Bug Localization by Learning to Rank and Represent Bug Inducing Changes

Pablo Loyola, Kugamoorthy Gajananan and Fumiko Satoh
IBM Research Tokyo
Tokyo, Japan
{e57095,gajan,sfumiko}@jp.ibm.com

ABSTRACT

In software development, bug localization is the process finding portions of source code associated to a submitted bug report. This task has been modeled as an information retrieval task at source code file, where the report is the query. In this work, we propose a model that, instead of working at file level, learns feature representations from source changes extracted from the project history at both syntactic and code change dependency perspectives to support bug localization.

To that end, we structured an end-to-end architecture able to integrate feature learning and ranking between sets of bug reports and source code changes.

We evaluated our model against the state of the art of bug localization on several real world software projects obtaining competitive results in both intra-project and cross-project settings. Besides the positive results in terms of model accuracy, as we are giving the developer not only the location of the bug associated to the report, but also the change that introduced, we believe this could give a broader context for supporting fixing tasks.

CCS CONCEPTS

• **Information systems** → **Information retrieval**; **Language models**; Learning to rank; • **Software and its engineering** → *Software evolution*; *Maintaining software*;

KEYWORDS

Bug Localization; Information Retrieval; Source Code Changes

ACM Reference Format:
Pablo Loyola, Kugamoorthy Gajananan and Fumiko Satoh. 2018. Bug Localization by Learning to Rank and Represent Bug Inducing Changes. In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18), October 22–26, 2018, Torino, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3269206.3271811

1 INTRODUCTION

Bug reports play a critical role in software engineering as they represent the primary way of communication between developers and end users [31]. A bug report consists of a portion of text, usually written in natural language, which is submitted to a bug tracker by an end user. Its content usually exposes a mismatch between the result of the execution of a component and its intended behavior [34].

After a bug report is submitted, a member of the development team proceeds to investigate it, a process usually called *bug localization* [41]. This task consists of firstly understand the report and then try to locate the specific portion of the program that is relevant to the reported issue, i.e., the files or modules that are more likely to contain the defect. At this time, the developer can use any auxiliary information, such as his own knowledge about the system or previous report history.

Bug localization can be relatively simple when the project is small, but in large scale projects, the number and complexity of both the reports and the source code files make any manual search infeasible [35]. Therefore, several automated bug localization techniques have been proposed over the years. These approaches share the same goal, which is to transfer a logic derived from natural language (the report) into the functional semantics of the source code.

In general, bug localization has been modeled as an information retrieval task, where the bug report is treated as a *query* and the set of sources code files that conform the system are *documents*. Therefore, the goal is to select the files that better match the query based on a defined similarity metric.

In this regard, reviewing the state of the art we found two main ways to operationalize such paradigm. The first one is what we can call a *similarity* based approach consists of mapping both the reports and source code files into a common feature space, and then choose the files more related to a report by computing a similarity metric. The second way is a *learning* based approach, which assumes the existence of a training set consisting of previous bug reports and their associated defective source code files. Within this learning-based paradigm, we can also visualize two main alternatives to treat the problem.

The first learning-based approach, a *ranking* based localization, consists of firstly compute a set of features for each bug report-source code file pair and then associate them to a relevance label. At training time, this resulting set is used for learning a score function, which iteratively updates the weights associated to each feature in a standard *learning to rank* framework [17]. At test time, for each new pair that is input to the model a score is obtained, allowing a subsequent ranking. One of the main issues of this paradigm is that both feature extraction and ranking are treated as isolated processes, therefore, there is no direct way to communicate the feedback from the ranking task to the feature learning process. Moreover, the process of feature construction has been mostly

CIKM '18, October 22–26, 2018, Torino, Italy

ACM ISBN 978-1-4503-6014-2/18/10…$15.00
https://doi.org/10.1145/3269206.3271811

conducted in a manual fashion, which naturally incorporate biases that could impact on model performance.

The second learning paradigm, *classification* based localization, proposes a model that consists of two feature extractors, one for bug reports and the other for source code files. The output from each extractor (usually a dense vector) is merged into a single representation that is passed to standard classifier to label the pair as relevant or not. Recent approaches within this paradigm have proposed ways to jointly perform feature learning and classification, guiding the feature learning process towards generating more expressive inputs for the classifier. One of the main issues of this paradigm is that the data set is extremely unbalanced, as for each bug report that is introduced, the relevant source code files are just a small fraction (if not a single one) of the total number of available files. This makes it necessary to incorporate strong regularization components and also perform ad-hoc modifications in the cost function in order to obtain a reasonable degree of generalization. This problem is especially prevalent when working in the intersection between natural and programming languages, given the disparity in terms of sequence lengths and vocabulary sizes.

Besides the decisions related to the learning paradigm used, bug localization approaches mainly work relating bug reports to source code files. While this can be seen as the natural way to proceed, we consider that, from a practical point of view, providing to the developers only the source code file associated to the bug report does not give enough context to actually fix the associated bug, as the developer will know *where* is the problem, but does not have a clue on *what* caused it [36].

Rather than working at the source code file level, we consider it would be more convenient to do it at source code change level, i.e., to identify the code change that was responsible for *introducing* the bug into the system. As each change is associated to a source code file, the developer not only will know *where* is the bug, but also *what* caused it, easing the fixing tasks (for example, reversing a specific change).

Having explored and analyzed the aforementioned issues, in this work we propose an approach that focuses on the efficient learning of feature representations from code changes as they key element to support bug localization. To achieve that, the model learns feature representations of code changes from a *syntactic* perspective, by learning the distributions of source code tokens through a recurrent language model, as well as from a change *dependency* perspective, by incorporating the inherent relationships between changes. To that end, the history of code changes is automatically transformed into a directed acyclic graph from which a graph embedding approach learns a feature representation for each change considering its context associated to the development activity.

Our hypothesis is that if learn simultaneously from these two perspectives, we could improve the expressiveness of the changes, as we are capturing not only the characteristics of the changes in terms of the content modified, but also in terms of the dynamics related to the incremental nature of software development. Simultaneously, the model learns feature representations for the bug reports by means of an attention-based recurrent module that considers both word and character level granularities.

For structuring an end-to-end training process, we consider taking the most beneficial elements from each localization paradigms described above: we are following a learn to rank inspired approach, but where the learning is performed in a gradient based fashion, allowing us to directly backpropagate the errors from the ranking task to both code change and bug reports feature learning modules.

We evaluate our approach on two datasets of bug reports and change history data from several real world Open Source systems. Our results show that the proposed architecture is able to efficiently learn unified feature representations for both natural language and source code at change level, allowing it to outperform current state of the art bug localization techniques. Moreover, we explored a cross project setting, in order to assess the ability of the model to transfer knowledge to an unseen project, showing also consistent results.

2 RELATED WORK

Bug localization has been an important line of research in the intersection between software engineering and natural language processing. In terms of the way the problem has been modeled, we can identify two main paradigms, which we can name as *similarity* based localization and *learning* based localization.

In similarity based localization, as the goal is to map both reports and source code files into a shared feature space, feature engineering plays a critical aspect to consider. Most approaches treat source code tokens as a natural language, and therefore several text analysis techniques has been exploited, including frequency weighting schemes [31], and topic models [25]. Additionally, combining several sources of data [34, 35, 41].

These attempts, while providing a relevant insights to developers, suffer largely from not considering the order of the tokens, i.e., the program *structure*. Additionally, considering features based only the term frequencies has the issue of sparsity, which naturally makes it more difficult to generalize in a statistical learning setting.

Learning based localization puts its efforts in making sense of the high amounts of data available in project repositories. For ranking based learning methods, we can mention the work of Ye et al [39] defining a set of syntax based features, and recently extending it by incorporating learned token embeddings [40]. Considering test case execution data, we can mention the work of Le et al. [1], where they rank class methods by considering suspiciousness scores and likely invariant diffs.

In the case of classification based learning, recent approaches have explored the use of representation learning to unify both natural language and source code into one dimensional space that could allow the extraction of distributed feature representations. The main assumption is that features obtained through these methods could be more expressive and flexible, based on their distributed nature. In that sense, the work of Huo et al. [14], using convolutional architectures has represented important contribution, as well as a recent extension by incorporating a recurrent layer presented in [13]. Also, the work of Lam et al. [20], using a deep neural network to extend the initial approach of Ye et al. [38].

Besides the original setting that works at source code file level, there has been recent work exploring the relationship between bug inducing changes and bug reports, such as the work of Wen et al. [36]. In such work, authors computed features based on the frequency of token on each change. In our case, we propose to

automatically learn the features from both a syntactic and change dependency perspective.

Code changes has received special interest in recent years, specially in the context of summarization, i.e., trying to characterize a code change and generate a coherent description of it. For example, [6, 22] propose a method based on a set of rules that considers the type and impact of the changes, and [4] combines summarization with symbolic execution [18]. The use of representation learning based models has been also explored recently, such as the work of [23] and [16]. Both approaches make use of an encoder-decoder architecture [5], which receives code change, in the form of a *diff* output[1] and the associated message submitted by the contributor.

The use of representation learning methods in software engineering research has increased in recent years. For example, there are several approaches tackling the problem of source code summarization, i.e., trying to map the functional properties of source code with a natural language description. In that case we can mention the use of recurrent architectures, such as [15]. Defect prediction has also been greatly benefited. The key element in this case resides on the feature learning task, in order to find source code or process representations that could be more expressive and flexible than standard metrics [24].

Nevertheless, the increasing adoption of deep learning methods in software engineering should not be seen as an indicator that this set of methods perform always better than conventionalones. Recently, there have been several studies that question the real applicability and performance in the context of software engineering. That is the case of the work of [9] and [10] which compares the programming language modeling techniques, focusing on the differences between n-grams methods and recurrent models, showing that fine-tuned ngram models can achieve a better performance than deep models.

3 PROPOSED APPROACH

The idea of generating unifying architectures between natural language and source code has been a challenging task for both natural language processing and software engineering research communities. The vision is that a seamless transition between both constructs could improve program understanding tasks, such as code comprehension, summarization. Such obtained knowledge could in turn help developers in more direct tasks such as debugging and repair. While recent research on the *naturalness* of code [11] has shown evidence of their similarities in distributional terms, there are inherent differences such as their structural and stylistic properties. In this work, we rely on representation learning as an alternative to find a common ground between both modalities.

3.1 Overview

For a given software project, we assume the existence of a set of B bug reports, extracted from the bug tracker, and a set of C source code changes, extracted from the commit activity. From these groups we assume a training set can be extracted by looking the history of resolved issues, in the sense of obtaining an explicit relationship between a bug report and the code changes that introduced such fault – for example, as a binary variable that takes value 1 if a given report-change is related and 0 if not. Then, our goal is, given a new bug report which has not been seen by our model, estimate which source code changes are more likely responsible for introducing the related fault, and as each change is associated to a file, obtain the file that contains the bug.

Our approach, as seen in Figure 1 (b), begins by processing each feasible[2] bug report-source code change pair and learning a unified feature vector representation, which is then passed to a learning-to-rank module, which iteratively estimates a relevance score for the pair. During this training step, the errors produced by the ranking module are backpropagated to the feature learning part, allowing us to automatically adapt the learning process towards more expressive feature representations.

3.2 Feature Learning

3.2.1 Learning Representations from Bug Reports. In the first place, we tokenize each bug report to obtain a fine-grained sequence to model. One thing to keep in mind is that bug reports can contain portions of source code, as the end user can add them to show a specific feature or log associated to intended bug. In other to account for the presence of both source code and natural language, our tokenizer needs to firstly distinguish between them and the tokenize accordingly[3].

The resulting sequence is passed through a token representation layer, which is in charge of associating each token in the report with a vector representation. In this case, this representation is based on both the use of the Glove [29] pre-trained word embeddings through a lookup table (for natural language tokens), and the learning of character level embeddings by splitting each token into its character components and passed through a LSTM module [12], from which the last hidden state is captured. The decision of incorporating a character-level feature extractor comes from recent evidence on the benefits of combining word level and character level representations, especially in the case of low frequency words [21, 37]. For tokens that are not identified as part of natural language or that does not appear on the Glove vectors, we define a random initialization.

The final representation of each token is then the concatenation of the token and character level representations. This is shown as the green sequence in Figure 1 (b).

The next step consists of generating a context rich representation of the sequence of embedded tokens. To do that, we rely on a bidirectional LSTM that reads the sequence and returns a hidden state at each step. More formally, given a sequence $S = \{t_1, \ldots, t_N\}$, let h_i be the context-rich word representation associated to the embedded token t_i, defined by the concatenation $h_i = [\overrightarrow{h}_i, \overleftarrow{h}_i]$ where $\overrightarrow{h}_i = LSTM(t_i, \overrightarrow{h}_{i-1})$ and $\overleftarrow{h}_i = LSTM(t_i, \overleftarrow{h}_{i-1})$ are the forward and backward passes of the bidirectional LSTM, respectively. As each sequence is of different size, the resulting representations

[1] The diff command in Unix allows to compare two documents. See http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html

[2] Feasible in the sense that a bug report can only be associated to a change if the timestamp of the report is bigger than the timestamp of the change.

[3] In practical terms we adapted a tool called *Pygments*, which allows us to identify source code portions via the use of lexers. See pygments.org

might vary in size. Therefore, we force a fixed length by implementing a pooling layer. Additionally, in order to allow the model to learn the specific parts of the sequences that have more expressive power at classification time, we implemented a global attention mechanism [26]:

$$u_i = \boldsymbol{v}^\top \tanh(W[\bar{\boldsymbol{h}}; \boldsymbol{h}_i]) \quad (1)$$

$$\alpha_i = \frac{\exp u_i}{\sum_{k=1}^{n} \exp u_k} \quad (2)$$

$$\boldsymbol{r}' = \sum_{i=1}^{n} \alpha_i \boldsymbol{h}_i \quad (3)$$

where $\boldsymbol{v}$ and W are trainable parameters and $\boldsymbol{r}'$ is the resulting sentence representation from the bug report.

3.2.2 Learning Representations from Code Changes. In the case of the code changes, we divide the process into two tasks, namely, learning from a syntactic and code dependency perspectives. For the first one, we learn feature representations from the source code tokens associated to the code change, i.e., the portion of source code that was modified, treated as a sequence of tokens.

In practical terms, a source code change is the output of applying the diff command between two consecutive versions of a program. This out contains both the lines that were *added* and the ones that were *removed*. We experimented several ways to combine both, for example, simply concatenate them or only selecting the tokens that are not repeated between them. From our exploratory tests, just concatenating added and removed lines gave us better results and we used that to represent a change. Finally, a source code tokenizer is used to obtain all the source code tokens from the change.

For each tokenized change, we also make use of a bilinear LSTM module that receives the sequence of tokens whose vector representations are initialized randomly. For each sequence, we captured the last hidden state and treat it as the feature representation from a syntactic perspective, as shown in a blue block in Figure 1 (b). Let us call this representation c_{syn}.

While learning feature representations of code changes from a syntactic perspective is a natural way to proceed, we are missing a unique characteristic which is their dependency, as software is usually built as a sequence of incremental modifications over time. For example, a change that introduced a new feature could trigger an additional feature built on top of it or even a bug fix.

We consider that modeling that dynamic and incorporating it into the main approach could benefit the degree of expressiveness of the learned representations, as they could provide a broader context for understanding the intent behind the change. In that sense, we propose a way to encode the temporal dependency of the changes.

To that end, structuring the code change activity through a graph can be seen as a natural decision, as it has shown the potential to capture structural and temporal dependencies at different levels of granularity of software development process in the past. One of such approaches is the concept of *code change genealogy*, introduced in [2], which is a graph representation of the code changes introduced to a software system during a defined period of time.

In the original formulation of the code change genealogy, the nodes represent change sets and the edges represent dependencies between them, computed based on a set of defined rules that consider the addition or modification of method calls between changes. To better understand this concept, let us illustrate it with an example, shown in Figure 1 (a). In this case, let c_0 be the initial implementation of a portion of source code, which consists of the addition of two functions, **foo**() and **bar**(). Subsequently, a new change c_i is performed, which consisted of a modification of the function **bar**() (the addition of a new parameter). As the function **bar**() is shared among both changes, a link is generated between them. Following that, we have the change c_j, where the function **foo**() is called. Consequently, we construct an edge between c_0 and c_j as the latter is explicitly using a function defined in c_0. Finally, we have the case of c_k, a change that is calling the method **bar**(), which last modification resides in the change c_i, therefore, we generate an edge between c_i and c_k.

Given that definition, we structure the set of code changes available into a code change genealogy. Then, the way of learning feature representations for each node in this graph resembles the work of Perozzi et al [30], in the sense that we begin computing random walks over the directed graph starting from each node. Random walks are especially convenient in this case, as they are able to capture local information surrounding the starting node and also their computation is easy to parallelize. Additionally, their modularity fits perfectly with the notion of change dependency: a change genealogy is a directed graph that is constantly being modified, as new changes (nodes) are added continuously as development progresses. As random walks have a locality property, it is not necessary to recompute their generation when a new node is added.

Having obtained a set of random walks, we used them as sequences to train a neural embedding model E, i.e., trying to maximize the probability of the code changes appearing in the context (neighborhood), given a defined code change c_i.

Following the standard configuration of a word embeddings model, the probability we want to maximize is:

$$P(c_j|c_i) = \frac{\exp(c_j \top c_i)}{\sum_{w_k \in V} \exp(c_k \top c_i)} \quad (4)$$

Therefore, in each epoch, we minimize the negative log likelihood of the elements surrounding (SU_{c_i}) a given element c_i:

$$J = \sum_{c_i \in t} \sum_{c_j \in SU_{c_i}} -logP(c_j|c_i) \quad (5)$$

The model parameters are optimized through gradient descent where the corresponding derivatives are computed using back-propagation and the specific values for α are obtained through a small validation set. We tested several heuristics such as hierarchical softmax and negative sampling to speed up the overall process. Let us call this learned representation c_{struc}.

At this point we have learned two vector representations for each source code change: one from a purely syntactic perspective, c_{syn} and the other that consider the structural dependency of the change c_{struc}. We then form a single representation for the code change by concatenating them, $\boldsymbol{c}' = [c_{struc}; c_{syn}]$.

3.3 Feature Integration and Ranking

Having learned feature vector representations for a given bug report-code change pair, the next step consists of integrating them into a single vector and pass it to the learning-to-rank module. For

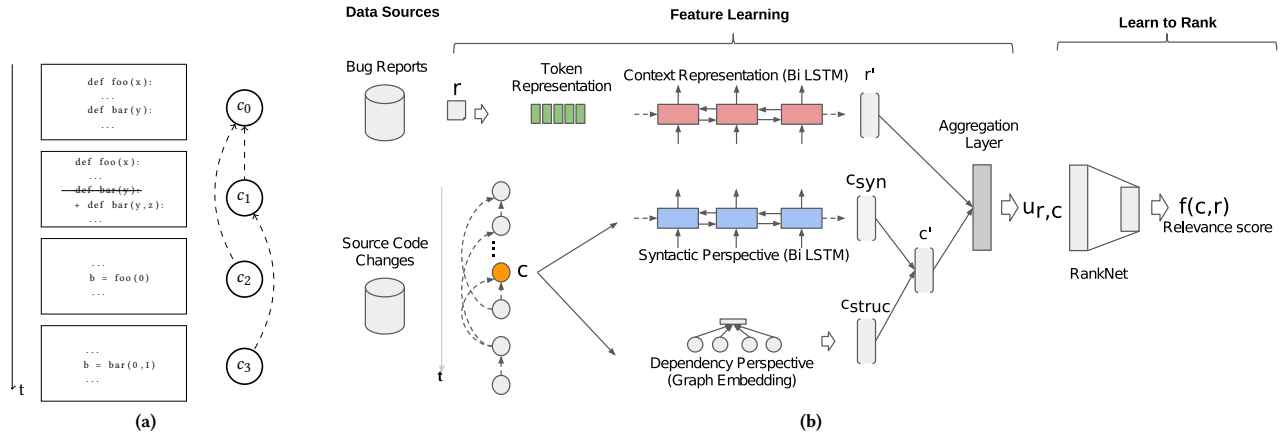

Figure 1: (a) Example of a code change genealogy. (b) Diagram of the proposed approach. For each bug report-source code change pair (r, c), our model jointly learns a feature representation and a relevance score.

this, the most natural way consists of concatenate both vectors. Unfortunately, our initial experiments did not perform well under that configuration. Therefore, we propose an aggregation layer, which consists of the concatenation of the learned representations of both the report $\boldsymbol{r'}$ and the change $\boldsymbol{c'}$, and also two arithmetic operations between them, as proposed by Mou et al. in [28]:

$$\boldsymbol{h}_{mul} = \boldsymbol{r'} \odot \boldsymbol{c'} \tag{6}$$

$$\boldsymbol{h}_{dif} = |\boldsymbol{r'} - \boldsymbol{c'}| \tag{7}$$

$$\boldsymbol{u}_{\boldsymbol{r},\boldsymbol{c}} = [\boldsymbol{r'}; \boldsymbol{c'}; \boldsymbol{h}_{mul}; \boldsymbol{h}_{dif}] \tag{8}$$

with this, we finally obtain $u_{r,c}$ as the integrated feature representation learned from a given bug report-code change pair.

As we want to explicitly relate the feature learning with the ranking task, we adapted a gradient based learn to rank approach, such as RankNet [3]. Under this configuration, the idea is to learn a model such as to minimize the number of incorrect ordering between a pair of instances. Formally, for a given report r and any pair of changes i, j which have associated learned feature vectors u_{r,c_i}, u_{r,c_j}, we want to learn a score function f that returns scores $s_i = f(u_{r,c_i})$ and $s_j = f(u_{r,c_j})$.

Let us assume that –based on a given criteria– that i should be ranked higher than j, therefore such probability can be modeled as $P_{ij} = \frac{1}{1+e^{-\sigma(s_i - s_j)}}$ and the cost we want to minimize is the associated cross entropy:

$$C = -\bar{P}_{ij} \log P_{ij} - (1 - \bar{P}_{ij}) \log(1 - P_{ij}) \tag{9}$$

where $\bar{P}_{ij}$ represents the ground truth probability. Note that assuming a set of discrete relevance labels $S_{ij} \in \{0, \pm 1\}$, $\bar{P}_{ij}$ can be expressed as $\bar{P}_{ij} = 1/2(1 + S_{ij})$. Re-writing the cross entropy replacing P_{ij} and $\bar{P}_{ij}$, we obtain:

$$C = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log(1 + e^{-\sigma(s_i - s_j)}) \tag{10}$$

Then, assuming a model with a learnable module W (which we parametrize with a standard feed forward neural network), we iteratively update the weights w_k trying to minimize the cost via stochastic gradient descent $w_k = w_k - \eta \frac{\partial C}{\partial w_k}$, with η a learning rate estimated. As both the ranking and feature learning modules are explicitly connected, we are able to backpropagate the errors from the ranking task (misclassification) to the feature learning. With this, we are guiding the learned representations to be more expressive towards the ranking and allowing an end-to-end training.

4 EMPIRICAL STUDY

We conducted an empirical study to assess the feasibility and performance of the proposed model. We considered two datasets covering several real world Open source systems whose commit and bug tracker activity have already been captured and we compare against three alternative approaches.

4.1 Data

The first dataset, *Dataset A*, consists of code change level bug localization published by [36]. It contains six open source projects where the relevance score between bug reports and code changes have been mapped.

To construct the second dataset, *Dataset B*, we adapted a source code file level dataset for bug localization, originally provided by Ye et al. [38]. In order to identify bug inducing changes, we implemented the SZZ heuristic proposed by Sliwerski et al. in [32], which works by considering a large set of rules derived from regular expressions. One element to take into consideration is that this heuristic is known to present some issues in the presence of noisy data, as reported in [7]. Nevertheless, we performed extensive manual sampling of the labeled pairs in order to guarantee a reasonable degree of the correctness.

For linking bug reports to changes, we follow a standard technique such as Dallmeier et al. [8], where the main idea is search on change description for patterns such as "Fixed issue 123" (bug fixing changes), where "123" is the identifier of the bug report. Such matching is performed also via sets of regular expressions.

Table 1 provides the number of bug reports, files and changes associated to each project for each dataset, as well as Avg I-D and

Avg O-D, the average in and out degree values for the resulting code change genealogies generated based on the total change history.

All the source code in both datasets in written in Java. We performed checking on the changes to make sure they were modifying actual source code files and not auxiliary files (such as README files).

	Project	Reports	Changes	Files	Avg I-D	Avg O-D
Dataset A	Zxing	20	3.140	391	5.9	4.2
	SWT	98	1.190	484	17.6	13.1
	AspectJ	244	770	6.485	12.7	17.7
	PDE	60	1.130	5.273	9.5	5.6
	JDT	94	2.170	6.775	16.5	14.2
	Tomcat	193	1.610	2.042	10.9	15.7
Dataset B	AspectJ	393	1.420	6.502	16.4	19.3
	Birt	100	800	3.117	13.1	12.6
	JDT	227	2.961	6.794	19.5	14.8

Table 1: Datasets statistics

4.2 Alternative Methods

We reimplemented three bug localization methods to compare against our model. **BLUiR** proposed by Saha et al. [31] and **Locus**, proposed by Wen et al. [36]. Both approaches work by boosting a vector space model and generating an explicit unified feature space between source code and natural language. In the specific case of Locus, it was designed to work at code change level.

Additionally, we implemented a version of **LS-CNN**, a recent work of Huo et al. [13], where are convolutional feature extractor is combined with a recurrent model to learn feature representations from source code.

4.3 Evaluation Metrics

Following standard studies on bug localization, we propose to use the Mean Average Precision (MAP), which given a query, takes the relevant answers associated to the ranking and computes the average scores, the Mean Reciprocal Rank (MRR), which is the average of the reciprocal ranks of results of a set of queries, and the Top-N Rank, which is the number of bugs whose associated changes are ranked in the top N ($N = 1, 5$) of the returned results.

4.4 Experiment Design Overview

For character-level learning, we initilized character-level embeddings with 40-dimensional vectors. The character-level LSTM hidden and output dimensions were also set to 40. In the case of the token-level learning (bug report learning module) we used the pretrained 300-dimensional GloVe vectors [29]. The word-level LSTM output and hidden states were set to 300.

For the code change feature learning, the recurrent model was configured with a 300 dimensional output. The feature leaning from the dependency graph considers the setting of the length of the random walk, which varies for each project, but on average was set as 60, and the context window, which was set as 10. RMSProp [33] was used as optimizer for all the recurrent models in the architecture.

One important element during the evaluation is how partition the data between training and testing. Both code changes and bug reports have a time component, which needs to be taken into consideration. For example, it is not valid to try to associate a bug report with a code change if the date of creation of the bug report is older than the creation date of the change. Therefore, there is a need to align both sources of data. In general, a bug report should be associated with code changes introduced before it. With that in mind, we firstly sorted the data sources by time of creation and then we generated chunks of align data across the full dataset, trying to keep a 80-20 ratio between the portion that will be used for training and the one for testing, respectively[4].

The reported metrics are then the average across all chunks of tests sets.

5 RESULTS AND DISCUSSION

Tables 2 and 3 shows the classification performance on both datasets. As it can be seen, the proposed approach achieves better performance in the majority of the projects on both datasets. In the case of JDT and PDE, the differences are not significant between our approach and LS-CNN. When we explored the code bases, we found that these projects have a extreme imbalance in terms of the natural language and source code vocabularies. We hypothesize this could be one of the reasons for this finding as such imbalance is not present in the rest of the projects. Top-1 and Top-5 metrics consistently show that our approach performs better, which is interesting as one practical application could be to implement a recommender system for the developers. In that sense, presenting a feasible set rather than the exact solution seems possible. The consistence among datasets shows that the model is competitive at different scale of data.

While in some instances we cannot see huge improvements over the other approaches implemented, we believe this is due on inherent complexity of the problem as usually one or two source code files are responsible for the bug report, from a set of possibly thousands. Nevertheless, while the performance is the main indicator, we consider the additional information we are providing to the developer, as locating the change responsible for the bug, the developer can now know not only "where" is the bug, but also "why", supporting subsequent fixing tasks.

Graph Feature Learning Variations: Our main interest resides on assessing the modularity of the learned representations from the code dependency graph, as we wanted to explore how better they can work along with the feature representations obtained from the syntactic perspective. To that end, we replaced the original embedding model (Graph Emb) by two other alternatives that consider the use of convolutional filters. We implemented the approach of Mou et al. [27] (TBCNN), which proposes a tree-based convolutional neural network and also the work of Kipf et al. [19] (Graph CNN), which is able to learn feature representations directly from the graph structure. As an example, Table 5 shows the results of those variations against the standard graph embeddings model on Aspect J for Dataset B. As we can see, the graph embedding approach performs slightly better for both MAP and MRR metrics.

[4] A similar partition had to be performed within the resulting training sets to obtain validation sets for model tuning.

Table 2: Performance metrics on Dataset A

Project	Method	MAP	MRR	TOP-1	TOP-5
ZXing	Locus	0.513	0.536	0.406	0.762
	BLUiR	0.405	0.423	0.422	0.583
	LS-CNN	0.561	0.619	0.528	0.839
	Ours	0.572	0.613	0.561	0.896
SWT	Locus	0.618	0.652	0.619	0.773
	BLUiR	0.591	0.593	0.445	0.491
	LS-CNN	0.603	0.617	0.622	0.690
	Ours	0.647	0.675	0.689	0.816
AspectJ	Locus	0.315	0.336	0.257	0.513
	BLUiR	0.286	0.291	0.196	0.492
	LS-CNN	0.310	0.287	0.261	0.494
	Ours	0.321	0.328	0.341	0.627
PDE	Locus	0.440	0.449	0.684	0.716
	BLUiR	0.385	0.382	0.510	0.560
	LS-CNN	0.457	0.428	0.552	0.649
	Ours	0.497	0.501	0.784	0.821
JDT	Locus	0.341	0.361	0.611	0.667
	BLUiR	0.251	0.277	0.598	0.583
	LS-CNN	0.323	0.395	0.472	0.538
	Ours	0.393	0.428	0.651	0.715
Tomcat	Locus	0.570	0.586	0.676	0.762
	BLUiR	0.411	0.442	0.524	0.619
	LS-CNN	0.504	0.511	0.633	0.693
	Ours	0.632	0.661	0.790	0.823

Table 3: Performance metrics on Dataset B

Project	Method	MAP	MRR	TOP-1	TOP-5
AspectJ	Locus	0.395	0.401	0.402	0.437
	BLUiR	0.337	0.381	0.461	0.482
	LS-CNN	0.455	0.448	0.502	0.562
	Ours	0.462	0.451	0.514	0.591
Birt	Locus	0.305	0.341	0.418	0.466
	BLUiR	0.294	0.289	0.341	0.362
	LS-CNN	0.376	0.383	0.480	0.492
	Ours	0.401	0.494	0.518	0.533
JDT	Locus	0.436	0.467	0.479	0.562
	BLUiR	0.381	0.339	0.462	0.481
	LS-CNN	0.463	0.477	0.539	0.561
	Ours	0.494	0.481	0.551	0.582

Only the tree-based approach (TBCNN) seems to be comparable, but in our experiments the time required for such approach is around 60% higher, making it unfeasible in a real world scenario.

Random walk length: The most sensitive parameter we found on the code dependency perspective is the length of the random walks. We held out a small set for model validation, showing that in general the increasing the random walk has a positive effect, as can be seen in Figure 2 for Dataset B. The drawback is related to the increase in computation time. Additionally, as the random walks are obtained from a directed graph, we empirically found that increasing their size make them converge to the same sequence, which introduces repeated instances on the feature learning module, something that is not desirable as it introduces frequency bias (over representation of certain elements).

Cross-project Setting: Assuming the existence of a relevant training dataset is not always realistic, as projects may have not enough historical data. For example, if we want to apply our system in a very new project, which only have few dozens of changes and similar number of bug reports, where the project is not small enough the perform manual search (no need for automated bug localization) but not big enough to train effectively a localization model?

In that sense, studying a cross-project configuration is interesting as it could open the door to transfer learning from a data source to an unknown setting. We performed a cross-project experiment in which we held out one project and train our model with the rest (for the code change perspective, the code genealogies are treated separately, but the source code portions are merged, learning from an unified vocabulary). As this is a more challenging task, we tested this configuration on Dataset B, which contains more data per project, and we compare against LS-CNN as it is the state of the art for learning based approaches.

Results shown on Table 4, show that for AspectJ and Birt, while there is an overall decrease in performance , our approach is still able to outperform LS-CNN in all the projects, especially in the case of Birt. For the specific case of JDT both models perform poorly as the number of changes is much higher and therefore the model is not able to transfer effectively. One of the possible reasons we found is the low intersection between the merged vocabularies of AspectJ and Birt and the vocabulary from JDT.

Table 4: Performance metrics for cross-project experiment on Dataset B.

Project	Method	MAP	MRR
AspectJ	LS-CNN	0.352	0.373
	Ours	0.371	0.411
Birt	LS-CNN	0.331	0.302
	Ours	0.340	0.348
JDT	LS-CNN	0.119	0.118
	Ours	0.121	0.128

Table 5: Impact of graph feature learning methods.

Name	MAP	MRR
TBCNN	0.317	0.310
Graph CNN	0.285	0.288
Graph Emb	0.321	0.328

5.1 Impact of the Code Change Dependency Module

One of the main differences between the proposed model and the state of the art in bug localization is the addition of the module capable of learning features from the code change dependencies. Our hypothesis is that such addition is useful as it gives more context to the code change, given the incremental nature of software development. One natural way to verify such assumption is to compare the performance of the model with and without the code

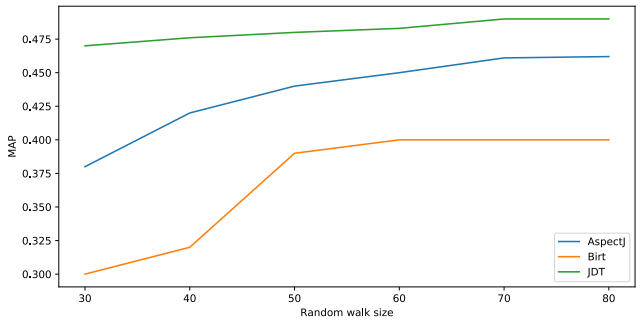

Figure 2: Impact of random walk size on MAP for Dataset B.

change dependency module, i.e, without computing c_{struct} as seen on Figure 1 (b).

Table 6 shows the results of removing the change dependency module for the projects on the Dataset B. In this case, we can see that removing the such module always decreases the performance of the model. This phenomenon is present in all projects and within all metrics computed. This finding provides us a strong insight of the importance of considering the change dependencies.

Table 6: Performance when removing the change dependency module on the Dataset B.

Project	Method	MAP	MRR	TOP-1	TOP-5
AspectJ	Ours NO Change Dep	0.402	0.397	0.422	0.441
	Ours	0.462	0.451	0.514	0.591
Birt	Ours NO Change Dep	0.381	0.354	0.444	0.451
	Ours	0.401	0.494	0.518	0.533
JDT	Ours NO Change Dep	0.402	0.398	0.499	0.517
	Ours	0.494	0.481	0.551	0.582

5.2 Impact of the Joint Learning Configuration

One of the main goals of this approach is to generate an end-to-end training model, where classification errors from the learn-to-rank task can be back-propagated to the feature learning process.

In order to test such assumption, we implemented a variation of our model where the feature learning process is isolated from the ranking task, which means, we do not backtrack the errors from the ranking task into the feature learning process. In this case, all feature extractors (for modeling the sequence of tokens from both bug reports and code changes) learn representations just maximizing the likelihood of an item given its context (not receiving any other signal).

The results of such variation show that and end-to-end training is more effective across all projects in the two datasets presented. In the case of dataset A, the end-to-end configuration reaches up to a 21% better MAP for the SWT project, while on average the differences are about 12% and 15% in terms of MAP and MRR respectively. In the case of Dataset B, the results are consistent, also showing an average increase of more than 10% for both metrics. This result gives us the idea that the combination between the feature learning and ranking task by backtracking the errors is feasible, and could open the door to further model extensions. Our initial concern was that the proposed architecture incorporates several aggregation steps, so we were wondering if the signal from the classification module was actually impacting on the feature learning process.

6 CONCLUSION AND FUTURE WORK

We proposed a model that learns a comprehensive feature representation from code changes to support for bug localization. Our results on datasets from several real world software systems suggest that the idea is feasible, being competitive against the state of the art. The idea of integrating the change dependency as a source of feature learning impacts positively on the performance of the model for the majority of the projects evaluated. This is aligned with our belief that the software development process is not a group of isolated actions, but a dependent sequence of changes, where each of them takes into account the previous history in an incremental way.

We consider that advancing on analytical tools that can automate software related task could help developers to focus on the creative process of code writing rather than large amounts of time on maintenance and debugging tasks.

For future work, our plan is to work on the interpretability of the learned representations, as we consider it is critical for suggesting changes into the project management. In that regard, we think a necessary task is to perform a human study to evaluate our approach from a more qualitative perspective, i.e., given a bug report, ask a developer if the output from our model is relevant or not and if it provides enough contextual information to support fixing tasks.

Additionally, we consider relevant the inclusion of other feature representations, such as program traces, in the sense of incorporating program behavior information. This can be extracted from test suite executions. All these represents an interesting challenge from a modeling perspective in the sense of finding the most expressive way to merge data from different dimensions.

REFERENCES

[1] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 177–188.

[2] Irina Ioana Brudaru and Andreas Zeller. 2008. What is the long-term impact of changes?. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*. ACM, 30–32.

[3] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 89–96.

[4] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 33–42.

[5] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[6] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 275–284.

[7] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2017. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2017), 641–657.

[8] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 433–436.

[9] Wei Fu and Tim Menzies. 2017. Easy over Hard: A Case Study on Deep Learning. *arXiv preprint arXiv:1703.00133* (2017).

[10] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.

[11] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[13] Xuan Huo and Ming Li. [n. d.]. Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code. ([n. d.]).

[14] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code.. In *IJCAI*. 1606–1612.

[15] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model.. In *ACL (1)*.

[16] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 135–146.

[17] Thorsten Joachims. 2002. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 133–142.

[18] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[19] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[20] 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*. IEEE Press, 218–229.

[21] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360* (2016).

[22] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 709–712.

[23] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Vol. 2. 287–292.

[24] Pablo Loyola and Yutaka Matsuo. 2017. Learning graph representations for defect prediction. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 265–267.

[25] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972–990.

[26] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[27] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.. In *AAAI*. 1287–1293.

[28] Lili Mou, Hao Peng, Ge Li, Yan Xu, Lu Zhang, and Zhi Jin. 2015. Discriminative neural sentence modeling by tree-based convolution. *arXiv preprint arXiv:1504.01106* (2015).

[29] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[30] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 701–710.

[31] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 345–355.

[32] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147

[33] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012), 26–31.

[34] 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*. ACM, 53–63.

[35] Shaowei Wang and David Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.

[36] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 262–273.

[37] Zhilin Yang, Bhuwan Dhingra, Ye Yuan, Junjie Hu, William W Cohen, and Ruslan Salakhutdinov. 2016. Words or characters? fine-grained gating for reading comprehension. *arXiv preprint arXiv:1611.01724* (2016).

[38] 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*. ACM, 689–699.

[39] Xin Ye, Razvan Bunescu, and Chang Liu. 2016. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering* 42, 4 (2016), 379–402.

[40] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 404–415.

[41] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 14–24.