Fast Changeset-based Bug Localization with BERT

Agnieszka Ciborowska
Virginia Commonwealth University
Department of Computer Science
Richmond, VA, USA
ciborowskaa@vcu.edu

Kostadin Damevski
Virginia Commonwealth University
Department of Computer Science
Richmond, VA, USA
kdamevski@vcu.edu

ABSTRACT

Automatically localizing software bugs to the changesets that induced them has the potential to improve software developer efficiency and to positively affect software quality. To facilitate this automation, a bug report has to be effectively matched with source code changes, even when a significant lexical gap exists between natural language used to describe the bug and identifier naming practices used by developers. To bridge this gap, we need techniques that are able to capture software engineering-specific and project-specific semantics in order to detect relatedness between the two types of documents that goes beyond exact term matching. Popular transformer-based deep learning architectures, such as BERT, excel at leveraging contextual information, hence appear to be a suitable candidate for the task. However, BERT-like models are computationally expensive, which precludes them from being used in an environment where response time is important.

In this paper, we describe how BERT can be made fast enough to be applicable to changeset-based bug localization. We also explore several design decisions in using BERT for this purpose, including how best to encode changesets and how to match bug reports to individual changes for improved accuracy. We compare the accuracy and performance of our model to a non-contextual baseline (i.e., vector space model) and BERT-based architectures previously used in software engineering. Our evaluation results demonstrate advantages in using the proposed BERT model compared to the baselines, especially for bug reports that lack any hints about related code elements.

KEYWORDS

bug localization, changesets, information retrieval, BERT

ACM Reference Format:
Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast Changeset-based Bug Localization with BERT. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510042

1 INTRODUCTION

Two of the most prevalent tools used today by software engineers are repositories to store project files (e.g., git) and bug trackers to report and monitor bug fixing activity (e.g., JIRA, BugZilla). Automatically linking a bug report in a bug tracker and related software artifacts from a repository 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, i.e., *bug localization* [35, 62]. However, 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 [32]. 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 [64]. More work is needed to improve the retrieval quality of bug localization techniques.

At the same time, 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 [33]. One key characteristic found beneficial in modern software projects is bug-inducing changeset- (or commit-) level 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 buggy source code elements due to the potentially large number of commits in the corpus.

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 on a much smaller dataset towards the specific task they are to be used for. Transformer-based models pre-trained on large software engineering corpora (e.g., StackOverflow) are now becoming available [48], with the potential to improve software engineering tasks like bug localization. In this paper, we use the BERT (Bidirectional Encoder Representations from Transformers) transformer-based architecture, which is a highly popular model introduced by Devlin et al. [9].

Bug localization is usually framed as an Information Retrieval (IR) task, where a document (i.e., a software artifact) is retrieved from a corpus-based on a query (i.e., the bug report text). A measure of semantic relatedness between the bug report and the software

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510042

artifact is necessary to rank the results retrieved from the corpus. Given the fact that transformer-based models consist of many neural layers and require heavy computation for each sentence, measuring relatedness between the query and the corpus quickly becomes expensive.

This paper applies BERT to 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.

Our experimental results indicate that the proposed approach improves upon popular bug localization techniques by, e.g., increasing the retrieval accuracy between 5.5% and 20.6% for bug reports with no or a limited number of localization hints. We note that using entire changesets as input granularity significantly hinders the models performance, while leveraging more fine grained input data, such as hunks, results in the highest retrieval quality. We also observe that the size of search space (i.e., the number of changesets in a project) significantly impacts the retrieval delay of different BERT-based models, though less in the case of the proposed model.

The main contributions of this paper are:

- **approach that applies BERT to the bug localization problem** (specifically, localizing bug-inducing changesets) that is more accurate than the state-of-the-art,
- **improvement over other recent BERT-based architectures** proposed towards changeset retrieval, showing significant advantages with respect to retrieval speed,
- **evaluation and recommendations for key design choices** in applying BERT to changesets (i.e., code change encoding, data granularity).

Significance of contribution. The BERT-based technique proposed in this paper enables semantic retrieval of software artifacts (specifically, changesets) for bug localization that goes beyond (and can complement) the exact term matching in the current popular state-of-the-art techniques (e.g., [45, 57]). Relative to a similar, recent BERT-based technique [27], we offer an approach that improves retrieval speed significantly, in a way that supports real-world use, while also enhancing retrieval quality.

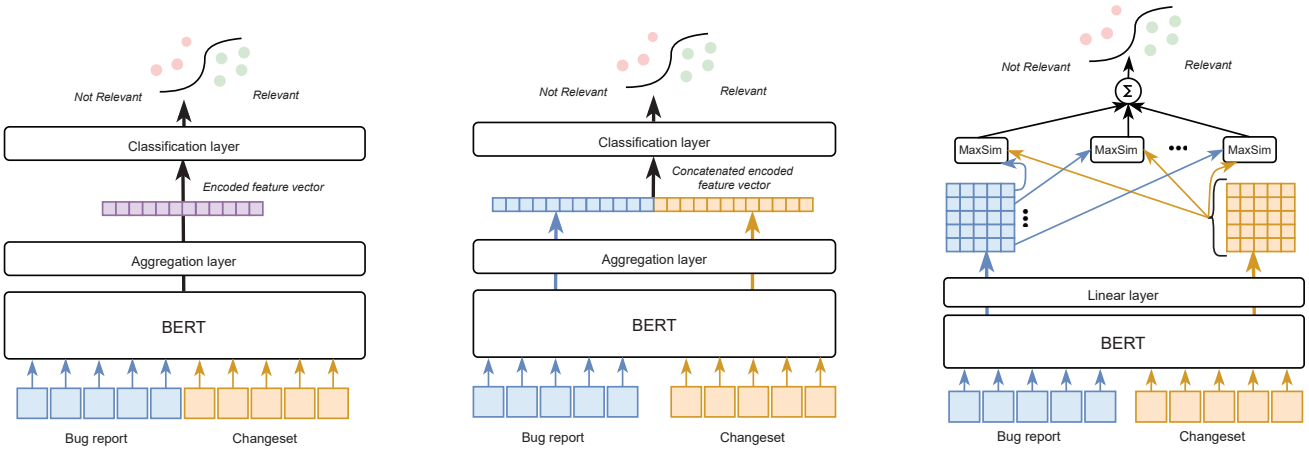
2 PROBLEM DESCRIPTION

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 [33, 64]. 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.

1. Localizing changesets [33]. Over the years, a large body of research has been dedicated to locating source code files (or classes) relevant to a bug report [7, 22, 36, 45, 55, 57]. 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 [33, 56, 64]. Adjusting for this finding, researchers shifted their efforts towards more fine grained code elements, such as file segments [57] and methods [50, 59, 61], 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 [7, 27, 56, 58] for bug localization because 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. 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 white spaces).

2. Leveraging semantics of input documents [33, 48]. 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 traditional IR systems based solely on token similarity [12]. 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 bug reports with large semantic gaps with the code base is likely more valuable to developers. For instance, one surveyed developer in the study by Zou et al. [48] stated the following about current bug localization, *"It seems that existing techniques mainly make use of the textual similarity between bug reports and source code files to perform bug localization. However, I encountered many bugs that have very little similarity between their bug reports and code files. I wonder what kind of bugs such techniques can localize? Maybe only simple bugs?"*. To bridge this gap, researchers have recently proposed to use deep learning models capable of building semantically rich document representations [4, 6, 12, 16, 24, 27, 33]. Transformer-based models, 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 contextual information encoded in the preceding and succeeding tokens, which leads to richer semantics that is more likely to detect related pairs of bug reports and changesets that do not share terms. Prior generations of word embeddings, e.g., word2vec [31] and GloVe [38], which have been frequently applied on software engineering tasks [5], do not use word context at inference time, i.e., each token maps to a vector regardless of the surrounding text.

3. Fast retrieval in a large search space [40]. 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



(a) Single BERT [8, 27, 37] **(b) Siamese BERT [27, 41]** **(c) FBL-BERT [21]**

Figure 1: BERT-based architectures for changesets retrieval

commits [43, 46], 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.

3 APPROACH

In order to address the above problem constraints, in this paper we investigate the use of a BERT model towards bug localization with changesets as a primary data granularity, which is also preferred by practitioners' (§2.1)[52]. We specifically selected BERT as it is the state-of-the-art in semantics modeling and extracting contextual information (§2.2). 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. Replication package is available at [1].

3.1 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. [32], 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 language 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 reuse 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

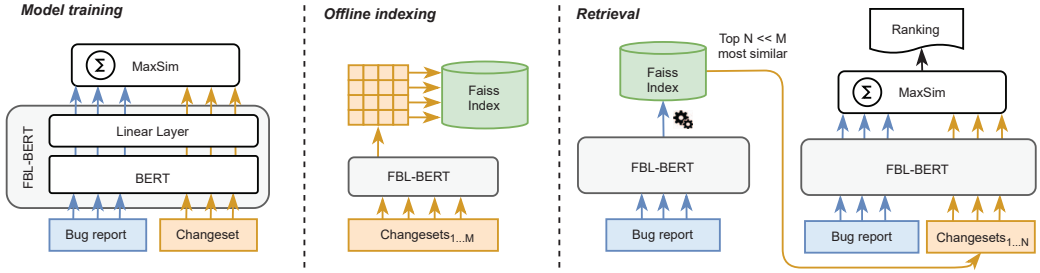


Figure 2: FBL-BERT for changeset-based bug localization pipeline.

pointed by Sachdev et al. [44], this simple aggregation strategy leads to a dissipative data representation that has the potential to negatively affect retrieval performance. In the next section, we describe an alternative strategy that takes advantage of the full matrix to encode input data.

In the simplest changeset retrieval scenario, presented in Fig. 1a, each newly 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 [8, 27, 37]. In an alternative retrieval architecture, called Siamese BERT [27, 41] and depicted in Fig. 1b, the bug report and the changeset are processed separately, 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.

3.2 Fast Bug Localization BERT

The FBL-BERT architecture, based on ColBERT by Khattab et al. [21], 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. 1c. 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 [9]). This modification is dictated by prior studies observing that that different layers of BERT encode different granularity of semantic information [9, 39, 51]. 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, 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. 2. First, as shown in the Model Training section of Fig. 2, 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 [19]. 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.

3.3 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 [10]), providing detailed code-change information to a machine learning model may affect the model negatively [60], 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.

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 [47]. 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 [48]. Secondly, BERT uses a pre-defined set of special tokens. In general, due to how BERT is trained (more details in [9]), 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. 3.

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

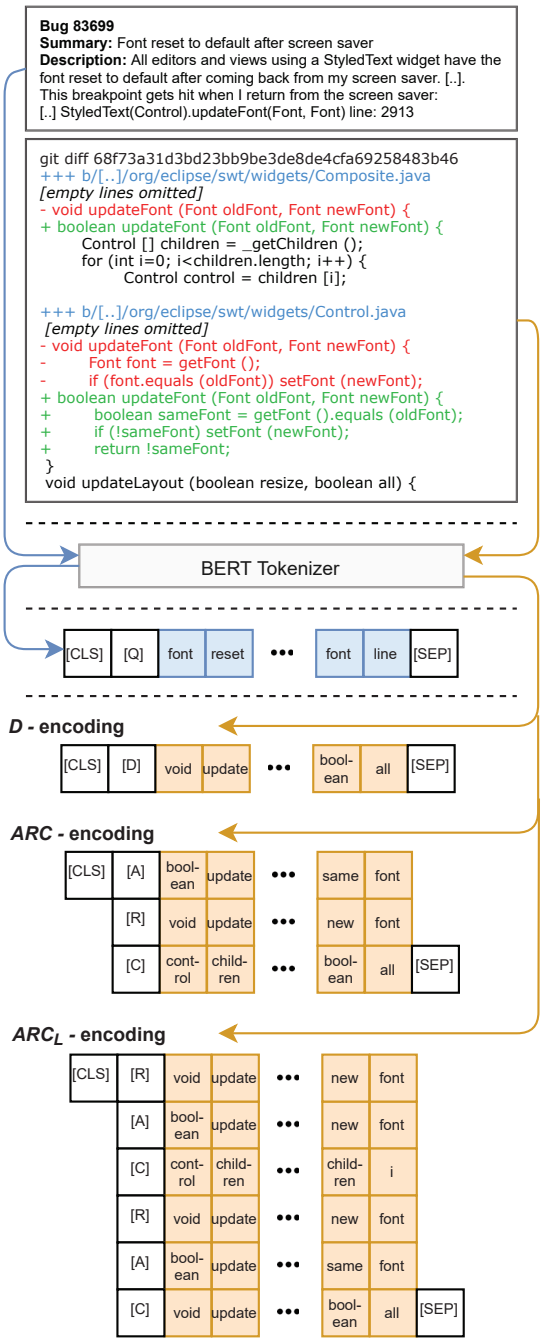


Figure 3: Changeset encoding strategies.

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.

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 [57, 59], thus the inherent fine granularity of hunks makes them less susceptible to noise when compared to whole source code files [56]. Second, dividing changesets into hunks alleviates issues caused by tangled commits [13]. 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 [48]. 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 [3, 27, 33].

4 EXPERIMENTAL EVALUATION

4.1 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 [17]. We follow the methodology proposed by Kochhar et al. [23] 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 [56], a state-of-the-art approach based on VSM that locates bug-inducing changesets, and (2) TBERT-Single and TBERT-Siamese [27] 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.

4.2 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. [56]; manually validated datasets remove the error that can be introduced by the SZZ algorithm that maps the bug fixing to the inducing commit [34]. This dataset includes 6 software projects, namely AspectJ, JDT, PDE, SWT, Tomcat and ZXing (descriptive statistics are presented in Table 1). 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 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 3.2). 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 [56], which is an unsupervised model that utilizes hunk-level

Table 1: Projects in evaluation dataset.

	#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

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 [18, 29]. As a second set of baselines, we employ TBERT architectures for software artifacts retrieval recently proposed by Lin et al. [27]. 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. 1 with an exception of using more advanced embedding aggregation operators [27].

4.3 Metrics

To evaluate the performance of the model, we employ a set metrics commonly used to evaluate performance of IR systems.

Mean Reciprocal Rank: MRR quantifies the ability of a model to locate the first relevant changeset to a bug report. The metric is calculated as an average of reciprocal ranks across B bug reports, while a reciprocal rank for a bug report B_i is equal to an inverted rank of the first relevant changeset in the ranking:

$$MRR = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{1stRank_{B_i}}.$$

Mean Average Precision: MAP measures how well a model can locate all changesets relevant to a bug report. MAP is calculated as the mean of average precision values ($AvgP$) for B bug reports, while average precision for a bug report B_i, $AvgP_{B_i}$, is computed based on the positions of all relevant changesets in the ranking:

$$MAP = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{AvgP_{B_i}}.$$

Precision@K: P@K evaluates how many of the top-K changeset in a ranking are relevant to a bug report. The value of P@K is equal to the number of relevant changesets $|Rel_{B_i}|$ located in the top-K position in the ranking averaged across B bug reports:

$$P@n = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{|Rel_{B_i}|}{K}.$$

4.4 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 [49]. 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 [9]. 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 changesets for re-ranking with FBL-BERT [21]. 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.

Table 2: 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**

5 RESULTS

5.1 RQ1: Retrieval performance

Retrieval accuracy. Table 2 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}) [23]. 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 evaluation projects, as shown in Section 5.2.

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. 4, 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 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.

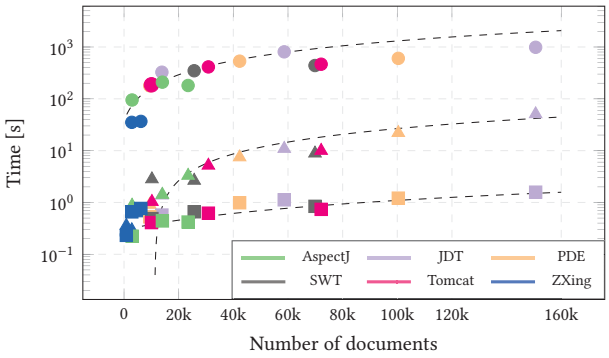


Figure 4: Average retrieval time per a bug report with different sizes of search space (● TBERT-Single, ▲ TBERT-Siamese, ■ FBL-BERT).

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 [53]. For 7 out of 20 bug reports,

Table 3: 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*

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 [20]. 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 [2, 11, 26].

5.2 RQ2: Changeset encoding strategy

Table 3 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 [14], 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 performed 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.

5.3 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 [9, 21]. 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 [23].

Leveraging changesets for bug localization poses another threat due to possible noise that can be introduced by SZZ [42], which could result in poor quality mapping between bug reports and bug-inducing changesets. However, the dataset was validated manually [56, 62], and therefore such mistakes, if they still exist, should not significantly affect our conclusions. Errors due to tangled changes [14, 30] 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 [45, 56, 57]. Another threat to external validity is in the chosen evaluation metrics, which may not directly gauge user satisfaction with our bug localization technique [54], 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.

6 RELATED WORK

Bug localization has generated significant research interest over the years. In this section, first, we survey related code element-based bug localization techniques, followed by approaches towards bug-inducing changeset retrieval. Finally, we review methods for encoding changesets characteristics.

6.1 Code element-based bug localization

Bug localization techniques predominantly utilize information retrieval 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. To compute similarity between bug reports and source code, the Vector Space Model (VSM) is often used as one of the simplest and effective information retrieval algorithms, which is leveraged by many bug localization techniques. For instance, BugLocator [22] combines two rankings, one produced by similarity between the bug report and code elements and another based on similarity of the bug report to prior fixed bug reports. BLUiR [45] uses code and bug report structure to create groups of terms and computes similarity between different groups separately, while AmaLgam [55] creates an ensemble consisting of BugLocator, BLUiR and a defect predictor leveraging the development history of a project. BRTracer [57] focuses on analyzing and prioritizing stack traces when they are included in bug reports. 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 [23]. VSM-based techniques are likely to perform well on such bug reports, though localizing them may not be as useful to developers [48]. 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 [32]. However, their findings do not preclude additional accuracy improvements by using more complex, semantic models, such as BERT.

More recently, software engineering researchers have been interested in the applying deep learning techniques towards bug localization. For instance, TRANP-CNN [17] is a recent technique that combines cross-project transfer learning and convolutional neural networks to achieve state-of-the-art performance on file-level bug localization. CooBa improves on TRANP-CNN by combining a shared encoder to capture cross-project with per-project features and using adversarial training to ensure that the per-project information remains unaffected by noise [63]. Lam et al.'s technique, DNNLOC, combines a deep neural network with the VSM in order to be effective across different types of similarity [25]. While we also leverage a deep learning model, BERT is significantly different from these prior techniques. Recent work in bug localization also includes reports on the value of retrieving changesets instead of source code elements [33, 56].

6.2 Changeset-based bug localization

The earliest work on changeset-based bug localization is Locus [56], which is based on VSM matching of bug reports to hunks. 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. [3] introduced Orca, a tool that uses a provenance graph to

identify commits leading to faulty builds. ChangeLocator [58] uses 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. One of the benefits of VSM is that it is an unsupervised approach, hence a training corpus of bug reports and their inducing commits is not required. However, as VSM fundamentally requires at least partial token overlap, while it ignores the context in which tokens appear in documents, all bug localization technique based on it have a limited accuracy ceiling [33].

Recently, researchers have also shifted their attention to deep learning models for changeset-based localization. For instance, Murali et al. [33] 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. [27] studied the tradeoffs between different BERT architectures for the purpose of changeset retrieval, and observed the accuracy of Siamese architecture is on pair with Single-BERT architecture, while being significantly faster. However, the speed and interactivity of these models is not on par with the BERT technique described in this paper.

6.3 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 prediction. Approaches that define novel changeset embeddings (vector representations of changeset), including CC2Vec [15] and Commit2Vec [28], leverage the difference between added and removed lines of code, among other changeset characteristics. Corley et al. [7] studied how including different types of lines from a changeset affects the performance of Latent Dirichlet Allocation-based feature location, observing that including context, additions, and log messages, but excluding removed lines, achieves the best performance. However, these studies did not utilize a transfer learning technique, like BERT, which requires compatibility with a pre-trained model, and also prior work did not extensively explore hunks as a primary data dimension.

7 CONCLUSION

This paper presents 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. [21], which speeds up the retrieval of results while performing fine grained matching across all embeddings in the two documents. The results show an improvement in retrieval accuracy for bug reports that lack localization hints or have only partial hints. We also evaluate different approaches for utilizing changesets in BERT-like models, producing recommendations on the input data granularity and the use of special tokens for the purpose of capturing changeset semantics.

REFERENCES

[1] 2020. *Replication package.* https://anonymous.4open.science/r/fbl-bert-D567/README.md

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

[3] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-scale Services. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. 493–509.

[4] J. Cao, S. Yang, W. Jiang, H. Zeng, B. Shen, and H. Zhong. 2020. BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

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

[6] S. Cheng, X. Yan, and A. A. Khan. 2020. 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)*.

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

[8] Zhuyun Dai and Jamie Callan. 2019. 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)*.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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*.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[11] Yu Gu, Robert Tinn, Hao Cheng, Michael Lucas, Naoto Usuyama, Xiaodong Liu, Tristan Naumann, Jianfeng Gao, and Hoifung Poon. 2021. Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing. arXiv:2007.15779 [cs.CL]

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

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

[14] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. 121–130.

[15] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

[16] X. Huo, F. Thung, M. Li, D. Lo, and S. Shi. 2019. Deep Transfer Bug Localization. *IEEE Transactions on Software Engineering* (2019).

[17] X. Huo, F. Thung, M. Li, D. Lo, and S. Shi. 2019. Deep Transfer Bug Localization. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2920771

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

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

[20] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Open-Vocabulary Models for Source Code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. 294–295.

[21] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT *(SIGIR '20)*.

[22] D. Kim, Y. Tao, S. Kim, and A. Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Transactions on Software Engineering* 39, 11 (Nov 2013), 1597–1610.

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

[24] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2015. 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)*. 476–481.

[25] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*

(Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, 218–229. https://doi.org/10.1109/ICPC.2017.24

[26] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2019. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics* (Sep 2019). https://doi.org/10.1093/bioinformatics/btz682

[27] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability Transformed: Generating more Accurate Links with Pre-Trained BERT Models. arXiv:2102.04411 [cs.SE]

[28] Rocio Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. 2019. Commit2Vec: Learning Distributed Representations of Code Changes. arXiv:1911.07605

[29] Walid Maalej and Hans-Jörg Happel. 2010. Can development work describe itself?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*.

[30] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An Empirical Study of Build Maintenance Effort. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. 141–150.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.).

[32] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. 2020. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering* 25 (2020).

[33] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2020. Industry-scale IR-based Bug Localization: A Perspective from Facebook. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*.

[34] E. C. Neto, D. A. da Costa, and U. Kulesza. 2018. 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)*.

[35] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. 2011. 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)*. 263–272. https://doi.org/10.1109/ASE.2011.6100062

[36] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. 2011. 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)*. 263–272.

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

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

[39] Matthew Peters, Mark Neumann, Luke Zettlemoyer, and Wen-tau Yih. 2018. Dissecting Contextual Word Embeddings: Architecture and Representation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

[40] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug Localization on Millions of Files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*.

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

[42] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. arXiv:2102.03300 [cs.SE]

[43] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit Guru: Analytics and Risk Prediction of Software Commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 966–969.

[44] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *Proceedings of the 2nd Workshop on Machine Learning and Programming Language (MAPL 2018)*.

[45] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving Bug Localization Using Structured Information Retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. 345–355.

[46] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 21–30.

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

[48] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and Named Entity Recognition in StackOverflow. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4913–4926. https://doi.org/10.18653/v1/2020.acl-main.443

[49] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and Named Entity Recognition in StackOverflow. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

[50] Chakkrit Tantithamthavorn, Surafel Lemma Abebe, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. 2018. The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology* (2018).

[51] Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT rediscovers the classical NLP pipeline. *arXiv preprint arXiv:1905.05950* (2019).

[52] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* 46, 11 (2020).

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

[54] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. 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). 1–11.

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

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

[57] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. 181–190.

[58] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: Locate Crash-Inducing Changes Based on Crash Reports. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*.

[59] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. 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* (Hong Kong, China) *(FSE 2014)*. 689–699.

[60] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep Just-in-Time Defect Prediction: How Far Are We? *(ISSTA 2021)*.

[61] Wen Zhang, Ziqiang Li, Qing Wang, and Juan Li. 2019. FineLocator: A novel approach to method-level fine-grained bug localization by query expansion. *Information and Software Technology* 110 (2019), 121–135.

[62] J. Zhou, H. Zhang, and D. Lo. 2012. 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)*. 14–24. https://doi.org/10.1109/ICSE.2012.6227210

[63] Ziye Zhu, Y. Li, Hanghang Tong, and Yu Wang. 2020. CooBa: Cross-project Bug Localization via Adversarial Transfer Learning. In *IJCAI*.

[64] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. 2020. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Transactions on Software Engineering* 46, 8 (2020).