SemirFL: Boosting Fault Localization via Combining Semantic Information and Information Retrieval

Xiangyu Shi, Xiaolin Ju*, Xiang Chen*, Guilong Lu, and Mengqi Xu
Nantong University, Nantong, Jiangsu, China
xiangyushi@stmail.ntu.edu.cn, ju.xl@ntu.edu.cn, xchencs@ntu.edu.cn, lululu_dream@163.com, xmq_sss@163.com
*corresponding author

***Abstract*—Automated fault localization aims to reduce software maintenance's workload during software development's evolution. Applying different features extracted from bug reports and source files can help locate faults. However, these approaches consider programming languages as natural when measuring similarity features, considering only precise term matching and ignoring deep semantic similarity features. Furthermore, existing bug localization approaches need to utilize the structural information extracted from source files, where program languages have unique structural features compared to natural languages. In this paper, we proposed SemirFL, a model combining both Convolutional Neural Network(CNN) and revised Vector Space Model(rVSM), which is feeded with four metadata features (bug-fixing recency, bug-fixing frequency, collaborative filtering score, and class name similarity). SemirFL has been studied on four open-source projects. The experimental results show that SemirFL can significantly outperform the existing representative techniques in locating faults in the buggy source files.**

***Keywords*—Fault localization; Deep Learning; Information retrieval; Convolutional neural network**

I. INTRODUCTION

Fault localization plays a vital role in ensuring the quality of software. Existing software fault localization methods can be divided into two categories: the first category is the dynamic program analysis-based fault localization method which focuses on the program's semantics and execution. This category analyzes the program's inner structure through the execution of test cases to determine the possible position of the defective statement in the program [1][2][3][4]. The second category is a static location method based on Information Retrieval(IR), this method treats a bug report as a query, then calculates the similarity between the query and all the source code files, and sorts the source code file according to the calculated similarity from largest to smallest, which places emphasis on the textual content found between bug reports and software modules rather than the semantics of the source code for locating faults.

Existing techniques are only suitable for dealing with simple scenarios where there is a direct text term match between the bug report and the source file [5]. However, most of the bug reports and source files have almost no similarity between them, which leads to the problem of vocabulary mismatch issue [6]. To bridge the lexical gap, Ye et al. proposed Learning-to-Rank, which additionally incorporates features from source files, API descriptions, bug-fixing, and bug repair history, in addition to the content of the bug report [7]. Although Learning-to-Rank has been shown effective in fault localization, the information may not be fully utilized because these information retrieval methods can not automatically extract powerful features [8]. Therefore, we need fault localization methods that can capture deep semantic information that goes beyond exact term matching.

With the emergence of various algorithms in the field of deep learning, many deep learning models have been introduced to fault location for their excellent ability to extract and represent high-dimensional features. However, these existing representation learning algorithms in deep learning cannot fully exploit hierarchies because they treat programming language and natural language equally. Although lexical features are considered, they ignore the structural information behind the programming language. Natural language has a "flat" form, i.e., swapping the order of upper and lower sentences does not change the semantic information of the sentence, whereas programming language organizes its statements in a "structured" way [9], where the function semantics will be changed by the statement order (e.g., if-then-else).

In this study, we propose SemirFL, a model combining CNN, and advanced information retrieval technique including rVSM [10] and integrate another four kinds of features using domain knowledge, called metadata features (bug fixing recency, bug fixing frequency, collaborative filtering score, class name similarity), extracted from bug reports, source codes, and bug-fixing history [7]. We use the rVSM model to measure textual similarity to match bug reports with the associated buggy files. We hope that CNN can automatically mine semantic information common to program language and natural language, which can help compensate for the lack of simple lexical matching. In addition, metadata features are used to further bridge the lexical gap between the content of bug reports and source files.

To evaluate the effectiveness of our proposed method, we also conduct empirical evaluations on four open-source software projects. The experimental results showed that our SemirFL can outperform the existing representative fault localization techniques. Our Empirical study revealed that CNN and rVSM similarity features and metadata features could complement well for fault localization.

The main contributions of our study can be summarized as follow:

- Proposed a fault localization technique called SemirFL, combining CNN and rVSM similarity features and metadata extracted from information retrieval methods.

2693-9371/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS-C57518.2022.00055

- Conducted an empirical study on four widely used projects, and the experiment results indicate that the proposed SemirFL can outperform the other three fault localization approaches.

II. BACKGROUND

A. IR-Based Fault Localization

Information Retrieval (IR) Based Fault Localization is a fault localization technique based on information retrieval to extract information from software documents and program activities [11]. In IR-based fault localization, bug reports are crucial. The bug report includes a text description and non-textual information (source snippets). At the same time, the summary section, which often provides many location cues, briefly explains the abnormal behaviour of the examined software.

Fault location matches source code files expressed in program language with bug reports written in natural language. It can be equated to an Information Retrieval (IR) task where documents (source files) are sorted according to their relevance to a query (bug report). Usually, the relevance can be measured in both direct and indirect ways. The direct way is to measure relevancy by calculating the textual similarity between bug reports and program elements. Indirect ways usually contain obtaining the bug fixing frequency class name similarity and other features according to the project metadata; gaining additional knowledge by learning API documentation and project documentation. Some researchers calculate semantic similarity by extracting semantic features of text from the perspective of semantics. This kind of method mainly constructs multi-layer neural networks. The features are extracted and calculated by deep learning.

The direct way of using textual similarity is only helpful if the bug report contains hint information for localization (i.e., a class name or identifier name in the corresponding source files). For those bug reports that that lack hint information, term matching that only stays at a superficial level is incapable of handling it. Therefore, both the textual similarity and the deep semantic information are considered in our approach to locating the buggy source files.

B. Word Embedding

The raw textual data should be encoded to word embedding vectors firstly for feeding it deep learning models for feature extraction. Word embedding aims to map each word to a dense vector in a low-dimensional space where the embedding vector representations of similar words are also close to each other [12]. Researchers have proposed various word embedding models for natural language processing, including word2vec, which relies on CBOW or Skip-gram to build neural word embedding [13]. The CBOW model predicts the centre word from its surrounding words, whereas the Skip-gram model predicts the surrounding words of a given centre word.

C. Convolutional neural network

Convolutional neural networks (CNN) are based on multi-layer perceptrons and biological processes. It was initially widely used in the field of computer vision [14]. The CNN network consists of three main layers: convolutional layer, pooling layer, and then followed by a fully connected layer. A convolutional layer applies different sizes of filter slides on the input for convolution operation. A pooling layer combines the outputs of neuron clusters at one layer into a single neuron in the next layer. The Rectified Linear Units (ReLU) function increases the network's nonlinear properties.

In order to extract structural information from programs, Mou et al. suggested a tree-based CNN and employed an unsupervised method to learn vector representations of programs [15]. Yoon Kim et al. proposed a text classification model with some changes to the input layer of the CNN by setting the size of the second dimension of the convolutional kernel to be the same as the word embedding layer, which is equivalent to extracting the N-Gram features during sliding [16].

Similar to the convolution characteristics of Zhang et al.'s work, we propose a semantic extraction model similar to a pseudo-siamese network structure, and we will describe it in detail in the III-B section.

III. APPROACH

This section describes our approach, a fault localization method that combines the CNN model and information retrieval. It can capture both the structural information of the programming language and the rich semantic information of bug reports written in natural language, and automatically locate buggy files based on bug reports to improve the performance of fault localization. As shown in Figure 1, our approach consists of two main parts. The first part is a semantic feature extractor based on deep learning models, and the second is a feature extractor based on information retrieval methods. The details of SemirFL are discussed in the following subsections.

A. Pre-Processing

The bug reports and source files represented in textual data form must first be pre-processed to take the data represented in textual content as input for the following phrases. The pre-processing operation includes the following four steps: 1). **Tokenizing.** Use a tokenizer to convert the text into a sequence of tokens; remove punctuation; 2). **Filtering.** Filter stop words in source files (e.g. "int, public" etc.), and stop words in bug reports (e.g. "a" "the" etc.). This is because stop words occur in large numbers and do not provide unique information; 3). **Stemming.** Each word is reduced to its stem form(e.g., "stemming", "stemmed" share the same root "stem"); 4). **Splitting.** Split compound words into multiple tokens according to CamelCase naming rules (e.g. "myStudentCount" can be split into "my", "student", and "count"). The vocabulary in the bug reports and source files constitute the entire project's corpus after the pre-processing phase.

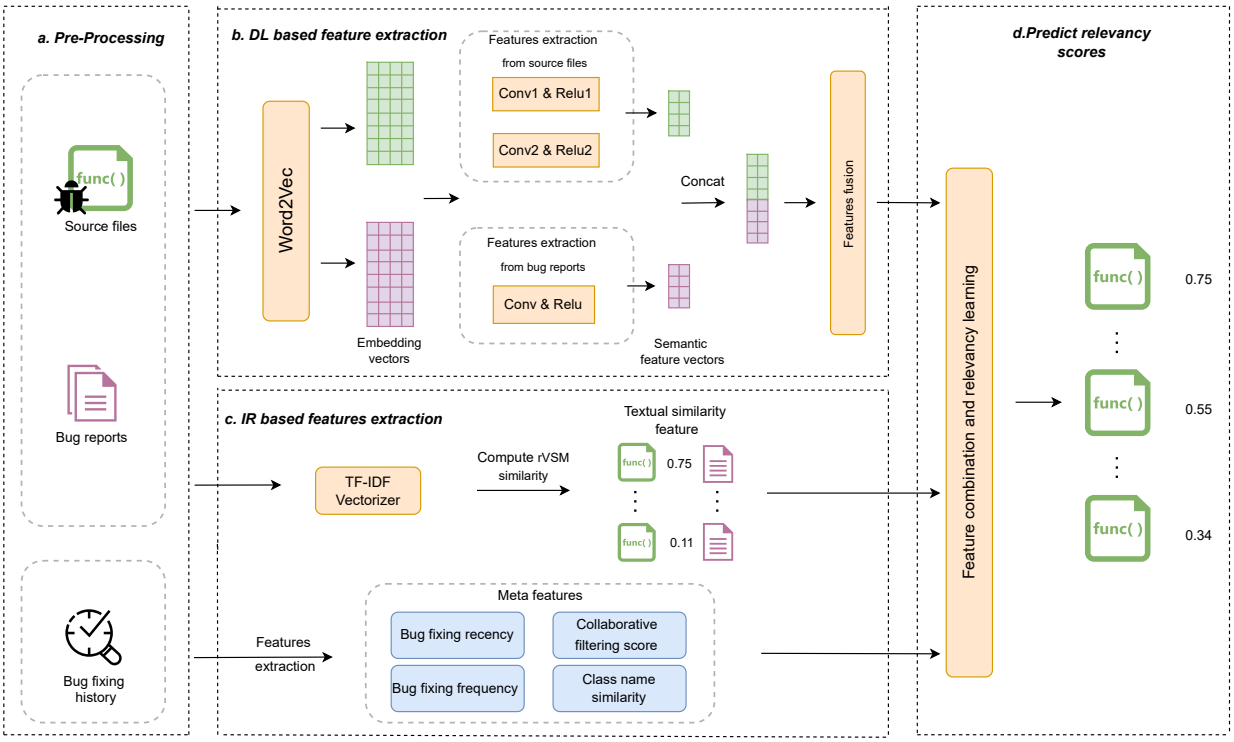

Figure 1. The framework of our approach

B. DL based feature extraction

All words after preprocessing also need to be converted into embedding vectors by the word2vec model before they can be fed into the CNN model. Specifically, we iterate through all the tokens that appear in the bug reports and source files, then eliminate the less frequent tokens and build a corpus for the remaining tokens. Secondly, each word in the bug reports and source files goes to the corpus to find the corresponding position id. Finally, the k-dimensional vector representation of each token is retrieved from the pre-trained word2vec model based on the position id. For tokens that do not appear in the corpus, we randomly initialize them as a k-dimensional vector.

The basic unit that can convey semantic information for natural languages is a word or term. In contrast, in programming languages, statements are the basic unit that can convey semantic information, and the semantics of the programming language must be inferred from the semantics of multiple statements and the interactions of these statements along their execution paths. [9]. Therefore, separate CNN feature extractors are designed for programming and natural language.

We extract features from bug reports in accordance to Zhang et al. and only use one layer of convolution, with convolutional kernels of different sizes, to capture lexical semantic features [16]. For program languages, extracting features only at the lexical granularity will lose the semantic and structural information conveyed between statements of the programming language. Based on this judgment, the CNN feature extractor for program languages uses a two-layer convolutional operation. The convolutional network's first layer extracts the inter-lexical features. The semantic differences between various sentences are captured by the second layer of the convolutional network employing convolutional kernels of various sizes.

An example of a convolution operation on source files is used to detail the specific steps since it contains one convolution operation on bug reports. The word2vec model first represents the source code files as an embedding matrix of size $n \times l \times d$, where n is the maximum number of sentences in the source files, and l is the maximum number of words in the sentence. The first layer of convolution uses a convolution kernel of size $h \times d$ to convolve the input embedding matrix, where h is the number of words covered by the convolution kernel at one time and d is consistent with the dimension of the embedding vector. m feature mappings are applied to each size of the convolution kernel to obtain different types of information. After the first convolution operation, the input dimension is transformed from $n \times l \times d$ to $n \times (l-h+1) \times m$, followed by a maximum pooling operation to extract the essential information from all words in the source file. The output dimension of the feature map is transformed to $n \times m$, with each line in the feature map representing a line of the program statement. The size of the convolution kernel for the second layer of convolution is $h \times m$, where h is the number of statements covered by the convolution kernel at one time, and by varying the size of h, the structural information between different numbers of statements can be captured at one time.

Finally, the feature maps extracted from the bug reports and source files are concatenated and fed into the feature fusion layer constructed from the fully connected layer to learn

feature fusion and dimension reduction.

C. IR-based features extraction

The second part of SemirFL combines traditional features based on information retrieval. This section extracts these features from bug reports, source files, and bug-fixing history. For each bug report $b \in BR$, each source file $s \in SF$, where BR is the set of bug reports, SF is the set of source files. We represent it pairwise as (b, s) and then extract textual similarity and four other metadata features separately for each pair of (b, s). The extraction processes are the same as [7][17].

1) Textual similarity feature: The source file corresponding to the bug reports will share some words. Therefore, textual similarity features can help fault localization to some extent.

Firstly, both bug reports and source files are represented as lexical weight vectors $\vec{V}_b$ and $\vec{V}_s$ separately using the TF-IDF method. Then we use the formula (1) to calculate the cosine score between the corresponding vector representations of (b, s) as the text similarity.

$$\cos(b, s) = \frac{\vec{V}_b \bullet \vec{V}_s}{\left|\vec{V}_b\right| \left|\vec{V}_s\right|} \tag{1}$$

Since source file bugs usually exist only in a code fragment, the faulty code section is much smaller than an entire source code file. Moreover, the longer the file length, the role provided by the key terms normalized for fault localization in the documentation is continuously diluted [10]. Therefore, it is important to consider the size of the source file, and the factor of file length should be introduced into the formula constructed as follows:

$$\begin{aligned} Similarity(b, s) &= g\left(n_t\right) \times \cos(b, s) \\ &= \frac{1}{1 + e^{h(n_t)}} \times \frac{\vec{V}_b \bullet \vec{V}_s}{\left|\vec{V}_b\right| \left|\vec{V}_s\right|} \end{aligned} \tag{2}$$

In the formula, n_t indicates the number of all different tokens that appear in the source file s, and $g\left(n_t\right)$ calculates the reciprocal of the length of the source file, $g\left(n_t\right)$ ensures that larger documents receive a higher relevance score, and $h\left(\cdot\right)$ denotes the Min-Max normalization operation [18].

2) Metadata features: Bug Fixing Recency(BFR): New bugs may be introduced in the process of fixing bugs in the source files [19]. Therefore, it is more likely that the recently fixed files still contain errors. Assuming that bug report b' is fixed in file s and is the most recent bug report fixed before bug report b was created, for each bug report $b \in BR$, the time of the creation of that bug report is denoted by $b.time$. We then define the bug-fixing recency feature of the fixed bug as the reciprocal of the time distance between b and b'.

For a pair of bug report b and source file s, the bug fixing frequency score can be defined as:

$$BFR = \frac{1}{b.time - b'.time + 1} \tag{3}$$

However, considering the work of Ye et al. using the difference between months as time difference, presents several problems: 1) in the same month of the bug report in accordance with (3) calculated by the result is 0. 2) for the difference in the same number of days and less than a month, but a cross-month calculated by the result is 0.5, another pair in the same month calculated by the result is 0. Therefore we consider the time dimension from months to days.

Collaborative Filtering Score (CFS): Originally used for recommender systems, and Murphy-Hill et al.'s work has shown that applying it to fault localization can also bring performance improvements [20], as previously fixed files may cause similar bugs, many keywords are shared in bug reports corresponding to similar bug issues. Consider a bug report b and a source file s, and use $br(b, s)$ to denote the set of historical bug reports associated with the repaired source file s prior to the creation of the bug report b. The collaborative filtering score can be defined as:

$$CFS(b, s) = sim(b, br(b, s)) \tag{4}$$

where sim denotes the text similarity between the content of the current bug report b and the summaries of all bug reports in $br(b, s)$ using the rVSM method.

Bug Fixing Frequency (BFF): Source files that require frequent repair may involve many modules that are inherently fault-prone, and are likely to still contain bugs. We count the numbers in which s was fixed prior to the creation of b, and define the bug fixing frequency feature as equation (5):

$$BFF = |br(b, s)| \tag{5}$$

Class Name Similarity(CNS): When a class name that implements a class is specifically mentioned in a bug report, there is a strong likelihood that the source file that implements the class indeed corresponds to the bug report. This likelihood increases as the class name get longer and more precise. To acquire CNS features, we compute CNS features as (6) by determining if the names of each class in the source file are also present in the bug report:

$$CNS = \begin{cases} \mid s.ClassName \mid & \text{if } s.ClassName \in b \\ 0 & \text{else} \end{cases} \tag{6}$$

where $s.ClassName$ indicates the class name that appears in b and $|s.ClassName|$ indicates the length of that class name.

D. Predict relevancy scores

The semantic features extracted by deep learning above are fused with the text similarity features and the four metadata features together for feature fusion through a fully connected layer. We hope that the text similarity features and the semantic features extracted by CNN can complement each other very well when connecting the same and different terms. In contrast, the metadata features can further bridge the lexical gap between the bug report content and the source files. Eventually, the fault localization problem is transformed into learning a

prediction function $F : \mathcal{BR} \times \mathcal{SF} \mapsto \mathcal{Y}$. $y_{ij} \in \mathcal{Y} = \{1, -1\}$ denotes whether a source code file $s_j \in \mathcal{SF}$ is associated with a bug report $b_i \in \mathcal{BR}$. The optimization function in (8) is minimized to train the prediction function.

$$f(r, s) = \text{Softmax}\left(\sum_{i=1}^{k} w_i * \varphi_i(r, s) + b_i\right) \tag{7}$$

$$\min_{f} \sum_{i,j} \mathcal{L}\left(f\left(r_i, s_j\right), y_{ij}\right) + \lambda\Omega(f) \tag{8}$$

The correlation score is first obtained by weighting the extracted k features ($k = 1, 2, 3$), where $\varphi(r, s)$ represents the feature between the measured bug report and the current source file. Then the predicted value is mapped to the 0-1 interval through a layer of softmax function. The higher the relevance score, the higher the probability that the source file points to the bug report. The $\mathcal{L}$ cross-entropy loss function is then used to calculate the loss between the predicted outcome and the true label for a single data pair.

IV. Empirical evaluation

We used the model to five open-source software projects and contrasted it with three standard representative fault localisation techniques in order to assess the efficacy of the proposed method. We also want to explore the following research questions:

RQ1: How much of a performance improvement SemirFL has compared to existing fault localization methods?

The SemirFL is compared with the most representative of the following several fault localization methods experimentally to evaluate its performance.

- NP-CNN[9]: Uses lexical and program structural knowledge to learn unified fault localization features.
- Learning to Rank[7]: Introduces a functional segmentation of source code files into methods, API descriptions, the bug-fixing history, and the history of code changes that leverages domain knowledge to create an adaptive ranking methodology.
- BugLocator[10]: Uses rVSM and takes into account data from earlier bug-fixing history.

RQ2: What is the contribution of each of the three features, semantic features, textual similarity features, and metadata features, in the overall performance of SemirFL?

A. Dataset

For comparison, four open-source projects are selected from the dataset provided in [7] (Eclipse Platform UI[1], SWT[2], Tomcat[3], and AspectJ[4]). These projects all use GIT as their version control system and Bugzilla as their issue tracking system, both of which have been widely used in prior research. Table I describes these datasets in detail.

[1]http://projects.eclipse.org/projects/eclipse.platform.ui
[2]http://www.eclipse.org/swt/
[3]http://tomcat.apache.org
[4]http://eclipse.org/aspectj/

TABLE I
Benchmark Datasets

Project	Time Range	#Bug Reports	#Java Files
Eclipse	10/01-01/14	6,495	3,454
AspectJ	03/02-01/14	593	4,439
Tomcat	07/02-01/14	1,056	1,552
SWT	02/02-01/14	4,151	2,056

B. Evaluation Metrics

We utilize three measures to assess the performance of the SemirFL: Accuracy@k, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR)

- Accuracy@k: If there is at least one buggy file associated with the current bug report in the returned top-k list, we consider the bug to have been located for the given bug report.
- MAP: A standard metric widely used in information retrieval, defined as the mean of the average precision ($AvgP$) scores of all bug report queries.

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|} \tag{9}$$

when a query may have several documents that are relevant. A single query's average precision is equal to the mean of the precision values it returned.

$$AvgP = \sum_{k \in K} \frac{Prec@k}{|K|} \tag{10}$$

where $Prec@k$ is defined as the ratio of the number of actual buggy files in top-k over k. The higher the MAP value, the better the performance of the method.
- MRR: The rank of the first relevant document in the ranked list is the inverse of the reciprocal rank of a query. The average of the reciprocal ranks of the results to a set of queries Q is the mean reciprocal rank.

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{first_q} \tag{11}$$

where $first_q$ indicates for each query q, the position of the first relevant document in the ranked list.

V. Result analysis

A. Answer to RQ1

To evaluate the effectiveness of SemirFL, we compared it with three state-of-the-art fault localization techniques. Their performance on the four projects is shown in Table II, III. We can observe that the scores of Accuracy@1, MAP, MRR are 0.417 to 0.572, 0.461 to 0.572, 0.501 to 0.623, respectively.

The reason that SemirFL is better than BugLocator is because SemirFL uses a CNN model to learn the deep semantic connections between bug reports and source files beyond simple word matching. In contrast to BugLocator, CNN model

TABLE II
Accuracy comparison with baseline methods

Project	Method	Accuracy@1	Accuracy@5	Accuracy@10
Eclipse	BL	0.271	0.538	0.616
	LR	0.397	0.654	0.743
	NPCNN	0.370	0.610	0.692
	SemirFL	0.442	0.675	0.820
AspectJ	BL	0.216	0.473	0.571
	LR	0.374	0.523	0.637
	NPCNN	0.475	0.547	0.615
	SemirFL	0.510	0.624	0.704
Tomcat	BL	0.354	0.515	0.709
	LR	0.419	0.597	0.717
	NPCNN	0.455	0.620	0.792
	SemirFL	0.572	0.674	0.852
SWT	BL	0.246	0.408	0.533
	LR	0.313	0.524	0.629
	NPCNN	0.365	0.612	0.810
	SemirFL	0.417	0.675	0.855

TABLE III
MAP and MRR comparison with baseline methods

Project	Method	MAP	MRR
Eclipse	BL	0.310	0.540
	LR	0.444	0.511
	NPCNN	0.471	0.547
	SemirFL	0.522	0.623
AspectJ	BL	0.216	0.369
	LR	0.414	0.440
	NPCNN	0.475	0.531
	SemirFL	0.572	0.603
Tomcat	BL	0.354	0.485
	LR	0.425	0.523
	NPCNN	0.455	0.597
	SemirFL	0.580	0.622
SWT	BL	0.246	0.325
	LR	0.354	0.460
	NPCNN	0.365	0.475
	SemirFL	0.461	0.501

can extract local abstract features to fill in the lexical gaps between bug reports and source files.

Compared with Learning to Rank (LR), SemirFL achieves the best performance on all items. Although LR also uses metadata features mined from domain knowledge to locate bugs, LR cannot learn content associations using only metadata features and does not take into account structural information of the programming language.

SemirFL obtained higher accuracy than NPCNN on all items and improved by 7% to 25% on Accuracy@1, 10% to 27% on MAP, and 13% to 20% on MRR. Although NPCNN considers the particular characteristics of program languages compared with natural languages, NPCNN ignores the complementary role of textual similarity features. Moreover, learning the fusion relationship between the deep semantic information, text similarity, and metadata features enrich the features extracted from the CNN model.

B. Answer to RQ2

So far, We have used SemirFL, which contains features in all three dimensions (including semantic feature, textual similarity feature, and metadata features), for comparison with representative baseline approaches. However, it is unclear whether all three feature dimensions are necessary for the technique. As a result, we further explore each feature dimension's significance for SemirFL by removing it from the complete feature sets.

The experimental dataset is set the same as the experiment of RQ1 (i.e., 80% for training and 20% for testing) for each of the four open source projects, and then calculate the average of the Accuracy@1, MAP, and MRR. The comparison results are reported in Table IV in three different settings (SemirFL-sem, SemirFL-rVSM, SemirFL-meta denote the removal of semantic features, textual similarity feature, and metadata features, respectively).

TABLE IV
Impacts of different dimensions of features

Project	Method	Accuracy@1	MAP	MRR
Eclipse	SemirFL	0.442	0.522	0.623
	SemirFL-sem	0.327	0.257	0.347
	SemirFL-rVSM	0.363	0.460	0.537
	SemirFL-meta	0.407	0.412	0.470
AspectJ	SemirFL	0.510	0.572	0.603
	SemirFL-sem	0.475	0.528	0.530
	SemirFL-rVSM	0.314	0.377	0.442
	SemirFL-meta	0.470	0.501	0.511
Tomcat	SemirFL	0.572	0.580	0.622
	SemirFL-sem	0.337	0.411	0.497
	SemirFL-rVSM	0.412	0.493	0.556
	SemirFL-meta	0.499	0.532	0.563
SWT	SemirFL	0.417	0.461	0.501
	SemirFL-sem	0.258	0.263	0.370
	SemirFL-rVSM	0.378	0.346	0.417
	SemirFL-meta	0.341	0.420	0.474

The experimental results are shown in the table IV, and the results show that: (1) as anticipated, using all three dimensions of features will result in the best performance and can have positive effects on fault localization; (2) With the exception of AspectJ, the outcome when the semantic feature is removed is the worst of the four projects, demonstrating the importance of semantic information for SemirFL. The possible reason why semantic features do not play a significant role in AspectJ is that the number of bug reports in AspectJ is much smaller than the other three project datasets, which results in SemirFL failing to learn enough semantic information. (3) Removing metadata features achieves better results than removing text similarity features, indicating that textual similarity features can be better fused with deep semantic features.

VI. Related Work

Information Retrieval Based Fault Location (IRFL) techniques are a major branch of research in fault location. The use of vector space models (VSM) in information retrieval methods is often seen as a simple and effective model. Gay et al. employed VSM to quantify the similarity between bug reports and source files [21]. Zhou et al. propose BugLocator [10] to compute the similarity of the bug report with the rVSM model which considers the document length factor and the similarity between fixed bug reports. Gore et al. proposed

a hybrid detection model by combining the N-Gram model with the VSM [22].

The LDA model has also been applied to the field of IRBL in early research. Lukins et al. proposed a Latent Dirichlet Allocation(LDA)-based technique that viewed bug reports as a combination of different subjects that spit out words with specific probability [23]. Robinson et al. proposed BugScout, a new method based on the LDA topic model, which significantly improved the effect of fault localization [24]. Wang et al. used the LDA model in their method STMLocator. It exploited the performance of the LDA model by using the historical information of the code base for supervised learning [25].

To obtain higher quality query representation, Zhang et al. developed the query expansion technique to incorporate more semantic information into the vectors of bug reports [26]. From a different angle, Chaparro et al.'s attempts to improve bug vectors by removing unnecessary or noisy content from bug reports [27].

Zhang et al. proposed query extension methods to add additional semantic information to the vector representation of bug reports to produce higher-quality query representations [26]. Based on the work of Chaparro et al. the query vector representation is enhanced from a different perspective by reducing redundant or distracting information in bug reports [27].

Spectrum-based fault localization is another widely used technique in locating faults. Furthermore, more information is introduced in this techniques [1][28]. For example, Ju et al. proposed HSFal, a Hybrid Spectrum-based fault localization, to improve the efficiency of locating faults [29], In terms of test cases, Landsberg et al. started to improve them to cover more runtime behaviors [30].

Deep learning-based bug localization methods are better suited for automatically extracting deep semantic information. Huo et al. proposes a unified framework based on the sequential nature of program structure and source code that combines LSTM and CNN models [31]. Lam et al. proposed DNNLOC, which combines a DNN with the rVSM to be effective across different types of similarity [17]. Liang et al. proposed CAST, which combines a tree-based convolutional neural network with customized ASTs [32].

Inspired by the above work, we propose SemirFL, which combines Deep Learning and Information Retrieval methods. Bug reports and source files are processed in different granularity units instead of taking source files as pure text as bug reports. It uses the rVSM model to measure textual similarity. It incorporates metadata features extracted through domain knowledge (bug-fixing recency, frequency, collaborative filtering scores, class name similarity) to connect bug reports with the related buggy files.

VII. Conclusion

This paper proposes SemirFL for fault localization, combining deep learning and information retrieval techniques, including rVSM and four other features using domain knowledge. Different CNN feature extractors are designed separately for natural and program languages to capture the deep semantic information of the programming language. The semantic feature can link bug reports and relevant buggy files that are not textually similar. The metadata feature can further close the lexical gap between bug reports and source files. The semantic features extracted from DL and IR methods can complement each other to achieve higher fault localization accuracy than individual models, according to experimental results on widely used software projects.

In the future, we will explore using large pre-trained models such as BERT to generate word embedding vectors containing contextual semantic information. In order to more accurately represent the high-level semantics of the source code, We will also explore dynamic information such as data flow and control flow in the program from the perspective of dynamic execution.

References

[1] James A Jones and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique". In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 2005, pp. 273–282.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. "An evaluation of similarity coefficients for software fault localization". In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE. 2006, pp. 39–46.

[3] Ruizhi Gao and W Eric Wong. "MSeer: an advanced technique for locating multiple bugs in parallel". In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 1064.

[4] W Eric Wong, Yan Shi, Yu Qi, and Richard Golden. "Using an RBF neural network to locate program bugs". In: *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2008, pp. 27–36.

[5] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. "How practitioners perceive automated bug report management techniques". In: *IEEE Transactions on Software Engineering* 46.8 (2018), pp. 836–862.

[6] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. "A topic-based approach for narrowing the search space of buggy files from a bug report". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 263–272.

[7] Xin Ye, Razvan Bunescu, and Chang Liu. "Learning to rank relevant files for bug reports using domain knowledge". In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014, pp. 689–699.

[8] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization". In: *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 2019, pp. 169–180.

[9] Xuan Huo, Ming Li, Zhi-Hua Zhou, et al. "Learning unified features from natural and programming languages for locating buggy source code." In: *IJCAI*. Vol. 16. 2016, pp. 1606–1612.

[10] Jian Zhou, Hongyu Zhang, and David Lo. "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 14–24.

[11] Zheng Li, Xue Bai, Haifeng Wang, and Yong Liu. "IRBFL: an information retrieval based fault localization approach". In: *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2020, pp. 991–996.

[12] Dongcheng Li, W Eric Wong, Mingyong Jian, Yi Geng, and Matthew Chau. "Improving Search-Based Automatic Program Repair With Neural Machine Translation". In: *IEEE Access* 10 (2022), pp. 51167–51175.

[13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[14] D Bhavana, K Kishore Kumar, Medasani Bipin Chandra, PV Sai Krishna Bhargav, D Joy Sanjanaa, and G Mohan Gopi. "Hand sign recognition using CNN". In: *International Journal of Performability Engineering* 17.3 (2021), pp. 314–321.

[15] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. "Convolutional neural networks over tree structures for programming language processing". In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[16] Ye Zhang and Byron Wallace. "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification". In: *arXiv preprint arXiv:1510.03820* (2015).

[17] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. "Bug localization with combination of deep learning and information retrieval". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension*. IEEE. 2017, pp. 218–229.

[18] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. "Multi-dimension convolutional neural network for bug localization". In: *IEEE Transactions on Services Computing* (2020).

[19] Foyzur Rahman and Premkumar Devanbu. "How, and why, process metrics are better". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 432–441.

[20] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. "The design of bug fixes". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 332–341.

[21] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. "On the use of relevance feedback in IR-based concept location". In: *2009 IEEE international conference on software maintenance*. IEEE. 2009, pp. 351–360.

[22] Alpa Gore, Siddharth Dutt Choubey, and Kopal Gangrade. "Improved bug localization technique using hybrid information retrieval model". In: *International Conference on Distributed Computing and Internet Technology*. Springer. 2016, pp. 127–131.

[23] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. "Bug localization using latent dirichlet allocation". In: *Information and Software Technology* 52.9 (2010), pp. 972–990.

[24] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 23–32.

[25] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Min Li, Feng Xu, and Jian Lu. "Bug localization via supervised topic modeling". In: *2018 IEEE international conference on data mining (ICDM)*. IEEE. 2018, pp. 607–616.

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

[27] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. "Using bug descriptions to reformulate queries during text-retrieval-based bug localization". In: *Empirical Software Engineering* 24.5 (2019), pp. 2947–3007.

[28] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. "Slice-based Statistical Fault Localization". In: *Journal of Systems and Software* 89.0 (2014), pp. 51–62. ISSN: 0164-1212. DOI: dx.doi.org/doi:10.1016/j.jss.2013.08.031.

[29] Xiaolin Ju, Shujuan Jiang, Xiang Chen, Xingya Wang, Yanmei Zhang, and Heling Cao. "HSFal: Effective Fault Localization using Hybrid Spectrum of Full Slices and Execution Slices". In: *Journal of Systems and Software* 90.1 (2014), pp. 3–17. DOI: 10.1016/j.jss.2013.11.1109. URL: http://www.sciencedirect.com/science/article/pii/S0164121213002823.

[30] David Landsberg, Youcheng Sun, and Daniel Kroening. "Optimising Spectrum Based Fault Localisation for Single Fault Programs Using Specifications." In: *FASE*. 2018, pp. 246–263.

[31] Xuan Huo and Ming Li. "Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code." In: *IJCAI*. 2017, pp. 1909–1915.

[32] Hongliang Liang, Lu Sun, Meilin Wang, and Yuxing Yang. “Deep learning with customized abstract syntax tree for bug localization”. In: *IEEE Access* 7 (2019), pp. 116309–116320.