Transactions on Engineering and Computing Sciences - Vol. 12, No. 2
Publication Date: April 25, 2024
DOI:10.14738/tecs.122.16373.

Ahmad, A., Lasheng, Y., Idris, H. A., Aliyu, B., Muhammad, A., & Diginsa, M. U. (2024). AttentiveBugLocator: A Bug Localization Model using Attention-based Semantic Features and Information Retrieval. Transactions on Engineering and Computing Sciences, 12(2). 01-20.

SERVICES FOR SCIENCE AND EDUCATION

AttentiveBugLocator: A Bug Localization Model using Attention-based Semantic Features and Information Retrieval

Aminu Ahmad
School of Computer Science and Engineering,
Central South University, Changsha, Hunan, China

Yu Lasheng
School of Computer Science and Engineering,
Central South University, Changsha, Hunan, China

Hussaini Aliyu Idris
Dept. of Computer Science and Engineering,
Egypt-Japan University of Scienceand
Technology (E-JUST), Alexandria, Egypt

Buhari Aliyu
Dept. of Computer Science,
Jigawa State Polytechnic Dutse, Nigeria

Adamu Muhammad
Dept of Electrical Electronics Engineering,
Jigawa State Polytechnic Dutse, Nigeria

Muhammad Umar Diginsa
Dept. of Computer Engineering Technology,
Bilyaminu Usman Polytechnic Hadejia, Nigeria

ABSTRACT

In recent years, deep learning-based algorithms such as CNN, LSTM, and auto-encoders have been proposed to rank suspicious buggy files. Meanwhile, representational learning has served to be the best ap-proach to extract rich semantic features of bug reports and source code to reduce their lexical mismatch. In this paper, we propose AttentiveBu-gLocator, a Siamese-based representational learning model for improved bug localization performance. AttentiveBugLocator employs BERT and code2vec embedding models to produce richer semantic representations and a Siamese BiLSTM network with context attention to learn semantic matching between BRs and SFs. To further improve the effectiveness of AttentiveBugLocator, the semantic matching features are carefully fused with VSM, stack trace, and code complexity features. Evaluation results on four data sets show that AttentiveBugLocator can identify buggy files on the scale of 56% and 62% on MAP and MRR – thus, outperforms sev-eral state-of-the-art approaches.

Keywords: Bug report, Bug Localization, Deep learning, Software Maintenance, Bug tracking systems, Siamese neural network.

INTRODUCTION

Previously, several techniques have been proposed to locate buggy source files (SFs) with respect to user bug reports (BRs). These techniques are largely based on Information Retrieval (IR) methods such as Vector Space modeling (VSM), Latent Dirichlet allocation (LDA), and Latent Semantic Index (LSI) [19,37,11]. This way, a bug report is used as a query to return the relevant source files based on keywords matching or textual similarity. Moreover, recently, deep learning (DL) approaches such as CNN and RBM are predominantly used to extract semantic features of bug reports and source code [31]; and RNN to improve learning the sequential and structural nature of source code [38]. However, lexical mismatch remains a key issue to the effective-ness of these methods for bug localization. Meanwhile, representational learning has served to be the best approach to extract rich semantic and contextual fea-tures of bug reports and source code [38,16]. Notwithstanding, finding suitable representation to exploit various information - textual and statistical and stack traces - from bug reports, and code complexity and program structure in order to reduce the lexical mismatch remains a key challenge.

Thus, in this paper, we propose AttentiveBugLocator - to investigate the effectiveness of Siamese-based representational learning of bug report and source code features in order to reduce the lexical mismatch problem and improve bug localization performance. Specifically, we utilize a BiLSTM network with context attention to extract highly relevant representations from both BRs and SFs.

We propose distinct encodings to extract unique features from bug report and source code using BERT and code2vec embedding models to produce richer semantic representations, and a Siamese BiLSTM network with context attention to learn semantic matching between BRs and SFs.

We initialize each word using pre-trained BERT, which produces a dynamic context-dependent representation for each sentence based on the overall context. Similarly, source code can also benefit from attention-based feature representa-tion, hence, we employ the recently proposed code2vec model source code repre-sentation [1]. Given a source file SF, code2vec generates its AST and produces a tuple of pairs for each terminal nodes in the file, and a path-context that connects them.

Hence, AttentiveBugLocator reduces the lexical mismatch by compactly learn-ing semantic and contextual information with careful fusion of five IR features: VSM score [37], stack trace score [28], and code complexity score [21] to improve bug localization. Hence, our main contributions are as follows:

1. We propose a suitable representation for bug reports and source files in order to extract richer context to compute a matching score between bug reports and source files.
2. We propose the AttentiveBugLocator model utilizing a Siamese BiLSTM with contextual attention to learn semantic features of bug reports and source files.

3. We apply IR methods to compute textual similarity, stack trace, and code complexity similarity scores in order to improve bug localization perfor-mance.
4. We design a feature fusion technique using FCN to produce vectorized rep-resentations for training our model.
5. We evaluate the performance of our model relative to the state-of-the-art models on four data sets; and ablation study to investigate the effect of attention mechanism and IR features on our model.

THE PROPOSED MODEL

In recent years representation learning has caught the attention of deep learn-ing research community to solve various problems in NLP, image processing, and generative learning [25,35,10]. Representation learning ensures design of ap-proaches suitable for any particular problem. For bug localization, recent studies of [38,14] propose representational methods to uniquely extract features of bug reports and source files.

In this paper, we propose AttentiveBugLocator to employ suitable encoding for extraction of semantic features of bug reports and source files. To learn the semantic matching features, our proposed AttentiveBugLocator uses a BiLSTM network with attention layer as shown in figure 3. Contextual attention helps AttentiveBugLocator eliminate irrelevant features from noisy bug reports and source codes. We also improve the performance of AttentiveBugLocator by care-fully fusing three IR features: rVSM, code complexity, and stack trace scores.

Bug Report Encoding

To prepare bug reports for feature extraction in our proposed model, we apply pre-processing techniques of tokenization and stemming; and Sentence trans-formers to convert word tokens to numerical vectors. Figure 1 shows the process of encoding in AttentiveBugLocator.

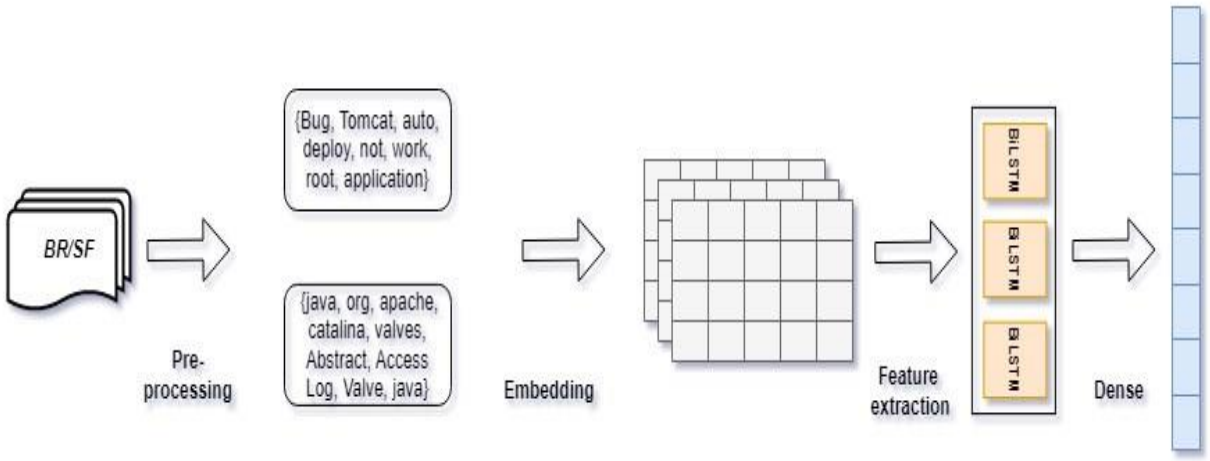


Fig.1: Proposed encoding of bug reports and source files

- Preprocessing: We utilize the NLTK library to apply preprocessing meth-ods of tokenization, stop-word removal, word inflection, and stemming to BR content and SF

source code. The results of this process are a sequence of k words {w1, w2, w3 ,..,wk} for each bug report.

- Word embedding: We initialize each word using pre-trained BERT [5], which produces a dynamic context-dependent representation for each sen-tence based on the overall context. Hence, bug report BR is represented as a word embedding vectors {v1, v2, v3..., vi} across the embedding layer, where vi is the word embedding of vi. The suc-ceeding bug report encoder receives the word embedding sequence after that.

The word vector sequence $\{v1, v2, v3, \ldots vi\}$ of bug report BR is represented by the hidden state of the last layer following the multi-layer bidirectional Transformer blocks.

We use the Bertbase model with the hidden layer size of 768, 12 Transformer blocks, 12 self-attention heads, and a total of 110M parameters.

Source Code Encoding

We apply code embedding - a version of word embedding technique especially for source code.

- Source code embedding: Given a source file SF, code2vec [1] generates its AST and produces a tuple of pairs for each terminal nodes in the file, and a path that connects them. It then uses an attention module to learn path-contexts for each pair of terminals connected by a path. Let the path-context be {ni, pm, nj}, where ni, nj ∈ N, are the values of the terminals and pm ∈ P is the path that connects them. Path-contexts are all looked up and assigned to the appropriate embeddings.

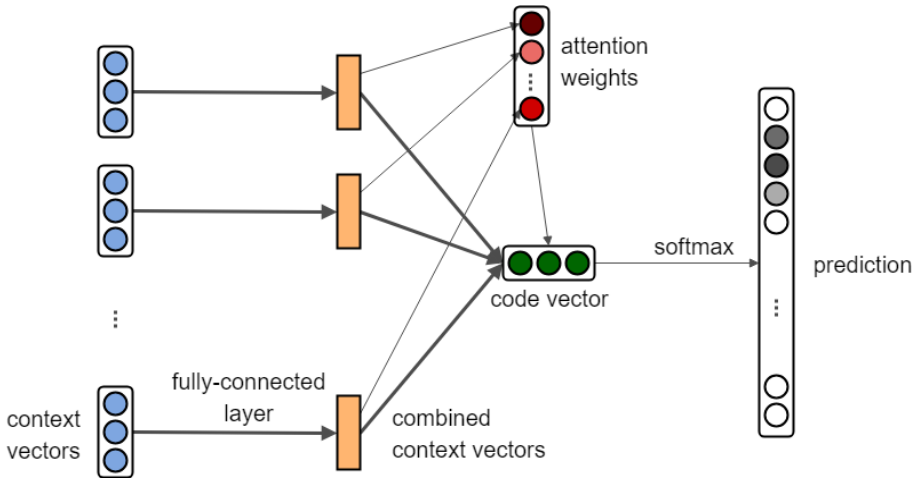


Fig.2: Learning source code representations with code2vec [1]

Semantic Feature Extraction

BiLSTM is a special type of RNN which reads input data in forward and back-ward directions by utilizing LSTM units [20]. Hence, training the BiLSTM model requires three main operations - Forward Pass, Backward Pass and Weights up-date. Figure 3 shows the representation of our model.

Given input embedding vectors for bug report and source file, the BiL- STM produces hidden state for both passes. Let X = {h^a_1, ha_2,...,h^i_1 } and Y = {h^b_1,h^b_2,...,h^j_1} be the concatenated hidden state vector of the forward and backward passes for bug report and source code, respectively. We apply attention mechanism [32] to compute a weighted sum of the hidden states for in-put sequence (x1, x2, x3,...,xn) ∈ X and (x1,x2,x3,...,xm) ∈ Y , respectively. Where m and n are the number of words for bug report and source file.

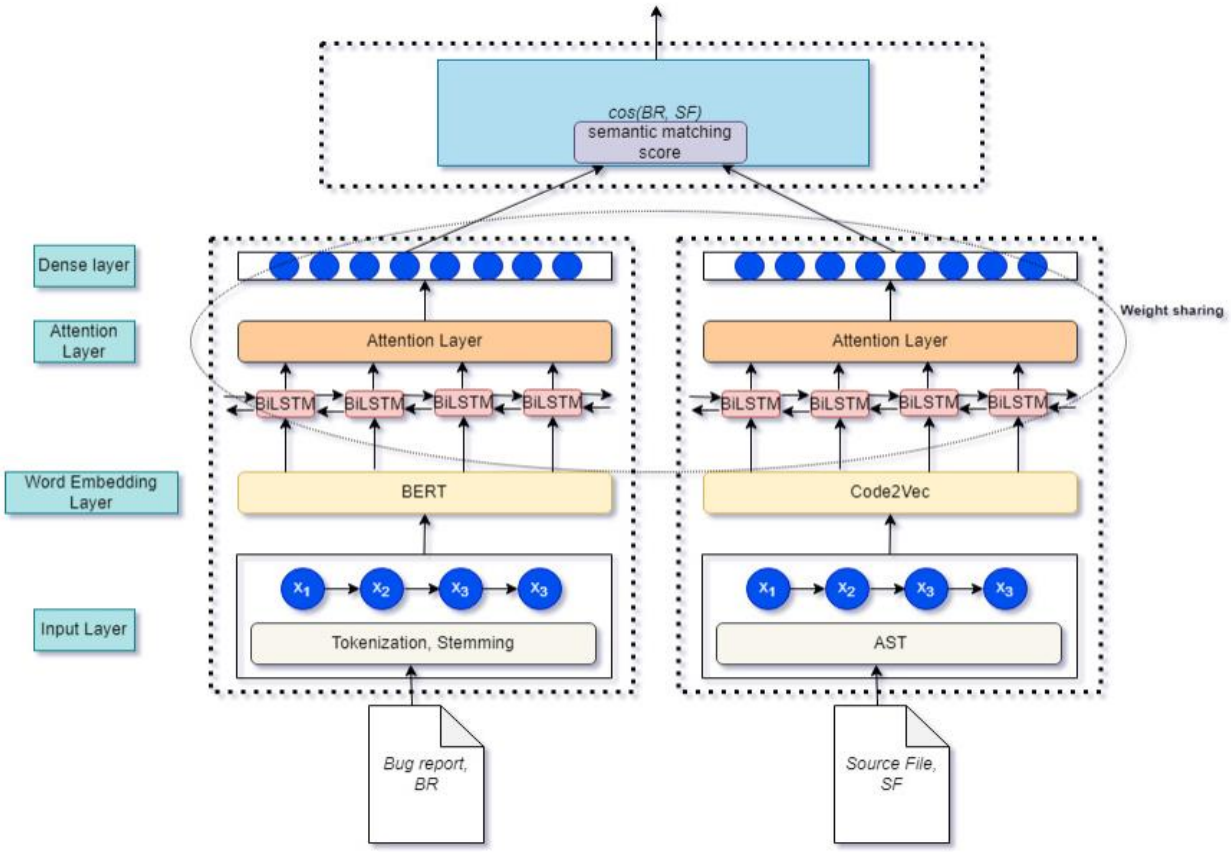


Fig.3: Feature extraction with Attention-based Siamese BiLSTM

Semantic Matching

The outputs of Siamese BiLSTM feature extractor of our model are weight vec-tors representing the attention semantic features of bug reports and source files. To calculate the matching scores, we apply cosine similarity measure:

$$\cos(BR, SF) = \frac{V_b r.V_s f}{|V_b r| \times |V_s f|} \quad (1)$$

Feature Combination with FCN

We employ a fully-connected network (FCN) to combine the resulting matching scores. Hence, each bug report is represented as 4XN feature matrix containing

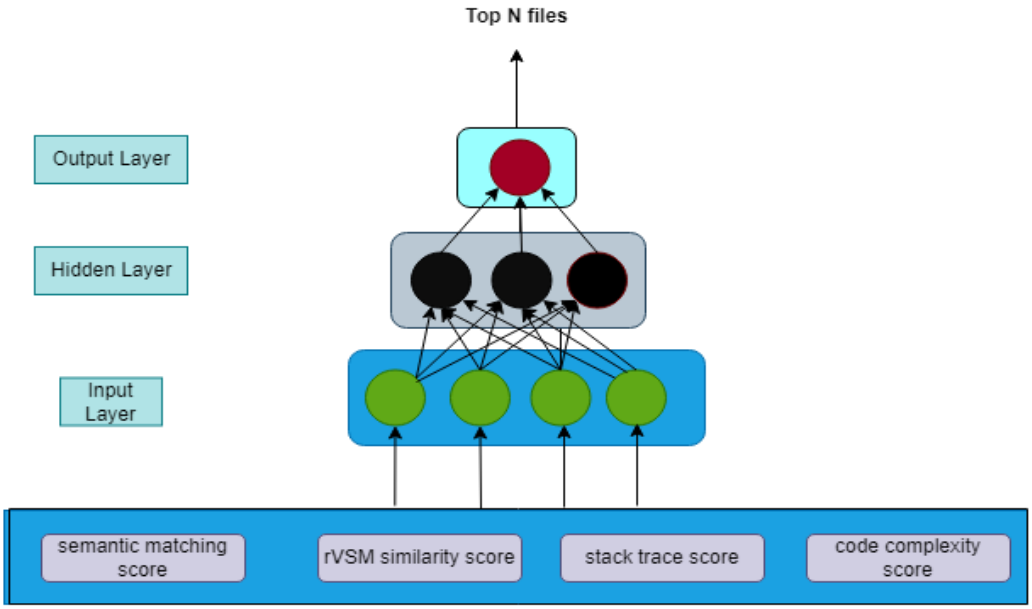


Fig.4: Feature combination with FCN

the attention-based semantic matching score, text similarity, stack trace, and code complexity scores. These feature vectors are used to train the FCN to rank the buggy files. Figure 4 shows the feature combination procedure applied in AttentiveBugLocator.

Figure 5 shows an example of a feature matrix for Eclipse bug report #333274: this bug report has three buggy source files associated with it - JointpointImpl.java, AroundClosure.java, ProceedJointPoint.java. Our model indicates the scores of the four features we utilize: semantic match score (S_m), text similarity (VSM), stack trace score (S_T), and code complexity score (CC). Algorithm 1 shows the bug localization steps of AttentiveBugLocator.

RESULTS AND ANALYSIS

In this section we highlight the data sets, implementation details, and evaluation metrics we adopted to evaluate our model; we also present the research questions proposed and results & analysis.

Data Sets

To evaluate the performance of our proposed model we use the data sets collected by [33] from bugzilla in 2014.

Algorithm 1 Bug Localization
Input: Bug reports BRs, and related source files SFs Output: Ranking of source files, SFs
1: procedure Preprocess
2: for BR,SF ∈ D do
3: tokenize()
4: getStem()
5: embed()

6: end for
7: return v1,v2,...vn
8: end procedure
9: procedure ComputeSemantic
10: for BR,SF ∈ D do
11: EmploySiamese()
12: AddBiLSTM()
13: Encode()
14: si = cos(BR,SF) = $\frac{V_br.V_sf}{\lvert V_br\rvert \times \lvert V_sf\rvert}$
15: end for
16: return s1,s2,...sn
17: end procedure
18: procedure ComputeIRScores
19: for BR,SF ∈ D do
20: V SM = cos(BR,SF) × LenScore_sf(N)
21: ST = 1/rank_s ifst ∈ ST,otherwise,0
22: CC = 1, if (CC) ≥ 40, CC = 0, otherwise
23: end for
24: return v1,v2,...vn 25: end procedure
26: Concat(sn,vn)
27: Normaliza(x...n)
28: Train (x...n)
29: Predict (x...n)

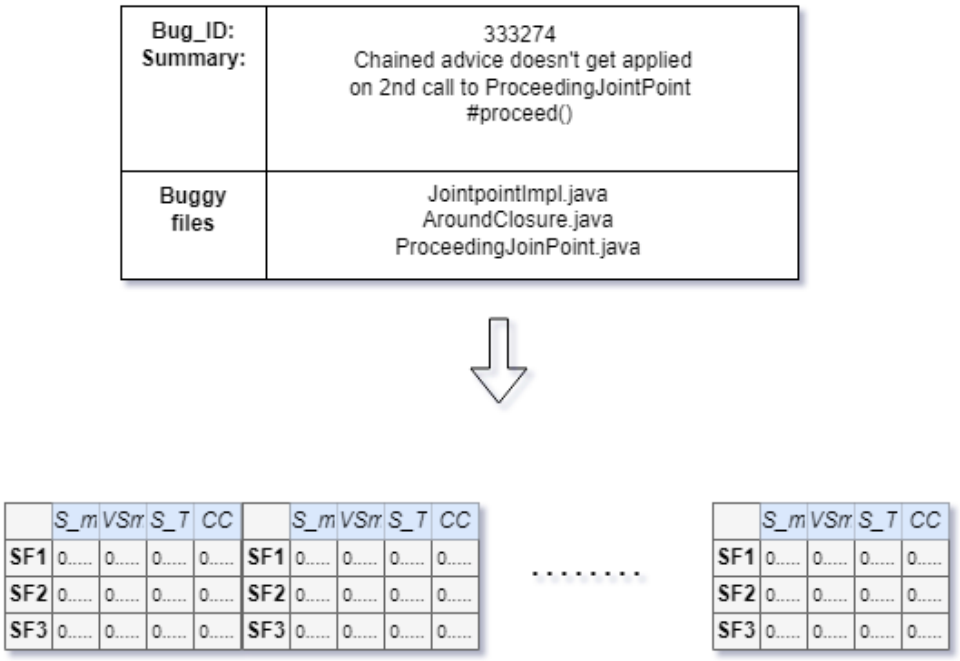


Fig.5: Example of feature matrix representation for Eclipse bug report #333274

Parameter Setting

We implement our models using Tensoflow and Keras libraries on a Windows computer with a 4GB RAM, 64-bit OS, AMD A4-5150M, and 2.70 GHz processor. We split the data sets into 70:30 ratio for training and testing of our model.

As highlighted earlier, we utilise a single-layer BiLSTM network with 256 neurons. Then, we employ a custom attention layer based on the proposal in [32]. We also apply Lambda layer to compute cosine distance between input vec-tors Dropout to reduce overfitting and Batch Normalization to achieve model generalization and maximum accuracy. We then employ Dense layers activated with ReLU and Sigmoid activation functions with ADAM as optimizer and cus-tom cosine loss function.

Evaluation Metrics

We adopted Top-k, mean average precision (MAP) and mean reciprocal rank (MRR) largely used by previous bug localization studies [38]:

Top-k accuracy indicates the percentage of buggy source files that appear in top-k list.

MAP shows the average position of a ranked source file for a given bug report:

$$MAP = \frac{1}{n} \sum_{k=1}^{k=n} AP_i \quad (2)$$

Where n and APi indicates the number of bugs reports and average precision, respectively. For ith bug report:

$$AP_i = \frac{1}{f_n} \sum_{j=1}^{n} Precision\ (n_i, j) \quad (3)$$

MRR is the reciprocal rank of returned buggy files with respect to the highest ranked file for a given bug report:

$$MRR = \frac{1}{n} \sum_{j=1}^{n} \frac{1}{rank_i}) \quad (4)$$

Where n refers to the number of bug reports, while ranki is the reciprocal of the rank of the first located file.

Research Questions

We propose the following research questions to evaluate the performance of our proposed model:

1. How does AttentiveBugLocator perform relative to its variants?
2. How does AttentiveBugLocator perform compared to state-of-the-art ap-proaches?
3. Does Attention mechanism improve bug localization performance?

RQ1: Model Performance

Figure 6 shows the performance of AttentiveBugLocator model and its variants. For top-k performance, AttentiveBugLocator achieves the performance ranging from 54% to 87% on AspectJ project; for JDT, it achieves slightly higher scores between 67% and 87%; and 57% to 91% on SWT and 66% and 80% on Tomcat. Similarly, AttentiveBugLocator shows higher MAP and MRR scores: the lowest and highest MAP scores are 43% and 56% achieved on AspectJ and Tomcat, respectively; while the MRR scores are between 48% and 62%. Overall, the results indicate that AttentiveBugLocator is effective in locating buggy source files. In comparison to derivative models, we observe several performance varia-tions relative to the evaluation measurements. For AspectJ, AttentiveBugLoca-tor surpasses all the derivative models on all metrics except MAP and MRR. The model Siam + Attn + CC shows better scores with percent increase of 8.3% and 2.8% on MAP and MRR, respectively. Siam + Attn + CC is to be the best competitor to AttentiveBugLocator on the AspectJ data set. Further analysis shows much superior performance of our approach on SWT, JDT, and Tomcat. However, the model Siam + Attn achieves slightly better top-k@1. Data set wise, AttentiveBugLocator achieves the best results in the following way: MAP, MRR and top-k@1 performance on Tomcat; top-k@5 and top-k@10 on AspectJ; and top-k@15 on SWT project.

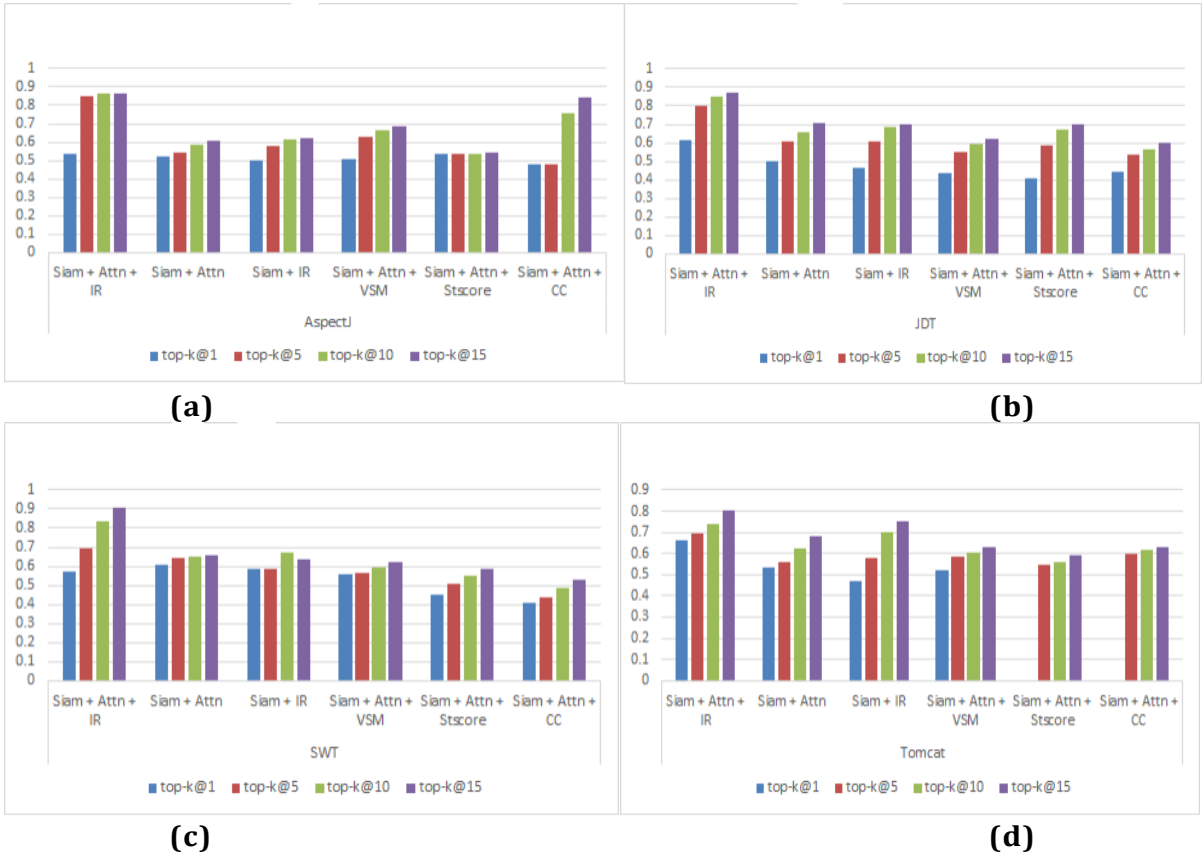


Fig. 6: Evaluation results of top-k, MAP, and MRR on the four data sets (a) AspectJ, (b) JDT, (c) SWT and (d) Tomcat

RQ2: Comparison with State-Of-The-Art Approaches

We compare the performance of our approach with the following models:

1. BugLocator [37]: This classic model improved VSM model (to a new model called revised VSM) to accurately rank suspicious buggy files. BugLocaa-tor ranks buggy files by computing textual similarity between bug reports and source files. Even though, later models have indicated several flaws to textual-similarity-only bug localization models, BugLocator remains highly cited and employed to be compared with recently proposed models – includ-ing the acclaimed deep learning-based models.
2. CAST [14]: This model utilized abstract tree and CNN to represent source files. The model also differentiated user-defined methods and system-defined methods to indicate their contributions to the resulting bug. Then source codes are converted to ASTs, and used Tree-based CNN to extract syntactic ASTs. For its customization of ASTs function, redundant features from the AST outputs are eliminated to enhance model performance.
3. DNNLOC [13]: This model combined various IR features using DNN to re-duce the effect of semantic gap. DNNLOC presents much similarity with our proposed model in the utilization of IR scores and computation of rele-vancy score using FCN. Other features utilized by DNNLOC are bug fixing recency and frequency. Furthermore, the authors proposed auto-encoder toapply dimensionality reduction to increase model training and deployment eficiency.
4. DeepLocator [31]: This model enhanced CNN to capture semantic informa-tion from multiple features of bug reports. Similar to DNNLOC, DeepLocator utilized the textual features of bug reports and bug fixing recency and fre-quency of source files. For word embedding, DeepLocator used both Sent2Vec and Word2Vec to produce feature vectors for training CNN to capture se-mantic information between bug reports and source files. Further evalua-tion shows the relative effectiveness of pre-trained Word2Vec over initialized model.
5. NPCNN [9]: This model utilized multiple CNNs to extract unified features of source files and bug reports. The authors considered the structural nature of source, hence, devised specialized convolution and pooling layers to capture statements and their connections with each other. Previous models suffer from loss of information such as program structure information. Thus [9], proposed CNNs to extract inter-relational features – both structural and lexical – from bug reports and source files. Like DeepLocator [31], design of specific convolution operations is proposed to achieve this objective. The second stage of this model is the cross-language feature fusion operation to combine features of bug reports and source files into a unified representation.

Table 1 shows the comparative results achieved by the state-of-the-art ap-proaches. Overall, AttentiveBugLocator indicates superior performance; how-ever, CAST achieves closely similar performance. While BugLocator attains the lowest performance; for example, in terms of top-k@1, our approach surpasses BugLocator by 43% on JDT. Generally, this implies that DL models are superior to traditional IR systems in locating buggy files.

Similarly, as we mention earlier that the three models, DNNLOC, DeepLoca-tor, and NPCNN proposed methods to combine multiple features of bug reports and source files to improve bug localization performance. However, AttentiveBu-gLocator utilizes a Siamese BiLSTM network

with context attention to perform similar task. Furthermore, similar to CAST we employ AST to extract code to-kens - class and method names. Hence, the promising performance of our AttentiveBugLocator proves that combining methods to improve extraction of features from both bug reports and source files can enhance bug localization.

Similarly, our experiments imply that, high performance bug localization depends largely on bug reports and source files encoding & representation.

Comprehensive comparative results between BugLocator and our Attentive-BugLocator show the following trends: For AspectJ, our top model shows high performance improvements of 32%, 37.6%, 29.5%, and 22.3% on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. For MAP and MRR, it shows 15.5% and 11.1% improvements, respectively. For SWT it achieves enhancements of 32.3%, 28.4%, 30.5%, and 31.1% on top-k@1, top-k@5, top-k@10, and top-k@15, respec-tively. Similarly, MAP and MRR evaluations show increase of 20.3% and 18.9%, respectively. For JDT, AttentiveBugLocator achieves 43%, 37.3%, 34.1%, 30.8% on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. On MAP and MRR, it attains 15% and 12.3%, respectively. Finally, for Tomcat data set, BugLocator shows competitive performance; it trails AttentiveBugLocator by 30.6%, 5.2%, 3.1%, 4.9%, 12.7% and 13.5% on top-k@1, top-k@5, top-k@10, top-k@15, MAP and MRR, respectively.

As indicated from the evaluation results, the improvements over BugLoca-tor are within the range of 3.1% - 43%. This indicates that deep learning-based models combining both textual similarity and semantic information can achieve superior performance. Although, AttentiveBugLocator attains relatively higher scores than BugLocator on all metrics, the best improvements come from top-k@1 for all data sets. However, the performance boosts on Tomcat data set are lower than on every other three data sets. Overall, we notice that, Attentive-BugLocator achieves its lowest performance on this data set. In the future, we intend to investigate this trend. We also find that the biggest trade-off is that our model can exhibit higher complexity than BugLocator. However, we simi-larly notice that other variants of our model, with less complexity – for example, Siam + IR and Siam + Attn – equally outperforms BugLocator on almost all metrics on the four data sets. Additionally, we study comprehensively the DNNLOC model proposed by [13].

As mentioned previously, DNNLOC utilizes TF-IDF, RBM, and auto-encoder for feature extraction, representation, and ranking of buggy files. To improve previous attempts, DNNLOC extracts features from API documents and bug fixing recency (a feature we do not include in our approach) to improve bug localization. Furthermore, to validate our choice of this model (beside being highly cited by previous studies [31,14,6]) some of the interesting questions we thought are: what can be the effect of dimensionality reduction with AE for bug reports and source files? How can our Siamese-based model perform com-pared to DNNLOC's RBM? Based on comprehensive evaluation we notice wide performance variations between the two models.

Specifically, for AspectJ AttentiveBugLocator achieves 5.8% improvement on top-k@1, 13.7% on top-k@5, 6.2% on top-k@10, and 0.4% on top-k@15. For MAP AttentiveBugLocator shows higher improvements of 11.1% while declines on MRR by 4%. We also notice that DNNLOC

similarly outperforms BugLocator and DeepLocator on this metric. It should also be mentioned that the improve-ments over BugLocator model are negligible: “The corresponding improvement values over rVSM are from 1–5%”. For SWT: 21.7%, 0.2%, 3.5%, 5.3% on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. For MAP and MRR, At-tentiveBugLocator shows 11.7% and 6.4% boosts, respectively. This indicates wide gap on top-k@1 and MAP. However, DNNLOC still shows competitive results compared to NPCNN, DeepLocator, and BugLocator.

Overall, the per-formance of the models show lower achievements on this data set relative to AspectJ, JDT, and Tomcat. For JDT, AttentiveBugLocator shows the follow-ing enhancements over DNNLOC: 21%, 15%, 10.6%, and 7.6% on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. For MAP and MRR, AttentiveBugLoca-tor shows 11.7% and 6.2% respectively. On average, the improvements of our approach over DNNLOC are highest on this data set. For Tomcat, DNNLOCoutperforms our model on top-k@5, top-k@10 by 3.2%, 6.4%, 5.8% on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. However, for top-k@1, MAP and MRR, AttentiveBugLocator achieves 12.1%, 3.8%, and 2%, respectively. Even though, DNNLOC attains higher performance than AttentiveBugLocator on the three metrics, our model still achieves promising performance relative to other comparative models. Further, in the future, we intend to conduct further analysis of AttentiveBugLocator’s performance on the Tomcat data set. Overall, the performance of AttentiveBugLocator with respect to DNNLOC indicates that combining multiple features is effective - specifically, for DNNLOC, bug fixing recency and API features - can improve bug localization. However, [13] report that “DNN by itself does not give high accuracy.” Similarly, for DNNLOC to be effective, the number of hidden nodes is set to be between 1000 and 1100 "for a satisfactory accuracy." For this, it is imperative to investigate further the utilization of DNNs for improved bug localization. Thus, we also suppose our utilization of Siamese and a feature combinator network may be a preferred approach to achieve high performance and reduced model complexity. We also compare the performance of AttentiveBugLocator with DeepLoca-tor [31]. DeepLocator proposes r-TF-IDuF (revised term frequency user-focused inverse document frequency) for bug reports processing and weighting based on search upon local collections [2]. DeepLocator uses bug fixing recency and fre-quency information to improve extraction of semantic information. Additionally, DeepLocator utilizes AST to process source code and word2Vec to process bug reports. Comparisons with SVM, Naïve Bayes, CNN, and HyLoc [12] shows the effectiveness of this approach with improvement of 36.29%.

Comparative results with DeepLocator shows the following trends: For As-pectJ: 13.6%, 18.9%, 8.6%, and 9.1% on top-k@1, top-k@5, top-k@10, and MAP, respectively. In contrast, DeepLocator achieves 1% improvement over our model on MRR. For SWT: 20.9%, 9.2%, 5.8%, on top-k@1, top-k@5, and top-k@10, respectively. For MAP and MRR, AttentiveBugLocator shows 9.7% and 3.4% improvements, respectively. Similarly, for JDT: 21.3%, 16%, and 11.9%, on top-k@1, top-k@5, and top-k@10, respectively. For MAP and MRR, AttentiveBugLocator shows 6.7% and 4.2%, respectively. Unsurprisingly for Tomcat data set (as we no-tice with other comparative models) AttentiveBugLocator falls behind DeepLo-cator by 2.3%, 6%, on top-k@5 and top-k@10. However, for top-k@10, MAP and MRR, AttentiveBugLocator shows 12.1%, 1.8% and 2%, respectively. From the evaluations, we can see that DeepLocator attains

competitive performance; in comparison to derivative models of our model (Siam + IR, Siam + Attn, Siam+Attn+V SM, Siam+Attn+Stscore, and Siam+Attn+CC), DeepLo-cator shows superior performance on most measurements. Thus, we can infer that careful utilization of r-TF-IDuF, AST, and CNN can improve bug localization. In contrast, AttentiveBugLocator is a more suitable approach for bug reports, source files encoding and semantic features extraction.

Furthermore, we perform comparative analysis of AttentiveBugLocator rela-tive to NP-CNN [9]. Previous models [12,37] ignore information such as program structure information. Thus, [9] proposed CNNs to extract inter-relational features – both structural and lexical – from bug reports and source files. Like DeepLocator [31], in this model specific convolution operations are proposed to achieve this objective. The second stage of this model is the cross-language fea-ture fusion operation to combine features of bug reports and source files into a unified representation.

Table 1: Performance comparison with state-of-the-art approaches

Project	Method	top-k@1	top-k@5	top-k@10	top-k@15	MAP	MRR
AspectJ	BugLocator	0.216	0.473	0.571	0.643	0.276	0.369
	DNNLOC	0.478	0.712	0.804	0.862	0.32	0.52
	DeepLocator	0.4	0.66	0.78	-	0.34	0.49
	NPCNN	0.46	0.73	0.81	0.852	0.401	0.531
	CAST	0.500	0.766	0.830	0.870	0.418	0.536
	AttentiveBugLocator	0.536	0.849	0.866	0.866	0.431	0.48
SWT	BugLocator	0.246	0.408	0.533	0.595	0.284	0.325
	DNNLOC	0.352	0.69	0.803	0.853	0.37	0.45
	DeepLocator	0.36	0.6	0.75	-	0.39	0.48
	NPCNN	0.365	0.7	0.812	0.865	0.391	0.475
	CAST	0.372	0.701	0.825	0.873	0.425	0.503
	AttentiveBugLocator	0.569	0.692	0.838	0.906	0.487	0.514
JDT	BugLocator	0.183	0.427	0.508	0.562	0.307	0.389
	DNNLOC	0.403	0.65	0.743	0.794	0.34	0.45
	DeepLocator	0.4	0.64	0.73	-	0.39	0.47
	NPCNN	420	0.669	0.786	0.804	0.386	0.461
	CAST	0.432	0.681	0.757	0.818	0.388	0.467
	AttentiveBugLocator	0.613	0.800	0.849	0.87	0.457	0.512
Tomcat	BugLocator	0.354	0.645	0.709	0.752	0.431	0.485
	DNNLOC	0.539	0.729	0.804	0.859	0.52	0.6
	DeepLocator	0.52	0.72	0.8	-	0.54	0.6
	NPCNN	0.53	0.7	0.792	0.853	0.529	0.597
	CAST	0.507	0.766	0.825	0.869	0.556	0.612
	AttentiveBugLocator	0.660	0.697	0.740	0.801	0.558	0.62

Comparative results with NPCNN shows that our proposed AttentiveBugLo-cator achieves superior performance on all data sets except on Tomcat. Specif-ically, for AspectJ, 7.6%, 11.9%, 5.6%, 1.4% higher on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. For MAP and MRR, AttentiveBugLocator shows 3% lags NPCNN by 5.1%. For SWT: 20.4%, 2.6%, 4.1%, 9.6% and 3.9% on top-k@1, top-k@5, top-k@10, and top-k@15, MAP and MRR, respectively. However, NP-CNN outperforms AttentiveBugLocator by minute score of 0.8% on top-k@5. For JDT, our model outperforms NPCNN on all metrics by 19.3%, 13.1%, 6.3%, 6.6% on top-k@1, top-k@5, top-k@10, and top-k@15, respectively. Similarly, for MAP and MRR, AttentiveBugLocator shows 7.1% and 5.1%, re-spectively. Finally, for Tomcat AttentiveBugLocator achieves higher scores of 13%, 2.9% and 2.3%, on top-k@1, MAP and MRR, respectively. In contrast, NP-CNN outperforms AttentiveBugLocator on top-k@5, top-k@10, and top-k@15 measurements by 0.3%, 5.2%, and 5.2%, respectively. Specifically, NP-CNN improves bug localization performance by extracting structural information of source code using special convolution operations. Another recently proposed model is CAST [14] utilizing AST and tree-based CNN (TBCNN) [18] to represent structural features of source code. Previously, many studies employ AST to perform similar task [31,9]; however, CAST elim-inates redundant methods from AST sub-trees to improve model training. Fur-ther, CAST identifies user-defined and system-designed methods, essentially to compute each method type's contributions. Evaluation results of AttentiveBugLocator relative to CAST on the four data sets shows the following trends: For AspectJ: 3.6%, 8.3%, 3.6%, and 1.3% on top-k@1, top-k@5, top-k@10, and MAP, respectively. While for top-k@15 and MRR, AttentiveBugLocator lags CAST by 0.4% and 5.6%. For SWT, AttentiveBugLocator similarly shows superior performance except for top-k@5 - where it falls behind CAST by minute 0.9%. AttentiveBugLocator achieves improvements of 19.7%, 1.3%, 3.3%, 6.2% and 1.1% on top-k@1, top-k@10, top-k@15, MAP and MRR, respectively. For JDT: 18.1%, 11.9%, 9.2%, 5.2%, 6.9% and 4.5% on top-k@1, top-k@5, top-k@10, top-k@15, MAP and MRR, respec-tively. For Tomcat AttentiveBugLocator shows higher performance by 15.3%, 0.2% and 0.8% on top-k@1, MAP and MRR, respectively. While CAST achieves higher performance on top-k@5, top-k@10, and top-k@15 measurements by 6.9%, 8.5%, 6.8%, respectively. However, the performance boost of AttentiveBugLocator over CAST is neg-ligible on most measurements. We also notice that CAST outperforms BugLoca-tor, DeepLocator, DNNLOC, and NP-CNN on all data sets except JDT. Thus, it can be implied that pruning of AST nodes enhances model training and localiza-tion performance. In contrast, to achieve similar objective, we utilize attention-based code processing using code2vec [1].

RQ2: Effect of Attention Mechanism

We perform an ablation study to evaluate the effectiveness of attention mecha-nism on AttentiveBugLocator. We implemented Siam + IR model without atten-tion to compare with our AttentiveBugLocator. Siam + IR utilizes the Siamese feature extractor and an IR module of VSM, stack trace and code complexity scores. As shown in Table 2, AttentiveBugLocator surpasses Siam + IR in all mea-surements except top-k@1 with negative margin of 1.4%. We attribute the posi-tive effect of attention on AttentiveBugLocator to the following: source files and bug reports contain noise and irrelevant methods for any particular bug report [28]; hence, applying attention provides extraction of contextual features [32]. This correlates with our earlier hypothesis that attention can help extract im-portant features of bug reports and source

files. Hence, we show that, a custom attention layer can achieve high performance, hence, reducing the need for a very complex bug localization model.

Table 2: Effect of attention mechanism

Project	Method	top-k@1	top-k@5	top-k@10	top-k@15	MAP	MRR
JDT	BugTranslator	-	0.554	-	-	0.34	0.41
	DEMoB	0.547	0.749	0.824	-	0.498	-
	Siam + IR	0.465	0.604	0.683	0.704	0.427	0.433
	AttentiveBugLocator	0.613	0.800	0.849	0.870	0.457	0.512
SWT	BugTranslator	-	0.562	-	-	0.34	0.4
	DEMoB	0.596	0.832	0.887	-	0.561	-
	Siam + IR	0.583	0.584	0.669	0.638	0.366	0.387
	AttentiveBugLocator	0.569	0.692	0.838	0.906	0.487	0.514

Furthermore, we compared the performance of our approach with recently proposed attention-based bug localization models: BugTranslator [30] and DE-MoB [38]. These models employed self-attention mechanism to improve seman-tic feature extraction for bug localization. However, in AttentiveBugLocator we employ an attention layer on top of both bug reports and source files feature extractors, hence our utilization of Siamese BiLSTM network [32]. Table 2 shows comparative results of the attention-based models on JDT and SWT data sets. AttentiveBugLocator surpasses both BugTranslator and DEMoB on JDT; while DEMoB achieves better results on SWT. Specifically, our approach lags DEMoB by 2.7%, 14%, 4.9%, and 7.4% on top-k@1, top-k@5, top-k@10, and MAP re-spectively. However, DEMoB may incur higher model complexity. For example, unlike our approach the source code encoder in DEMoB uses multiple layers of Dynamic CNN (DCNN) and BiLSTMs. We also note that the authors of BugTranslator and DEMoB failed to utilize IR features employed by AttentiveBugLocator; instead, both models rely solely on bug report textual contents, source files, and API documents.

Other Works - Graph-Based Models, CNN-LSTM

In this section we analyze the performance of non-IR-based models - such as graphical solutions and CNN-LSTM - for bug localization. Most notably Graph CNN or GCN - a CNN that works on graphs has been widely utilized as gener-alized version of CNN for prediction tasks [23]. BugPre [27] - a GCN-based model is proposed to detect code & version changes, dependencies, and change-sets to predict software defects. Similar to BugPre, we utilize ASTs to extract structural features, but these models are purely graph-based - it implies that "GCN model enables to learn deeper in-formation about the code." However, results are limited to a small amount of data set and little comparison with state of the art – thus, yet to establish the significance of graphs for bug localization. [4] proposed BugPecker to utilize re-vision graphs to extract relationships from methods with little information. [36] proposed Knowledge Graph and attention mechanism to perform closely similar task. Table 3 shows the comparative results achievd by AttentiveBugLocator rel-ative to BugPecker and KGBLocator. It can be observed that AttentiveBugLo-cator achieves the best MRR and lags negligibly on the Top-k@10 and MAP measurements.

Table 3: Comparison with graph-based models

Project	Method	top-k@10	MAP	MRR
Tomcat	BugPecker	0.158	0.121	0.143
	CG-CNN	-	0.51	0.56
	KGBugLocator	0.828	0.56	0.619
	AttentiveBugLocator	0.74	0.558	0.62

Finally, CNN-LSTM has shown promising results for extracting semantic and structural information [8]. In the future, we aim to further investigate this model in comparison to our BiLSTM Siamese-based model.

DISCUSSION

Why does AttentiveBugLocator work? We identify three challenges with current bug localization systems:

1. Finding suitable representation that reduce the lexical mismatch between bug reports and source files. For this, many bug localization models utilize textual similarity and ranking algorithms. However, recently, researcher fo-cus on designing models that treat bug reports and source files uniquely to leverage the unique features of both entities [14].
2. Researches [28,29] similarly focus on reducing noise in bug report files to improve bug report processing. The main task here is to extract keywords that can aid in effective bug localization.
3. How to effectively represent bug reports and source files in order to achieve rich representations that can eficiently match each other in semantic vector space [8,24].

Our model derives effectiveness by utilizing specific encoding for bug report and source files and a Siamese BiLSTM model with attention to extract rich se-mantic features. Our motivation for choosing Siamese-based approach is model eficiency through weight sharing, imbalance learning, and flexibility of loss func-tion selection. Hence, Siamese-based models have been widely shown to achieve superior performance for semantic similarity tasks [3]. We conducted preliminary experiments to validate our choice of BERT for this research. Evaluations on Eclipse data sets indicate higher performance rel-ative to Word2Vec model. However, we noted variable performance when we utilized the IR components. Unlike Word2Vec, BERT produces dynamically con-textual word vectors for bug report tokens. For this, we assert that the BERT model is a better approach for bug localization. Theoretically, our choice of code2vec for source code representation is in-spired by the recent performance analysis in method name prediction, code se-mantic extraction [34]. However, we cannot be sure whether the heavy compu-tation and the cost of employing code2vec are worthy trade-offs. Similarly, this model could behave quiet differently with different components of AttentiveBu-gLocator. We aim to investigate these areas in our future research. Finally, a thorough overview of bug localization models [11,22] shows the ef-fects of increasing components on bug localization performance. Our model cor-relates well with this hypothesis: AttentiveBugLocator (Siam+Attn+IR) shows the best performance on all data sets. Similarly, the derived models with indi-vidual components Siam+Attn+VSM and Siam+Attn+CC attain higher perfor-mance relative to

Siam+Attn. However, the model Siam+Attn+Stscore achieves relatively fewer promising results on the AspectJ and SWT data sets. Essentially, the presence/absence of stack trace in large number of bug reports affect model performance [17].

Threat to Validity

In this section, we describe the threats to the correctness of our experiments for replication and further analysis by the research community. Internal Validity: To produce suitable representations for bug reports and source files, we use several pre-procssing procedures such as tokenization, stem-ming, and word embedding; hence, the performance of AttentiveBugLocator is largely dependent on the built-in design of these models. External Validity: Because we evaluate AttentiveBugLocator on open-source data sets, we cannot ensure the correctness of bug reports with any associated source files. We choose these data sets based on their availability and reports from past studies [9]. Construct Validity: Data imbalance is another issue yet to be addressed in bug localization [14]. For this, data splitting strategies can have disparate effects on localization performance. For future studies, we intend to deploy multiple data splitting strategies to conduct interpretative analysis of results based on major evaluation metrics utilized in bug localization studies.

RELATED WORKS

Several techniques have been proposed in the past to locate buggy source files with respect to user bug reports based on IR, topic modeling, and machine learning [11]. [15] examine the accuracy and scalability of LDA-based method for bug lo-calization on search space of over 300 bugs. Other features such as source code structure, document length and bug fixing history are proposed. [37] proposed BugLocator based on revised VSM (rVSM) to rank source files by extracting weights for words in the bug reports, and fix history to improve bug localiza-tion. [19] proposed Bug Localization using IR (BLUiR) based on structured IR approach which considers code's method, class, and variable names and com-ments similarity to localize bugs. To further improve the effectiveness of bug localization models, recent studies proposed stack trace analysis, POS tagging and code segmentation to improve query quality [11]. Furthermore, traditional ML classifiers have been utilized to apply topic mod-eling on bug reports and source files [26]. However, to improve extraction of both semantic and textual features of both bug reports and source files, DL-based bug localization approaches have been proposed based on CNN, RNN, and At-tention mechanism [38]. For this, many proposed approaches employed feature fusion technique - by using a pair of networks such as CNNs to compute semantic correlations between bug reports and source files [9]. However, the lexical mismatch remains a key issue in the effectiveness of these methods for bug localization. Very recent studies show that suitable representa-tions of bug reports and source files using both deep learning and IR methods can improve the effectiveness of bug localization [16,7].

CONCLUSION

In this paper we proposed AttentiveBugLocator, a representational learning ap-proach utilizing unique ecodings for bug reports and source files using BERT transformer and code2vec models to obtain embedding vectors; and a Siamese BiLSTM feature extractor network with contextual attention mechanism to learn richer semantic features for accurate bug localization. Analysis results showed that AttentiveBugLocator outperforms several state-of-the-art approaches on

four open-source data sets. In the future, we will perform wide analysis of At-tentiveBugLocator on cross-project data sets.

References

1. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3(POPL), 1–29 (2019)'

2. Beel, J., Langer, S., Gipp, B.: Tf-iduf: a novel term-weighting scheme for user modeling based on users' personal document collections (2017)

3. Bölücü, N., Can, B., Artuner, H.: A siamese neural network for learning semantically-informed sentence embeddings. Expert Systems with Applications 214, 119103 (2023)

4. Cao, J., Yang, S., Jiang, W., Zeng, H., Shen, B., Zhong, H.: Bugpecker: Locating faulty methods with deep learning on revision graphs. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 1214–1218 (2020)

5. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirec-tional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)

6. Fang, F., Wu, J., Li, Y., Ye, X., Aljedaani, W., Mkaouer, M.W.: On the classifica-tion of bug reports to improve bug localization. Soft Computing 25(11), 7307–7323 (2021)

7. Huang, X., Xiang, C., Li, H., He, P.: Sbuglocater: Bug localization based on deep matching and information retrieval. Mathematical Problems in Engineering 2022 (2022)

8. Huo, X., Li, M., Zhou, Z.H.: Control flow graph embedding based on multi-instance decomposition for bug localization. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 4223–4230 (2020)

9. Huo, X., Li, M., Zhou, Z.H., et al.: Learning unified features from natural and programming languages for locating buggy source code. In: IJCAI. vol. 16, pp. 1606–1612 (2016)

10. Jia, C., Yang, Y., Xia, Y., Chen, Y.T., Parekh, Z., Pham, H., Le, Q., Sung, Y.H., Li, Z., Duerig, T.: Scaling up visual and vision-language representation learning with noisy text supervision. In: International Conference on Machine Learning. pp. 4904–4916. PMLR (2021)

11. Kim, M., Kim, Y., Lee, E.: An empirical study of ir-based bug localization for deep learning-based software. In: 2022 IEEE Conference on Software Testing, Verifica-tion and Validation (ICST). pp. 128–139. IEEE (2022)

12. Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: 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). pp. 476–481. IEEE (2015)

13. Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). pp. 218–229. IEEE (2017)

14. Liang, H., Sun, L., Wang, M., Yang, Y.: Deep learning with customized abstract syntax tree for bug localization. IEEE Access 7, 116309–116320 (2019)

15. Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using latent dirichlet allocation. Information and Software Technology 52(9), 972–990 (2010)

16. Luo, Z., Wang, W., Cen, C.: Improving bug localization with effective contrastive learning representation. IEEE Access (2022)

17. Moreno, L., Treadway, J.J., Marcus, A., Shen, W.: On the use of stack traces to im-prove text retrieval-based bug localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 151–160. IEEE (2014)

18. Mou, L., Li, G., Jin, Z., Zhang, L., Wang, T.: Tbcnn: A tree-based convo-lutional neural network for programming language processing. arXiv preprint arXiv:1409.5718 (2014)

19. Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Confer-ence on Automated Software Engineering (ASE). pp. 345–355. IEEE (2013)

20. Schuster, M., Paliwal, K.K.: Bidirectional recurrent neural networks. IEEE trans-actions on Signal Processing 45(11), 2673–2681 (1997)

21. Seyam, A.A., Hamdy, A., Farhan, M.S.: Code complexity and version history for enhancing hybrid bug localization. IEEE Access 9, 61101–61113 (2021)

22. Shi, Z., Keung, J., Bennin, K.E., Zhang, X.: Comparing learning to rank techniques in hybrid bug localization. Applied Soft Computing 62, 636–648 (2018)

23. Veličković, P.: Everything is connected: Graph neural networks. Current Opinion in Structural Biology 79, 102538 (2023)

24. Wang, B., Xu, L., Yan, M., Liu, C., Liu, L.: Multi-dimension convolutional neural network for bug localization. IEEE Transactions on Services Computing (2020)

25. Wang, H., Wang, J., Wang, J., Zhao, M., Zhang, W., Zhang, F., Xie, X., Guo, M.: Graphgan: Graph representation learning with generative adversarial nets. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)

26. Wang, Y., Yao, Y., Tong, H., Huo, X., Li, M., Xu, F., Lu, J.: Bug localization via supervised topic modeling. In: 2018 IEEE international conference on data mining (ICDM). pp. 607–616. IEEE (2018)

27. Wang, Z., Tong, W., Li, P., Ye, G., Chen, H., Gong, X., Tang, Z.: Bugpre: an intel-ligent software version-to-version bug prediction system using graph convolutional neural networks. Complex & Intelligent Systems pp. 1–21 (2022)

28. Wong, C.P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 181–190. IEEE (2014)

29. Xiao, X., Pan, Y., Zhang, B., Hu, G., Li, Q., Lu, R.: Albfl: A novel neural ranking model for software fault localization via combining static and dynamic features. Information and Software Technology 139, 106653 (2021)

30. Xiao, Y., Keung, J., Bennin, K.E., Mi, Q.: Machine translation-based bug localiza-tion technique for bridging lexical gap. Information and Software Technology 99, 58–61 (2018)

31. Xiao, Y., Keung, J., Mi, Q., Bennin, K.E.: Improving bug localization with an enhanced convolutional neural network. In: 2017 24th Asia-Pacific Software Engi-neering Conference (APSEC). pp. 338–347. IEEE (2017)

32. Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., Hovy, E.: Hierarchical attention networks for document classification. In: Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies. pp. 1480–1489 (2016)

33. Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports us-ing domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 689–699 (2014)

34. Yu, D., Yang, Q., Chen, X., Chen, J., Xu, Y.: Graph-based code semantics learning for eficient semantic code clone detection. Information and Software Technology 156, 107130 (2023)

35. Zhang, C., Yang, Z., He, X., Deng, L.: Multimodal intelligence: Representation learning, information fusion, and applications. IEEE Journal of Selected Topics in Signal Processing 14(3), 478–493 (2020)

36. Zhang, J., Xie, R., Ye, W., Zhang, Y., Zhang, S.: Exploiting code knowledge graph for bug localization via bi-directional attention. In: Proceedings of the 28th Inter-national Conference on Program Comprehension. pp. 219–229 (2020)

37. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? more accurate in-formation retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 14–24. IEEE (2012)

38. Zhu, Z., Li, Y., Wang, Y., Wang, Y., Tong, H.: A deep multimodal model for bug localization. Data Mining and Knowledge Discovery 35(4), 1369–1392 (2021)