of the augmented data, the proposed strategy leverages a set of augmentation operators that acts on different constituent components of bug reports, such as natural language, stack traces, or code snippets. At the same time, augmented bug reports are used selectively to address the imbalance of training examples pertaining to different parts of the source code. The results indicate that the proposed strategies improve retrieval accuracy across all studied deep learning models.

CHAPTER 8

FUTURE WORK

The contributions of this work highlight the advantages of leveraging contextual models for bug localization, while identifying numerous challenges pertaining to data availability and diversity as well as models' scalability. Based on the presented results, we envision the following research directions.

8.1 More extensive evaluation

There are a few possibilities to further extend the evaluation scope of the proposed bug localization approaches. First, it would be interesting to see the performance of the proposed techniques on more software projects, particularly those written in Python and Javascript given the popularity of those programming languages [170], and their relatively infrequent use in the evaluation datasets for bug localization. Secondly, this work uses BERTOverflow as a pre-trained BERT base, however more choices of pre-trained BERTs are available (e.g., CodeBERT [82]). Moreover, since the time BERT was originally proposed, researchers have devised multiple BERT-like models that aim to address problems of the initial architecture, such as input limit size (e.g., LongTransformer [171], Reformer [172]), and incorporating structural information (e.g., StructBERT [173]). Given that those problems have the potential to impact software engineering data in general, and bug localization in particular, further investigation of those models could be beneficial for the research community.

8.2 Capturing project-specific information

While using an embedding-based model for bug localization brings certain advantages, such as capturing semantics, and recognizing idiosyncratic coding conventions, at the same time it results in the lossy data representation, which eliminates the possibility for exact term matching [108]. This issue is particularly apparent when comparing retrieval accuracy between our BERT-based approach and a VSM-based model for bug reports that contain code references, in which case the later model is superior. Although such bug reports are likely to pose less of a challenge for software developers, a bug localization tool that fails to localize "easy" cases is unlikely to gain developers' trust, hence it may not be used in practice. Researchers in the medical NLP field recognized the same problem in the context of specific and unique medical terms, and proposed an architecture combining a general purpose BERT with BERT trained only on medical terms [174]. To address the lossy representation problem, one possibility it to extract and preserve project-specific terms explicitly, with a separate, lightweight module that extends the FBL-BERT architecture.

8.3 Extrapolating further from the available data

Data augmentation is a promising research direction in the context of small datasets in the software engineering domain. The key limiting factor of the proposed DA approach pertains to the novelty of synthetic bug reports. More specifically, while augmented bug reports strengthen the connection between natural language and the relevant source code, they are operating within a boundary of certain software features. In other words, even though new source code is constantly developed, as long as no bug report for the new code exists, the code is not reflected in the training set. Hence, the question is: *can we generate truly artificial bug reports for the newly*

developed code?. A potential answer may lie in the realm of Generative Data Augmentation which aims to generate new examples leveraging the knowledge and patterns learned by large pre-trained models [175, 176].

8.4 Interpretability of the results

One of the disadvantages of using deep learning models is that they are considered "black box" models, i.e., while the values on the input and output are known, the internal working of the model and its reasoning is obscure. Given the remarkable accuracy of deep learning models on multiple tasks, the lack of interpretability remains one of the key factors that limits deployment of such models in practice, since users are unlikely to trust them [177]. Therefore, one of the possible future research directions is in devising an interpretability method able to clarify the results of a bug localization tool and shed some light on why a particular changeset is deemed to cause the bug. While constructing such a method is certainly not straightforward for many deep learning models, the architecture of FBL-BERT is particularly interpretability-friendly as it uses token to token matching.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010.

[3] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. First. Addison-Wesley Professional, 2007.

[4] Rachel Potvin and Josh Levenberg. "Why Google stores billions of lines of code in a single repository". In: *Communications of the ACM* 59.7 (2016), pp. 78–87.

[5] Ranjita Bhagwan et al. "Orca: Differential Bug Localization in Large-scale Services". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA, 2018, pp. 493–509.

[6] Harry McCracken. *The Year That Software Bugs Ate The World*. Dec. 2017. URL: `https://www.fastcompany.com/40505226/the-year-that-software-bugs-ate-the-world`.

[7] Jinshui Wang et al. "How developers perform feature location tasks: a human-centric and process-oriented exploratory study". In: *Journal of Software: Evolution and Process* 25.11 (2013).

[8] A. T. Nguyen et al. “A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report”. In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 2011, pp. 263–272. DOI: 10.1109/ASE.2011.6100062.

[9] J. Zhou, H. Zhang, and D. Lo. “Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports”. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 14–24. DOI: 10.1109/ICSE.2012.6227210.

[10] Chris Mills et al. “On the relationship between bug reports and queries for text retrieval-based bug localization”. In: *Empirical Software Engineering* 25 (2020).

[11] Vijayaraghavan Murali et al. “Industry-scale IR-based Bug Localization: A Perspective from Facebook”. In: *Proceedings of the 42nd International Conference on Software Engineering*. ICSE ’20. 2020.

[12] Qianqian Wang, Chris Parnin, and Alessandro Orso. “Evaluating the Usefulness of IR-Based Fault Localization Techniques”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. Baltimore, MD, USA, 2015, pp. 1–11.

[13] Mohammad Masudur Rahman and Chanchal K Roy. “Improving IR-based bug localization with context-aware query reformulation”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 621–632.

[14] Pieter Hooimeijer and Westley Weimer. “Modeling Bug Report Quality”. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. ACM, 2007, pp. 34–43.

[15] A. N. Lam et al. “Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 476–481.

[16] F. Servant and J. A. Jones. “WhoseFault: Automatic developer-to-fault assignment through fault localization”. In: *2012 34th International Conference on Software Engineering (ICSE)*. June 2012, pp. 36–46.

[17] A. Marcus et al. “An information retrieval approach to concept location in source code”. In: *11th Working Conference on Reverse Engineering*. Nov. 2004, pp. 214–223.

[18] G. Gay et al. “On the use of relevance feedback in IR-based concept location”. In: *2009 IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 351–360.

[19] M. B. Zanjani, H. Kagdi, and C. Bird. “Using Developer-Interaction Trails to Triage Change Requests”. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. May 2015, pp. 88–98.

[20] M. B. Zanjani, H. Kagdi, and C. Bird. “Automatically Recommending Peer Reviewers in Modern Code Review”. In: *IEEE Transactions on Software Engineering* 42.6 (June 2016), pp. 530–543.

[21] R. K. Saha et al. “An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes”. In: *2015 IEEE/ACM 37th IEEE*

International Conference on Software Engineering. Vol. 1. June 2015, pp. 268–279.

[22] C. D. Nguyen, A. Marchetto, and P. Tonella. "Test Case Prioritization for Audit Testing of Evolving Web Services Using Information Retrieval Techniques". In: *2011 IEEE International Conference on Web Services*. July 2011, pp. 636–643.

[23] J. Chen et al. "Test Case Prioritization for Compilers: A Text-Vector Based Approach". In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2016, pp. 266–277.

[24] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. "Locus: Locating Bugs from Software Changes". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore, 2016, pp. 262–273.

[25] C. S. Corley, K. Damevski, and N. A. Kraft. "Changeset-Based Topic Modeling of Software Repositories". In: *IEEE Transactions on Software Engineering* (2018).

[26] Jinfeng Lin et al. *Traceability Transformed: Generating more Accurate Links with Pre-Trained BERT Models*. 2021. arXiv: `2102.04411 [cs.SE]`.

[27] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. "Semantically Enhanced Software Traceability Using Deep Learning Techniques". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. 2017. ISBN: 9781538638682.

[28] Jeniya Tabassum et al. "Code and Named Entity Recognition in StackOverflow". In: *Proceedings of the 58th Annual Meeting of the Association for*

Computational Linguistics. Online: Association for Computational Linguistics, July 2020, pp. 4913–4926. DOI: 10.18653/v1/2020.acl-main.443. URL: https://www.aclweb.org/anthology/2020.acl-main.443.

[29] Ripon K. Saha et al. "Improving Bug Localization Using Structured Information Retrieval". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. Silicon Valley, CA, USA, 2013, pp. 345–355.

[30] X. Huo et al. "Deep Transfer Bug Localization". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. DOI: 10.1109/TSE.2019.2920771.

[31] A. T. Nguyen et al. "A topic-based approach for narrowing the search space of buggy files from a bug report". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Nov. 2011, pp. 263–272.

[32] J. Zhou, H. Zhang, and D. Lo. "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports". In: *2012 34th International Conference on Software Engineering (ICSE)*. June 2012, pp. 14–24.

[33] Yan Xiao et al. "Bug Localization with Semantic and Structural Features Using Convolutional Neural Network and Cascade Forest". In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. 2018, pp. 101–111.

[34] Brian Eddy, Nicholas Kraft, and Jeff Gray. "Impact of structural weighting on a latent Dirichlet allocation-based feature location technique: Impact of structural weighting on a latent Dirichl et al location-based feature location technique". In: *Journal of Software: Evolution and Process* 30 (Sept. 2017).

[35] Andrian Marcus and Timothy Menzies. “Software is Data Too”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER ’10. 2010, pp. 229–232.

[36] G. Salton, A. Wong, and C. S. Yang. “A Vector Space Model for Automatic Indexing”. In: *Commun. ACM* 18.11 (Nov. 1975).

[37] David M Blei, Andrew Y Ng, and Michael I Jordan. “Latent dirichlet allocation”. In: *Journal of Machine Learning Research* 3.Jan (2003), pp. 993–1022.

[38] Tomas Mikolov et al. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems*. Ed. by C. J. C. Burges et al. 2013.

[39] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014.

[40] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2019.

[41] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. 2017.

[42] Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. “On Combining IR Methods to Improve Bug Localization”. In: ICPC ’20. Seoul, Republic of Korea, 2020.

[43] A. Alali, H. Kagdi, and J. I. Maletic. "What's a Typical Commit? A Characterization of Open Source Software Repositories". In: *Proceedings of the 16th IEEE International Conference on Program Comprehension*. June 2008, pp. 182–191.

[44] W. Maalej and H. Happel. "Can development work describe itself?" In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. May 2010.

[45] J. Shen et al. "On Automatic Summarization of What and Why Information in Source Code Changes". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. June 2016, pp. 103–112.

[46] Eugene W. Myers. "AnO(ND) difference algorithm and its variations". In: *Algorithmica* 1.1 (Nov. 1986).

[47] Ahmed E Hassan and Richard C Holt. "Predicting change propagation in software systems". In: *Proceedings of the 2004 20th IEEE International Conference on Software Maintenance*. IEEE. 2004, pp. 284–293.

[48] Atif Memon et al. "Taming Google-scale continuous testing". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press. 2017, pp. 233–242.

[49] T. L. Graves et al. "Predicting fault incidence using software change history". In: *IEEE Transactions on Software Engineering* 26.7 (July 2000), pp. 653–661.

[50] Shane McIntosh et al. "An Empirical Study of Build Maintenance Effort". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA, 2011, pp. 141–150.

[51] Kim Herzig and Andreas Zeller. “The Impact of Tangled Code Changes”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR ’13. San Francisco, CA, USA, 2013, pp. 121–130.

[52] Andrew Gelman et al. *Bayesian Data Analysis*. Vol. 2. Boca Raton, FL: CRC Press, 2014.

[53] W. Zou et al. “How Practitioners Perceive Automated Bug Report Management Techniques”. In: *IEEE Transactions on Software Engineering* 46.8 (2020).

[54] D. Kim et al. “Where Should We Fix This Bug? A Two-Phase Recommendation Model”. In: *IEEE Transactions on Software Engineering* 39.11 (Nov. 2013), pp. 1597–1610.

[55] Tezcan Dilshener, Michel Wermelinger, and Yijun Yu. “Locating Bugs without Looking Back”. In: *Automated Software Engineering* 25.3 (2018), pp. 383–434.

[56] Shaowei Wang and David Lo. “Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization”. In: *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India, 2014, pp. 53–63.

[57] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. “Bug Localization Using Latent Dirichlet Allocation”. In: *Inf. Softw. Technol.* 52.9 (Sept. 2010), pp. 972–990.

[58] An Ngoc Lam et al. “Bug Localization with Combination of Deep Learning and Information Retrieval”. In: *Proceedings of the 25th International Conference on Program Comprehension*. ICPC ’17. Buenos Aires, Argentina: IEEE

Press, 2017, pp. 218–229. ISBN: 9781538605356. URL: https://doi.org/10.1109/ICPC.2017.24.

[59] Xuan Huo, Ming Li, and Zhi-Hua Zhou. "Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. IJCAI'16. 2016.

[60] Xuan Huo and Ming Li. "Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI'17. 2017.

[61] Rongxin Wu et al. "ChangeLocator: Locate Crash-Inducing Changes Based on Crash Reports". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. 2018.

[62] Thong Hoang et al. "CC2Vec: Distributed representations of code changes". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020.

[63] Rocìo Cabrera Lozoya et al. *Commit2Vec: Learning Distributed Representations of Code Changes*. 2019. arXiv: 1911.07605.

[64] Pavneet Singh Kochhar, Yuan Tian, and David Lo. "Potential Biases in Bug Localization: Do They Matter?" In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden, 2014, pp. 803–814.

[65] Chris Mills et al. "Are Bug Reports Enough for Text Retrieval-based Bug Localization?" In: *Proceedings of the 34th IEEE International Conference*

on Software Maintenance and Evolution (ICSME'18). Madrid, Spain: ACM, Sept. 2018.

[66] O. Chaparro, J. M. Florez, and A. Marcus. "Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2017, pp. 376–387.

[67] Misoo Kim and Eunseok Lee. "A Novel Approach to Automatic Query Reformulation for IR-Based Bug Localization". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus, 2019.

[68] Tien-Duy B. Le, Ferdian Thung, and David Lo. "Will This Localization Tool Be Effective for This Bug? Mitigating the Impact of Unreliability of Information Retrieval Based Bug Localization Tools". In: *Empirical Softw. Engg.* 22.4 (Aug. 2017), pp. 2237–2279. ISSN: 1382-3256.

[69] D. Kim et al. "Where Should We Fix This Bug? A Two-Phase Recommendation Model". In: *IEEE Transactions on Software Engineering* 39.11 (Nov. 2013).

[70] Julian Aron Prenner and Romain Robbes. *Making the most of small Software Engineering datasets with modern machine learning*. 2021. arXiv: `2106.15209 [cs.SE]`.

[71] Hamel Husain et al. "CodeSearchNet challenge: Evaluating the state of semantic code search". In: *arXiv preprint arXiv:1909.09436* (2019).

[72] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep Code Search". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 933–944.

[73] Xinli Yang et al. "Deep Learning for Just-in-Time Defect Prediction". In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 17–26. DOI: 10.1109/QRS.2015.14.

[74] Jian Li et al. "Software Defect Prediction via Convolutional Neural Network". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, pp. 318–328.

[75] Thong Hoang et al. "DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 34–45.

[76] Song Wang et al. "Deep Semantic Feature Learning for Software Defect Prediction". In: *IEEE Transactions on Software Engineering* 46.12 (2020).

[77] Chanathip Pornprasit and Chakkrit Tantithamthavorn. "DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction". In: *IEEE Transactions on Software Engineering* (2022).

[78] Yan Xiao et al. "Improving bug localization with word embedding and enhanced convolutional neural networks". In: *Information and Software Technology* 105 (2019).

[79] X. Huo et al. "Deep Transfer Bug Localization". In: *IEEE Transactions on Software Engineering* (2019).

[80] Ziye Zhu et al. "CooBa: Cross-project Bug Localization via Adversarial Transfer Learning". In: *IJCAI*. 2020.

[81] Jeniya Tabassum et al. "Code and Named Entity Recognition in StackOverflow". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020.

[82] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Nov. 2020.

[83] Suchin Gururangan et al. “Don’t stop pretraining: adapt language models to domains and tasks”. In: *arXiv preprint arXiv:2004.10964* (2020).

[84] Jaekwon Lee et al. “Bench4BL: Reproducibility Study on the Performance of IR-based Bug Localization”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands, 2018, pp. 61–72.

[85] Matthew Hoffman, Francis R Bach, and David M Blei. “Online learning for Latent Dirichlet Allocation”. In: *advances in neural information processing systems*. 2010, pp. 856–864.

[86] S. Amasaki, H. Aman, and T. Yokogawa. “On the Effects of File-level Information on Method-level Bug Localization”. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 314–321.

[87] Yu Wang, Eugene Agichtein, and Michele Benzi. “TM-LDA: Efficient Online Modeling of Latent Topic Transitions in Social Media”. In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’12. Beijing, China, 2012, pp. 123–131.

[88] Xin Ye, Razvan Bunescu, and Chang Liu. “Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China, 2014, pp. 689–699.

[89] Yaojing Wang et al. “Enhancing supervised bug localization with metadata and stack-trace”. In: *Knowledge and Information Systems* (2020), pp. 1–24.

[90] Bader Alkhazi et al. “Learning to rank developers for bug report assignment”. In: *Applied Soft Computing* 95 (2020).

[91] Chu-Pan Wong et al. “Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis”. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME ’14. 2014, pp. 181–190.

[92] David Binkley et al. “Understanding LDA in source code analysis”. In: *Proceedings of the 22nd international conference on program comprehension*. ACM. 2014, pp. 26–36.

[93] Sergei Koltcov, Olessia Koltsova, and Sergey Nikolenko. “Latent dirichlet allocation: stability and applications to studies of user-generated content”. In: *Proceedings of the 2014 ACM conference on Web science*. ACM. 2014, pp. 161–165.

[94] Amritanshu Agrawal, Wei Fu, and Tim Menzies. “What is wrong with topic modeling? And how to fix it using search-based software engineering”. In: *Information and Software Technology* 98 (2018), pp. 74–88.

[95] Christoph Treude and Markus Wagner. “Predicting Good Configurations for GitHub and Stack Overflow Topic Models”. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR ’19. Montreal, Quebec, Canada, 2019, pp. 84–95.

[96] J. Romano et al. “Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen’sd for evaluating group differences on the

NSSE and other surveys?" In: *annual meeting of the Florida Association of Institutional Research*. 2006, pp. 1–3.

[97] Klaus Changsun Youm et al. "Bug Localization Based on Code Change Histories and Bug Reports". In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 2015, pp. 190–197.

[98] Shivani Rao and Avinash Kak. "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. Waikiki, Honolulu, HI, USA, 2011, pp. 43–52.

[99] Chakkrit Tantithamthavorn et al. "The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization". In: *Information and Software Technology* (2018).

[100] Wen Zhang et al. "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion". In: *Information and Software Technology* 110 (2019), pp. 121–135.

[101] S. Cheng, X. Yan, and A. A. Khan. "A Similarity Integration Method based Information Retrieval and Word Embedding in Bug Localization". In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 2020.

[102] J. Cao et al. "BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs". In: *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020.

[103] Zimin Chen and Martin Monperrus. "A literature study of embeddings on source code". In: *arXiv preprint arXiv:1904.03061* (2019).

[104] Michael Pradel et al. “Scaffle: Bug Localization on Millions of Files”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. 2020.

[105] T. Savor et al. “Continuous Deployment at Facebook and OANDA”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, pp. 21–30.

[106] Christoffer Rosen, Ben Grawi, and Emad Shihab. “Commit Guru: Analytics and Risk Prediction of Software Commits”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 966–969. ISBN: 9781450336758.

[107] Z. Wan et al. “Perceptions, Expectations, and Challenges in Defect Prediction”. In: *IEEE Transactions on Software Engineering* 46.11 (2020).

[108] Saksham Sachdev et al. “Retrieval on Source Code: A Neural Code Search”. In: *Proceedings of the 2nd Workshop on Machine Learning and Programming Language*. MAPL 2018. 2018.

[109] Zhuyun Dai and Jamie Callan. “Deeper Text Understanding for IR with Contextual Neural Language Modeling”. In: *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR’19. 2019.

[110] Rodrigo Nogueira and Kyunghyun Cho. *Passage Re-ranking with BERT*. 2020. arXiv: 1901.04085 [cs.IR].

[111] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. arXiv: 1908.10084 [cs.CL].

[112] Omar Khattab and Matei Zaharia. “ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT”. In: SIGIR ’20. 2020.

[113] Ian Tenney, Dipanjan Das, and Ellie Pavlick. “BERT Rediscovers the Classical NLP Pipeline”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. July 2019.

[114] Matthew Peters et al. “Dissecting Contextual Word Embeddings: Architecture and Representation”. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018.

[115] Jeff Johnson, Matthijs Douze, and Hervé Jégou. “Billion-scale similarity search with GPUs”. In: *arXiv preprint arXiv:1702.08734* (2017).

[116] Jean-Rémy Falleri et al. “Fine-grained and accurate source code differencing”. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*. 2014, pp. 313–324. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982.

[117] Zhengran Zeng et al. “Deep Just-in-Time Defect Prediction: How Far Are We?” In: ISSTA 2021. Virtual, Denmark, 2021.

[118] M. Schuster and K. Nakajima. “Japanese and Korean voice search”. In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012.

[119] Kim Herzig and Andreas Zeller. “The impact of tangled code changes”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013.

[120] E. C. Neto, D. A. da Costa, and U. Kulesza. "The impact of refactoring changes on the SZZ algorithm: An empirical study". In: *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. SANER 2018. 2018.

[121] Siyuan Jiang, Ameer Armaly, and Collin McMillan. "Automatically generating commit messages from diffs using neural machine translation". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017.

[122] Qianqian Wang, Chris Parnin, and Alessandro Orso. "Evaluating the Usefulness of IR-Based Fault Localization Techniques". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA, 2015, pp. 1–11.

[123] Rafael-Michael Karampatsis et al. "Open-Vocabulary Models for Source Code". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ICSE '20. Seoul, South Korea, 2020, pp. 294–295.

[124] Jinhyuk Lee et al. "BioBERT: a pre-trained biomedical language representation model for biomedical text mining". In: *Bioinformatics* (Sept. 2019). Ed. by JonathanEditor Wren. ISSN: 1460-2059. DOI: 10.1093/bioinformatics/btz682. URL: http://dx.doi.org/10.1093/bioinformatics/btz682.

[125] Iz Beltagy, Kyle Lo, and Arman Cohan. "SciBERT: Pretrained Language Model for Scientific Text". In: *EMNLP*. 2019. eprint: arXiv:1903.10676.

[126] Yu Gu et al. *Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing*. 2021. arXiv: 2007.15779 [cs.CL].

[127] Giovanni Rosa et al. *Evaluating SZZ Implementations Through a Developer-informed Oracle*. 2021. arXiv: 2102.03300 [cs.SE].

[128] Huy Tu, Zhe Yu, and Tim Menzies. "Better Data Labelling With EMBLEM (and how that Impacts Defect Prediction)". In: *IEEE Transactions on Software Engineering* 48.1 (2022), pp. 278–294.

[129] Bogdan Vasilescu et al. "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. 2015.

[130] Daniel Alencar da Costa et al. "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes". In: *IEEE Transactions on Software Engineering* (2017).

[131] Huy Tu and Tim Menzies. "FRUGAL: Unlocking SSL for Software Analytics". In: *arXiv preprint arXiv:2108.09847* (2021).

[132] Steven Y Feng et al. "A survey of data augmentation approaches for nlp". In: *arXiv preprint arXiv:2105.03075* (2021).

[133] Christian Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015. DOI: 10.1109/CVPR.2015.7298594.

[134] Sebastien C. Wong et al. "Understanding Data Augmentation for Classification: When to Warp?" In: *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. 2016, pp. 1–6. DOI: 10.1109/DICTA.2016.7797091.

[135] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. "Text data augmentation for deep learning". In: *Journal of big Data* 8.1 (2021), pp. 1–34.

[136] Jason Wei and Kai Zou. "EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Nov. 2019.

[137] Gözde Gül Sahin and Mark Steedman. "Data Augmentation via Dependency Tree Morphing for Low-Resource Languages". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Ed. by Ellen Riloff et al. Association for Computational Linguistics, 2018, pp. 5004–5009.

[138] Ge Yan et al. "Data Augmentation for Deep Learning of Judgment Documents". In: *International Conference on Intelligent Science and Big Data Engineering*. Springer. 2019, pp. 232–242.

[139] Hongyu Guo, Yongyi Mao, and Richong Zhang. "Augmenting data with mixup for sentence classification: An empirical study". In: *arXiv preprint arXiv:1905.08941* (2019).

[140] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Improving Neural Machine Translation Models with Monolingual Data". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Aug. 2016.

[141] Yu Li et al. "A Diverse Data Augmentation Strategy for Low-Resource Neural Machine Translation". In: *Information* 11.5 (2020).

[142] Ashutosh Kumar et al. "Submodular Optimization-based Diverse Paraphrasing and its Effectiveness in Data Augmentation". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computa-*

tional Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). June 2019.

[143] Qizhe Xie et al. *Unsupervised Data Augmentation for Consistency Training*. 2020. arXiv: 1904.12848 [cs.LG].

[144] Alexander Fabbri et al. "Improving Zero and Few-Shot Abstractive Summarization with Intermediate Fine-tuning and Data Augmentation". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. June 2021.

[145] Ateret Anaby-Tavor et al. "Do not have enough data? Deep learning to the rescue!" In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 05. 2020, pp. 7383–7390.

[146] Husam Quteineh, Spyridon Samothrakis, and Richard Sutcliffe. "Textual Data Augmentation for Efficient Active Learning on Tiny Datasets". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Nov. 2020.

[147] Venelin Kovatchev et al. "Can vectors read minds better than experts? Comparing data augmentation strategies for the automated scoring of children's mindreading ability". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Aug. 2021.

[148] Yanru Qu et al. *CoDA: Contrast-enhanced and Diversity-promoting Data Augmentation for Natural Language Understanding*. 2020. arXiv: 2010.08670 [cs.CL].

[149] Marzieh Fadaee, Arianna Bisazza, and Christof Monz. "Data Augmentation for Low-Resource Neural Machine Translation". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. July 2017.

[150] Kenton Lee et al. *Neural Data Augmentation via Example Extrapolation*. 2021. arXiv: `2102.01335 [cs.CL]`.

[151] Jason Wei et al. "Few-Shot Text Classification with Triplet Networks, Data Augmentation, and Curriculum Learning". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2021).

[152] Varun Kumar et al. *A Closer Look At Feature Space Data Augmentation For Few-Shot Intent Classification*. 2019. arXiv: `1910.04176 [cs.CL]`.

[153] Rahul Gupta, Aditya Kanade, and Shirish Shevade. "Neural Attribution for Semantic Bug-Localization in Student Programs". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. 2019.

[154] Nicolas Bettenburg et al. "What Makes a Good Bug Report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. 2008.

[155] Ratnadira Widyasari et al. "On the Influence of Biases in Bug Localization: Evaluation and Benchmark". In: *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), RENE Track.* SANER 2022. 2022.

[156] Toan Q Nguyen, Kenton Murray, and David Chiang. "Data Augmentation by Concatenation for Low-Resource Translation: A Mystery and a Solution". In: *arXiv preprint arXiv:2105.01691* (2021).

[157] Rahul Premraj et al. “Extracting structural information from bug reports”. In: *Proceedings of the 2008 international workshop on Mining software repositories - MSR 2008*. 2008.

[158] Yang Song and Oscar Chaparro. “BEE: A Tool for Structuring and Analyzing Bug Reports”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.

[159] Agnieszka Ciborowska and Kostadin Damevski. *Fast Changeset-based Bug Localization with BERT*. 2021. arXiv: `2112.14169 [cs.SE]`.

[160] Michael Pradel et al. “Scaffle: bug localization on millions of files”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020.

[161] Matteo Paltenghi and Michael Pradel. “Thinking Like a Developer? Comparing the Attention of Humans with Neural Models of Code”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 867–879.

[162] Oscar Chaparro et al. “Detecting Missing Information in Bug Descriptions”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. 2017.

[163] Nathan Ng et al. “Facebook FAIR’s WMT19 News Translation Task Submission”. In: *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*. Aug. 2019.

[164] Xueqing Wu et al. “mixSeq: A Simple Data Augmentation Methodfor Neural Machine Translation”. In: *Proceedings of the 18th International Conference on Spoken Language Translation*. IWSLT 2021. 2021.

[165] Jiaao Chen et al. “Hiddencut: Simple data augmentation for natural language understanding with better generalizability”. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2021, pp. 4380–4390.

[166] Dinghan Shen et al. “A Simple but Tough-to-Beat Data Augmentation Approach for Natural Language Understanding and Generation”. In: *arXiv preprint arXiv:2009.13818* (2020).

[167] Gemma Catolino et al. “Not all bugs are the same: Understanding, characterizing, and classifying bug types”. In: *Journal of Systems and Software* (2019).

[168] Rahul Yedida and Tim Menzies. “On the value of oversampling for deep learning in software defect prediction”. In: *IEEE Transactions on Software Engineering* (2021).

[169] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. “When Do Changes Induce Fixes?” In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. MSR ’05. 2005.

[170] Github. *The 2021 State of the Octoverse*. 2021. URL: `https://octoverse.github.com/#top-languages-over-the-years`.

[171] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. DOI: 10.48550/ARXIV.2004.05150. URL: https://arxiv.org/abs/2004.05150.

[172] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. "Reformer: The efficient transformer". In: *arXiv preprint arXiv:2001.04451* (2020).

[173] Wei Wang et al. "Structbert: Incorporating language structures into pre-training for deep language understanding". In: *arXiv preprint arXiv:1908.04577* (2019).

[174] Wen Tai et al. "exBERT: Extending Pre-trained Models with Domain-specific Vocabulary Under Constrained Training Resources". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Nov. 2020, pp. 1433–1439.

[175] Timo Schick and Hinrich Schütze. "Generating Datasets with Pretrained Language Models". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Nov. 2021.

[176] Kang Min Yoo et al. "GPT3Mix: Leveraging Large-scale Language Models for Text Augmentation". In: *Findings of the Association for Computational Linguistics: EMNLP 2021*. Nov. 2021.

[177] Supriyo Chakraborty et al. "Interpretability of deep learning models: A survey of results". In: *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. 2017.

Vita

Agnieszka Ciborowska received her BSc. in Computer Science in 2016 and her MSc. in Computer Science in 2017, both from the University of Science and Technology in Wroclaw, Poland. As a full-time graduate student in the Ph.D. program at Virginia Commonwealth University, her research is focused on context-aware information retrieval systems for the software engineering domain.

Publications:

[1] [Under revision] A. Ciborowska, K. Damevski. *Not Enough Bug Reports? Exploring Data Augmentation for Improved DL-based Bug Localization.* IEEE Transactions on Software Engineering, 2022

[2] A. Ciborowska, K. Damevski. *Fast Changeset-based Bug Localization with BERT.* In proceedings of the 44th International Conference on Software Engineering (ICSE'22), Technical Track, virtual event, 2022.

[3] A. Ciborowska, A. Chakarov, R. Pandita. *Contemporary COBOL: Developers' Perspectives on Defects and Defect Location.* In proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME'21), Research Track, virtual event, 2021. **IEEE Computer Society TCSE Distinguished Paper Award**

[4] A. Ciborowska, M. J. Decker, K. Damevski. *Online Adaptable Bug Localization for Rapidly Evolving Software.*
https://arxiv.org/abs/2203.03544

[5] M. M. Imran, A. Ciborowska, K. Damevski. *Automatically Selecting Follow-up Questions for Deficient Bug Reports.* In proceedings of the 18th International Conference on Mining Software Repositories (MSR'21), Technical Track, virtual event, 2021.

[6] A. Ciborowska, K. Damevski. *Software artifacts retrieval based on changesets.* In proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME'20), Doctoral Symposium, virtual event, 2020.

[7] A. Ciborowska, K. Damevski. *Recognizing Developer Activity Based on Joint Modeling of Code and Command Interactions.* IEEE Access, 2020.

[8] H. Chen, A. Ciborowska, K. Damevski. *Using Automated Prompts for Student Reflection on Computer Security Concepts.* In proceedings of the 24th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'19), Aberdeen, UK, 2019.

[9] M.A. Nishi, A. Ciborowska, K. Damevski. *Characterizing Duplicate Code Snippets between Stack Overflow and Tutorials.* In proceedings of the 16th International Conference on Mining Software Repositories (MSR'19) – Mining Challenge, Montreal, Canada, 2019.

[10] A. Ciborowska, N. Kraft, K. Damevski. *Detecting and Characterizing Developer Behavior Following Opportunistic Reuse of Code Snippets from the Web.* In proceedings of the 15th International Conference on Mining Software Repositories (MSR'18) – Mining Challenge, Gothenburg, Sweden, 2018. **MSR'18 Mining Challenge Winner**