BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs

Junming Cao, Shouliang Yang, Wenhui Jiang, Hushuang Zeng, Beijun Shen, Hao Zhong
{junmingcao,ysl0108,swingteki,zenghushuang,bjshen,zhonghao}@sjtu.edu.cn
Shanghai Jiao Tong University, China

ABSTRACT

Given a bug report of a project, the task of locating the faults of the bug report is called *fault localization*. To help programmers in the fault localization process, many approaches have been proposed, and have achieved promising results to locate faulty files. However, it is still challenging to locate faulty methods, because many methods are short and do not have sufficient details to determine whether they are faulty. In this paper, we present BugPecker, a novel approach to locate faulty methods based on its deep learning on revision graphs. Its key idea includes (1) building revision graphs and capturing the details of past fixes as much as possible, and (2) discovering relations inside our revision graphs to expand the details for methods and calculating various features to assist our ranking. We have implemented BugPecker, and evaluated it on three open source projects. The early results show that BugPecker achieves a mean average precision (MAP) of 0.263 and mean reciprocal rank (MRR) of 0.291, which improve the prior approaches significantly. For example, BugPecker improves the MAP values of all three projects by five times, compared with two recent approaches such as DNNLoc-m and BLIA 1.5.

ACM Reference Format:
Junming Cao, Shouliang Yang, Wenhui Jiang, Hushuang Zeng, Beijun Shen, Hao Zhong. 2020. BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3418934

1 INTRODUCTION

With the increase of software development's scale and complex, endless emerging bugs cost a huge amount of development time and effort. It is rather difficult and time-consuming for programmers to locate faulty code precisely, especially in projects with many source files. Therefore, given a bug report, various *fault localization* techniques have been proposed to identify its faulty files [27]. The identified faulty files could either be delivered to programmers for repairs, or serve as the inputs of *automated program repair* [8].

Generally, *fault localization approaches* could be divided into two mainstreams [9]. The first line of approaches is spectra-based fault localization, which analyzes the program execution information of passing and failing test cases [10]. However, as introduced by Anil *et al.* [8], test cases are usually not available when bugs are reported. In contrast, the second line of approaches compare bug reports with source files, and identify similar source files as faulty files [20, 21, 25]. These approaches are known as IR-based fault localization, and they consider the fault localization as a search problem: bug reports are queries and faulty files are answers. Most IR-based approaches localize faulty files [9, 20, 25]. Only a few recent approaches [21, 23] localize faulty methods, but their effectiveness is less impressive. For example, the evaluation of a recent approach [23] shows that the MAP values on Ant are only 0.0498 (Youm et al. [21]), 0.0494 (Zhang et al. [24]), and 0.0550 (Zhang et al. [23]). It is challenging to improve the prior approaches, because some methods are short and lack sufficient details to be matched against bug reports.

In this paper, we present BugPecker, a novel approach to locate faulty methods. Our early results in Section 3 show that BugPecker significantly improves the state-of-the-art approaches [9, 21]. For example, **the MAP on Tomcat is increased from 0.017 of Lam et al. [9] and 0.005 of Youm et al. [21] to our 0.121**. BugPecker makes significant improvements, because it introduces deep learning and is the first to encode the commits and bug reports into revision graphs. As shown in Table 1, revision graphs are graphs whose nodes are bug reports, commits, files, and methods and whose edges denote their relations.

Figure 1 shows an overview of BugPecker. It consists of three components: (1) the *revision analyzer* constructs revision graphs from past fixes; (2) the *semantic matcher* calculates semantic similarity scores between bug reports and methods; and (3) the DNN *learner* locates faulty methods based the scores.

This paper makes the following contributions:

(1) Analyzing past fixes as revision graphs. We construct revision graphs through analyzing code, commits, and bug reports. Furthermore, we leverage the graphs to enrich method details by incorporating its related methods, and calculate various features to assist the recommendation.

(2) Learning source files via abstract syntax trees (ASTs). AST based code representation could capture both the lexical (i.e. the leaf nodes of ASTs such as identifiers) and syntactical (i.e. the non-leaf nodes of ASTs like the grammar construct *WhileStatement*) information [22]. To our best knowledge, BugPecker is the first to introduce AST parsing into IR-based fault localization.

(3) An open source tool and evaluations. We have released BugPecker on Github, and evaluated it on three open source projects. The results show that BugPecker could localize bugs at method level more precisely than the baselines, achieving a *MAP* of 0.263 and a *MRR* of 0.291.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418934

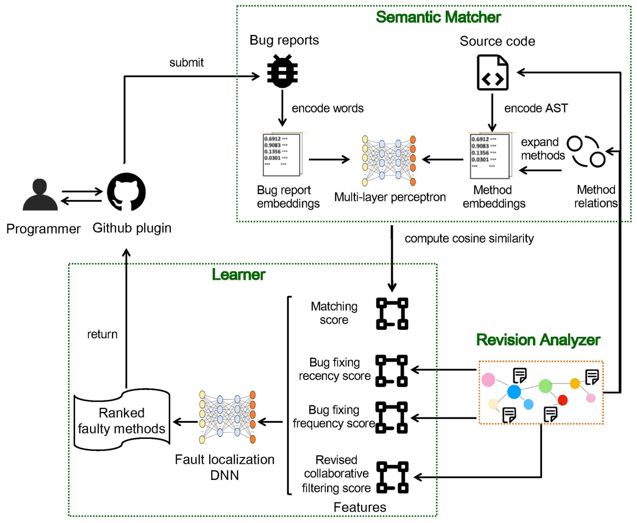

Figure 1: An overview of BugPecker

2 BUGPECKER

We next explain the three components of BugPecker.

2.1 The revision analyzer

The analyzer builds revision graphs from code, commits and past bug reports. As shown in Table 1, revision graphs define the following entities and relations between them: repository, bug report, commit, file and method. As the code related entities, file and method both have a "version" attribute.

The analyzer extracts the *has*, *modify* relations from the repository; uses Spoon [16] to extract *call* relations between methods; and adopts the approach presented by Dallmeier and Zimmermann [4] to extract the *fix* relations between code commits and bug reports. The *modify* relations between a bug report and methods are inferred from the *modify* relations between a bug report and its commits.

In particular, the analyzer applies a structural-context similarity, *SimRank* [6], to identify the *similar-to* relations. Let S^b_{ij} denote the similarity between bug reports b_i and b_j, and S^m_{ij} denote the similarity between methods m_i and m_j. *SimRank* has the following steps: In Step 1, S^b_{ii} and S^m_{ii} are set as 1 and will not be updated later. All other similarity scores are initialized as 0. In Step 2, similarity scores are updated iteratively (5 rounds of iteration) by the following equations:

$$S^b_{ij} = \frac{C}{|M(b_i)||M(b_j)|} \sum_{k=1}^{|M(b_i)|} \sum_{l=1}^{|M(b_j)|} S^m_{kl}\,, \tag{1}$$

$$S^m_{ij} = \frac{C}{|B(m_i)||B(m_j)|} \sum_{k=1}^{|B(m_i)|} \sum_{l=1}^{|B(m_j)|} S^b_{kl}\,, \tag{2}$$

where $M(b_i)$ is the set of methods modified by b_i, $B(m_i)$ is the set of bug reports that modified m_i, and C is the rate of decay as similarity flows in the revision graphs, which is set 0.8. In Step 3, we pick the pairs that have a similarity score larger than 0.001 (except S^b_{ii} and S^m_{ii}) as similar bug reports and methods. Instead of content based similarity measurement, *SimRank* can work well for short methods lacking sufficient semantic information.

Table 1: Relations in revision graphs

Relation	Bug report	Commit	File	Method
Repository	has	has	has	has
Bug report	similar-to	-	-	modify
Commit	fix	similar-to	modify	modify
File	-	-	-	has
Method	-	-	-	similar-to, call

Except similar-to, all the other relations are directed.

The revision graphs offer the following benefits to BugPecker:

(1) It enables to alleviate the method information insufficient problem. As revision graphs provide comprehensive relations of methods, the semantic matcher allows matching semantic contents and expanding the details for short methods.

(2) It enables to calculate a revised collaborative filtering feature for bug location ranking. Revision graphs record the relations among methods, commits and bug reports, which are required by the learner to calculate the most influential feature – *revised collaborative filtering score.*

During training or testing, revision graphs also allow obtaining versions (e.g., the version before a fix) and bug reports efficiently.

2.2 The semantic matcher

The matcher calculates the semantic similarity between a method (m) and a bug report (b). We introduce the *ASTNN* [22] model to embed m, which preserves the code structure and semantic information. Meanwhile, we use the *Word2Vec* to embed each word token in b into a vector, and get the global semantic of b by *Bi-GRU*. The embeddings of m and b are not in the same vector space, so it is hard to measure their semantic similarity. Inspired by the idea of domain-invariant feature extraction in transfer learning, we adopt a *multi-layer perceptron* to map the embeddings of methods and bug reports into the same vector space. The cosine similarity is calculated to measure the semantic similarity between the two embeddings. We pre-trained the embeddings of special syntactic symbols (such as identifiers) with the source code of all the projects in the dataset. They are served as initial parameters to train statement vectors in ASTNN. The *Word2Vec* is used to pre-train the embeddings of top 5,000 high-frequency word tokens.

We notice that many methods are short. For example, in Tomcat, there are 46.49% methods within 3 statements and 77.59% methods within 5 statements. Because of the sparsity and ambiguity of information in these short methods, it is challenging to match them with the bug report correctly. Zhang et al. [23] proposed a method expansion algorithm to alleviate this problem. Inspired by this work, we expand short methods (within 5 statements) with related methods. However, the expansion algorithm proposed by [23] applied cosine similarity to retrieve similar methods as expansion information, and consequently for short methods, these so-called similar methods are not similar. Thus, we propose an improved method expansion approach on revision graphs, described as follows. For method m, the top 10 methods with the highest *SimRank similarity score* from revision graphs are collected in M_{sim}, and all methods that m calls are collected in M_{call}. Then, M_{rel}, as the union of M_{sim} and M_{call}, are used to expand methods. Every $m_i \in M_{rel}$ is also embedded by *ASTNN*. Different from directly adding the vectors of methods in [23], which may bring about noisy data, we introduce a soft-attention mechanism [18] to retrieve the useful information u for m, defined as follows:

$$a_i = Softmax\left(\vec{m}^{\mathrm{T}}\vec{m}_i\right), \tag{3}$$

$$\vec{u} = \sum_{i=1}^{|M_{rel}|} a_i \vec{m}_i \ , \tag{4}$$

where $\longrightarrow$ denotes embedding, and a_i denotes the attention probability of m over method m_i. After that, we employ a *Gated Recurrent Unit (GRU)* gate [3] to integrate the embedding of retrieved information $\vec{u}$ into $\vec{m}$. As a result, the embedding of expanded method $\vec{m}^{'}$ becomes richer and denser than the original embedding $\vec{m}$.

$$\vec{q} = \sigma\left(\mathbf{W}^{(q)}\vec{m} + \mathbf{U}^{(q)}\vec{u}\right), \tag{5}$$

$$\vec{r} = \sigma\left(\mathbf{W}^{(r)}\vec{m} + \mathbf{U}^{(r)}\vec{u}\right), \tag{6}$$

$$\vec{u}^{'} = \tanh\left(\mathbf{W}\vec{m} + \vec{r} \circ \mathbf{U}\vec{u}\right), \tag{7}$$

$$\vec{m}^{'} = (1 - \vec{q}) \circ \vec{m} + \vec{q} \circ \vec{u}^{'}, \tag{8}$$

where $\circ$ denotes elementwise multiplication, σ is the sigmoid activation function, tanh is the tanh activation function, $\vec{q}$ is the weighting vector between $\vec{m}$ and $\vec{u}^{'}$, and $\vec{u}^{'}$ denotes the information used to expand $\vec{m}$. The output $\vec{m}^{'}$ is the expanded embeddings of the short input method $\vec{m}$, and the original embedding $\vec{m}$ of method m is replaced by $\vec{m}^{'}$ to calculate the semantic similarity.

2.3 The learner

Given a bug report (b_n), the learner aims to select the most possible faulty methods from the method set (M) and rank them by the suspicious score. Specifically, it extracts four features from revision graphs, applies a non-linear DNN with four hidden layers to get the suspicious score of b_n and each m in M in the repository, and returns the top-ranked faulty methods by this score. Besides the semantic matching score, three features are designed and calculated as the inputs of the DNN model:

(1) Revised collaborative filtering score (*rcfs*). *cfs* calculates the relevant value between a new bug report (b_n) and all methods (M) according to the previous revision history [25]. This model relies on *similar-to* relations between b_n and previous fixed bug reports B^{prev}, and *fix* relations between B^{prev} and M. However, there are no existing *similar-to* relations of b_n, because *SimRank* could only be used for bug reports that we have known their fixed methods. Moreover, our revision graphs do not have many *fix* relations, and can underestimate the similarity values. Therefore, we propose Algorithm 1 to compute *rcfs* for each m in M. In Step 1, we revise the cosine similarity values between b_n and $b_i \in B_{prev}$ with S^b. In Step 3, *cfs* scores are revised into *rcfs* via S^m.

(2) Bug fixing recency score (*bfr*) and frequency score (*bff*). *bfr* measures how recent a method has been fixed, and *bff* measures the bug-fixing frequency for a method. Let $b^{'}$ be the bug report that was most recently fixed in method m before b_n. If the commit time

Algorithm 1: The procedure of calculating *rcfs*

```
Input: $b_n$: a new given bug report
       $M$: all methods in the repository to be localized
       $B^{prev}$: bug reports fixed before $b_n$ (by commit time)
       $S^b_{ij}$: similarity scores between $b_i$ and $b_j$ in $B^{prev}$
       $S^m_{ij}$: similarity scores between $m_i$ and $m_j$ in $M$
       (both $S^b_{ij}$ and $S^m_{ij}$ are provided by revision graphs)
Output: revised cfs score: $rcfs_{m_i}$, for $m_i \in M$
/* Step 1: get $S^b_{ni}$, for $b_i \in B^{prev}$ */
for $b_i \in B^{prev}$ do
    $S^b_{ni} = \sum_{j=1}^{|B^{prev}|} S^b_{ij} * cosSim(b_j, b_n) + cosSim(b_i, b_n)$
/* Step 2: get $cfs_{m_i}$, for $m_i \in M$ */
$M_{b_i}$ denotes methods that have been modified by $b_i$
for $m_i \in M$ do
    for $b_j \in B^{prev}$ do
        if $m_i \in M_{b_j}$ then
            add $b_j$ into $B_{m_i}$
    $cfs_{m_i} = \sum_{j=1}^{|B_{m_i}|} S^b_{nj} * \frac{1}{|M_{b_j}|}, b_j \in B_{m_i}$
/* Step 3: revise $cfs_{m_i}$ with $S^m$ */
for $m_i \in M$ do
    $rcfs_{m_i} = \sum_{j=1}^{|M|} cfs_{m_j} * S^m_{ij} + cfs_{m_i}$
```

of $b^{'}$ was earlier than b_n by k month, then *bfr* for method m can be calculated as:

$$bfr = \frac{1}{k+1}, \tag{9}$$

and *bff* can be scored as the times that B^{prev} are fixed in m. We normalize the matching scores and features to be within [0,1].

3 EARLY RESULT

In our early study, we explore the following two research questions:

RQ1. Can BugPecker outperform existing fault localization approaches at method level?

RQ2. How does method expansion, *rcfs*, *bfr* and *bfs* contribute to the fault localization performance of BugPecker?

We have implemented the BugPecker tool as a Github plugin [2] in Java. Our implementation and dataset have been published on Github: https://github.com/RAddRiceee/BugPecker.

3.1 Setup

Dataset. The benchmark dataset created by [20] from three open-source projects (AspectJ, SWT and Tomcat) are used for our evaluation. The prior studies [7, 14] show that some bug reports already describe which files and methods are faulty, and we call such bug reports as *localized bug reports*. As faults of these bug reports are already revealed, they do not need any fault localization technique to determine their faulty files or methods. As Kochhar et al. [7] did, we remove those *localized bug reports* from the dataset of our evaluation. Table 2 shows the dataset information about each project. We filter out added methods in commits, and consider deleted methods the same as updated methods. For fixes related to multiple methods, we generate multiple positive samples for every method. If at least one ground truth methods appearing in the top ranked faulty methods list, the bug localization succeeds.

Fault localization models are trained with the oldest 80% bug reports, and tested with the newest 20% bug reports. To avoid the model cheating from future ground truths, the *similar-to* and *call*

Table 2: Dataset

Project	Not Localized Bug Reports	Methods	Similar-to (methods)	Similar-to (bug reports)	Call
Tomcat	927	36,569	110,350	1,804	41,224
AspectJ	316	34,670	71,933	811	88,719
Swt	1,491	13,456	19,422	8,376	58,405

relations are discovered only with the training set. In the testing phase, the same set of relations is used. The benchmark dataset records faulty files. We compare them with clean versions and consider modified methods as our true labels.

Metrics. We use *Top-k accuracy*, *Mean Average Precision (MAP)* and *Mean Reciprocal Rank (MRR)* as evaluation metrics, which have been widely used in fault localization and information retrieval [9, 20, 21, 23]. For these metrics, a larger value indicates a better result.

Baselines. We compare BugPecker with two state-of-art approaches:

(1) BLIA 1.5 [21]. It is the most advanced IR-based approach. By integrating the *stack trace feature*, *collaborative filtering feature*, and *commit history feature* with *structural VSM*, it achieves better results than previous IR-based approaches like bugLocator [25]. It provides the localization at method level.

(2) DNNLoc [9]. It proposed a combining approach between rVSM and DNN to learn to connect terms in bug reports to code tokens in source files. As the author did not provide the source code, we use a replication package from Github [1]. The original implementation of DNNLoc is for file level fault localization, thus we adapt it to method level ("DNNLoc-m"). In DNNLoc-m, features of methods, including *rVSM*, *DNN relevancy score*, etc., are computed in the same way of files.

Table 3: Overall results

Project	Approach	Top1(%)	Top5(%)	Top10(%)	MAP	MRR
Tomcat	DNNLoc-m	2.326	5.814	5.814	0.017	0.021
	BLIA 1.5	0	0.108	0.215	0.005	0.005
	BugPecker	12.230	13.669	15.827	0.121	0.143
AspectJ	DNNLoc-m	4.167	10.417	20.833	0.051	0.113
	BLIA 1.5	0.316	1.294	2.916	0.012	0.011
	BugPecker	22.917	27.083	35.417	0.263	0.291
Swt	DNNLoc-m	5.34	16.087	20.434	0.058	0.102
	BLIA 1.5	1.073	2.481	3.689	0.016	0.018
	BugPecker	23.913	28.261	36.956	0.253	0.267

3.2 Results to RQ1

Table 3 presents the comparison results of BugPecker, BLIA 1.5 and DNNLoc-m. DNNLoc-m achieves better results than BLIA 1.5. On our dataset and setting, BLIA 1.5 achieves much poorer than what were reported in their paper [21]. There are two reasons for this: (1) We evaluate with *not localized bug reports*. Without hints in bug reports, IR-based fault localization approaches suffer from a more severe *semantic gap* problem. (2) BLIA 1.5 only picks methods in the top-10 localized files as candidates. Indeed, we did not compare with FineLocator [23], because it is not open source. Although they evaluate BLIA 1.5 on different projects, their results are consistent with ours. Besides, FineLocator makes only marginal improvements over BLIA 1.5. For example, the MAP is improved from 0.0666 to 0.0741 on 94 Aspectj bug reports, and from 0.0409 to 0.0624 on 87 Maven bug reports.

Compared with their results, Table 3 shows that for all three projects, BugPecker improves the MAP values by five times and all the other measures are much better than the baselines.

Table 4: The impacts of our techniques on Tomcat

Feature	Top1(%)	△	MAP	△	MRR	△
complete model	12.230	-	0.121	-	0.143	-
w/o rcfs	7.194	-5.036	0.074	-0.047	0.087	-0.056
w/o bfr	11.511	-0.719	0.113	-0.008	0.127	-0.016
w/o bff	8.633	-3.587	0.085	-0.036	0.092	-0.051
w/o method expansion	10.072	-2.158	0.097	-0.024	0.117	-0.026

3.3 Results to RQ2

We analyze the feature contributions to the localization performance through an ablation experiment on Tomcat dataset.

As Table 4 shows, *rcfs* contributes mostly, because it could improve the localization of short methods significantly by exploiting *similar-to* relations in revision graphs. *bff* also plays an important role. This follows the pattern that the frequently fixed methods in the past are the ones that are likely to be fixed in the future. It seems that method expansion is not very helpful. A possible reason is that relations between methods have already been exploited in *rcfs* more explicitly.

4 CONCLUSION AND RESEARCH PLAN

In this paper, we propose BugPecker to locate faulty methods via deep learning on revision graphs. Our early results show that BugPecker outperforms DNNLoc-m and BLIA 1.5 approaches significantly, with a *MAP* of 0.263 and *MRR* of 0.291. To extend this work to a full paper, our research plan is as follows:

1. Obtaining more empirical evidence to show our improvements. Although our early results are quite positive, BugPecker is evaluated on limited subjects and compared with only two recent approaches. To ensure that we make stable improvements, we will compare with more approaches (e.g., FineLocator [23], MULAB [24] and Blizzard [17]) on more subjects and measures (e.g., recall@k). In addition, we will collect more empirical evidence to explain the rationales behind our design, equations, and thresholds.

2. Improving our effectiveness with more features. We plan to extract more features from our revision graphs. First, it can be feasible to extract features from other sources such as Stack Overflow [17, 26], comments [9], and API sequence [5, 12]. Second, it is feasible to extract social network features by analyzing the structures of our revision graphs. Finally, it is worth exploring more features with graph neural networks [11, 15].

3. Learning from revision graphs of other projects. As BugPecker learns from historical bug reports and their fixed methods, it may suffer from a cold start problem for newly established projects. A feasible solution to this problem is to learn from other projects, but we have to overcome two barriers. First, the data from other projects can be quite large, so our algorithms shall be optimized to analyze so large data. Second, the knowledge from other projects may not apply to a new project. We plan to introduce transfer learning techniques [13, 19] to bridge the knowledge gap between a project and its referenced projects.

ACKNOWLEDGMENTS

Beijun Shen is the corresponding author. This research was sponsored by National Key R&D Program of China (Project No. 2018YFB1-003903), and National Nature Science Foundation of China (Grant No. 61472242).

REFERENCES

[1] 2020. The DNNLoc replication package. https://github.com/emredogan7/bug-localization-by-dnn-and-rvsm.

[2] 2020. GitHub Apps. https://developer.github.com/v3/apps/.

[3] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014).

[4] 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*. 433–436.

[5] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. arXiv:1605.08535 [cs.SE]

[6] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *KDD*. ACM, 538–543.

[7] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential biases in bug localization: do they matter?. In *ASE*. ACM, 803–814.

[8] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *ESEC/SIGSOFT FSE*. ACM, 314–325.

[9] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *ICPC*. IEEE Computer Society, 218–229.

[10] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *ISSTA*. ACM, 169–180.

[11] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 162:1–162:30.

[12] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *ESEC/SIGSOFT FSE*. ACM, 120–130.

[13] Jie Lu, Vahid Behbood, Peng Hao, Hua Zuo, Shan Xue, and Guangquan Zhang. 2015. Transfer learning using computational intelligence: A survey. *Knowledge-Based Systems* 80 (2015), 14–23.

[14] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota, and Sonia Haiduc. 2018. Are Bug Reports Enough for Text Retrieval-Based Bug Localization?. In *ICSME*. IEEE Computer Society, 381–392.

[15] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning convolutional neural networks for graphs. In *ICML*. 2014–2023.

[16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[17] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-based bug localization with context-aware query reformulation. In *ESEC/SIGSOFT FSE*. ACM, 621–632.

[18] Jian Tang, Yue Wang, Kai Zheng, and Qiaozhu Mei. 2017. End-to-end Learning for Short Text Expansion. In *KDD*. ACM, 1105–1113.

[19] Shuhan Yan, Beijun Shen, Wenkai Mo, and Ning Li. 2017. Transfer Learning for Cross-Platform Software Crowdsourcing Recommendation. In *APSEC*. IEEE Computer Society, 269–278.

[20] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *SIGSOFT FSE*. ACM, 689–699.

[21] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *Inf. Softw. Technol.* 82 (2017), 177–192.

[22] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*. IEEE / ACM, 783–794.

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

[24] Yun Zhang, David Lo, Xin Xia, Giuseppe Scanniello, Tien-Duy B Le, and Jianling Sun. 2018. Fusing multi-abstraction vector space models for concern localization. *Empirical Software Engineering* 23, 4 (2018), 2279–2322.

[25] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *ICSE*. IEEE Computer Society, 14–24.

[26] Jiangang Zhu, Beijun Shen, Xuyang Cai, and Haofen Wang. 2015. Building a Large-scale Software Programming Taxonomy from Stackoverflow. In *SEKE*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 391–396.

[27] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2018. An Empirical Study of Fault Localization Families and Their Combinations. *CoRR* abs/1803.09939 (2018).