Locating Faulty Methods with a Mixed RNN and Attention Model

Anonymous Author(s)

***Abstract*—*IR-based fault localization* approaches achieves promising results when locating faulty files by comparing a bug report with source code. Unfortunately, they become less effective to locate faulty methods. We conduct a preliminary study to explore its challenges, and identify three problems: *the semantic gap problem*, *the representation sparseness problem*, and *the single revision problem*.**

To tackle the these problems, we propose *MRAM*, a mixed RNN and attention model, which combines bug-fixing features and method structured features to explore both implicit and explicit relevance between methods and bug reports for method level fault localization task. The core ideas of our model are: (1) constructing code revision graphs from code, commits and past bug reports, which reveal the latent relations among methods to augment short methods and as well provide all revisions of code and past fixes to train more accurate models; (2) embedding three method structured features (token sequences, API invocation sequences, and comments) jointly with RNN and soft attention to represent source methods and obtain their implicit relevance with bug reports; and (3) integrating multi-revision bug-fixing features, which provide the explicit relevance between bug reports and methods, to improve the performance.

We have implemented *MRAM* and conducted a controlled experiment on five open-source projects. Comparing with state-of-the-art approaches, our *MRAM* improves MRR values by 3.8-5.1% (3.7-5.4%) when the dataset contains (does not contain) localized bug reports. Our statistics test shows that our improvements are significant.

***Index Terms*—fault localization, code revision graph, recurrent neural network, soft attention**

I. Introduction

With the increase of software's complexity and scale, endless emerging bugs consume a huge amount of development time and effort. It is rather difficult and time-consuming for developers to locate faulty code precisely, especially when a project has many source files. Program debugging and fault localization can take up to 50% of developer's working hours and 25% of total software development costs [1]. As a result, efficient fault localization techniques are desirable to reduce the costs.

When programmers encounter bugs, they often report them through issue trackers. Given a bug report, *information retrieval (IR)-based* fault localization approach reduces the localization problem to a search problem: bug reports are queries, and code snippets are answers [2]–[5]. Most existing works [2], [3], [6] localize faults at the granularity of file level. However, according to a survey on desirable fault localization approaches [7], 51.81% programmers prefer to locate faulty methods, and only 26.42% of programmers are satisfied with locating faulty files. Therefore, recent approaches [4], [5] start to explore locating faulty methods, but their effectiveness is less impressive.

As the number of methods are much more than source files, intuitively, locating faulty methods is more challenging than locating faulty files. Besides this issue, we identify three challenging problems in locating faulty methods (Section III): **(1) The semantic gap problem.** Most IR-based approaches treat both bug reports and source files as natural language texts, and use bag-of-words features to correlate bug reports and source files in the same lexical feature space. However, the terms in bug reports are written in natural languages, and may differ from the tokens in source code. Thus, there is a semantic gap between bug reports and source methods. Even though embedding techniques [5] have been used to encode bug reports and source methods into the same space, they usually ignore the syntax and semantics of code, and thus still suffer from this problem. **(2) The representation sparseness problem.** Short methods are popular in software projects. For example, in Tomcat, we find that more than 70% of methods have fewer than 5 statements, and more than 45% of methods have fewer than 3 statements. With only several statements, short methods are ambiguous, and difficult to be matched against a bug report. **(3) The single revision problem.** A source method in a software repository often has a long revision history, but the prior approaches typically analyze only the latest code to locate faulty methods. This problem results in lacking of sufficient historical data to train an accurate model.

To resolve the three problems, we propose a mixed neural network called *MRAM* for locating faulty methods. *MRAM* combines method structured features and bug-fixing features through code revision graphs, and it identifies the faulty methods of a bug report accurately with recurrent neural network (RNN) [8], soft attention mechanism [9], and multi-layer perception (MLP) [10]. Compared with the state-of-the-art approaches [4], [5], [11], our experimental results show that *MRAM* significantly improves their effectiveness. For example, the average MAP is increased from 0.0305 of Youm *et al.* [4], 0.0249 of Rahman *et al.* [11], and 0.0311 of Zhang *et al.* [5] to our 0.0606. We used the Mann-Whitney U Test [12] to compare our top recommendations with those of the prior approaches, and the results show that our improvements are significant. *MRAM* is able to make these improvements, because it considers both implicit and explicit relevance between bug reports and methods, and is the first to enrich short methods with their past fixes.

This paper makes the following contributions:

1) **A mixed RNN and Attention model**. *MRAM* uses RNN and soft attention to merge extra structural information of source methods to obtain their implicit relevance to bug reports, which solves the semantic gap problem. Then, the implicit relevance along with three bug-fixing features, which provide the explicit relevance, are combined to calculate the correlations between bug reports and methods.
2) **Code revision graphs**. In this paper, we build code revision graphs from past code commits and bug reports. As our graphs reveal latent relations among methods, they are useful to expand short methods, and thus resolve the representation sparseness problem. Furthermore, we extract accurate bug-fixing features from our graphs. As our features are extracted from the revision history of a method than its latest version, we resolve the single revision problem.
3) **An open source tool and evaluations**. We have released our tool and dataset on Github[1] and evaluated it on five large scale open source Java projects. The results show that it could localize bugs at the granularity of methods more precisely than baselines, achieving the MAP of 0.0606 and the MRR of 0.0764. We perform a series of empirical experiments to show the improvement of *MRAM* over the prior approaches.

II. Background

A. Bidirectional RNN for Sequence Embedding

Embedding is a widely used technique to boost the performance of natural language processing (NLP) tasks. It learns to map entities like words, phrases, sentences, or images to vectors of real numbers in a dense vector space. Word embedding is a typical embedding technique that produces a fixed-length vector for a unique word, where similar words are close to each other in the vector space. It can be represented as a function $E : \mathcal{V} \rightarrow \mathbb{R}^d$ that takes a word w in vocabulary $\mathcal{V}$ and maps it to a vector in a d-dimensional vector space.

Convolutional neural network (CNN) and RNN are the two main types of deep neural network architectures for embedding technique. Compared to CNN, RNN performs better at modeling unit sequence. Trained to predict the next symbol in a sequence, RNN models the probability distribution of a sequence. Therefore, RNN achieves impressive results in sequence embedding. Among RNN variants, bidirectional RNN (BRNN) [13] consists of two unidirectional RNNs in opposite directions, therefore can utilize information from past and future states simultaneously to capture the context of the sequence.

In a sentence $T = t_1, ..., t_{N_T}$, each word t_i is mapped to a vector $\boldsymbol{t_i}$ with d-dimensional by word embedding [14]:

$$\boldsymbol{t_i} = Embedding(t_i). \tag{1}$$

Then each of the vector $\boldsymbol{t_i}$ along with the preceding hidden state $\overrightarrow{h}_{i-1}$ updates the hidden state of forward RNN from front to back; and then updates the hidden state of backward RNN from back to front with the following hidden state $\overleftarrow{h}_{i+1}$. Next, $\overrightarrow{h_i}$ and $\overleftarrow{h_i}$ are concatenated to present current hidden state:

$$\begin{aligned} \overrightarrow{h}_i &= \tanh(\boldsymbol{W_M^f}[\overrightarrow{h}_{i-1}; \boldsymbol{t_i}]), \\ \overleftarrow{h}_i &= \tanh(\boldsymbol{W_M^b}[\overleftarrow{h}_{i+1}; \boldsymbol{t_i}]), \\ \boldsymbol{O_i} &= \boldsymbol{W_M}[\overrightarrow{h}_i; \overleftarrow{h}_i] + b, \end{aligned} \tag{2}$$

where $[a; b] \in \mathbb{R}^{2d}$ represents the concatenation of two vectors, $\boldsymbol{W_M^f} \in \mathbb{R}^{2d \times d}$, $\boldsymbol{W_M^b} \in \mathbb{R}^{2d \times d}$, $\boldsymbol{W_M} \in \mathbb{R}^{2d \times d}$ and b are the matrices of trainable parameters, $\overrightarrow{h}_i \in \mathbb{R}^d$ and $\overleftarrow{h}_i \in \mathbb{R}^d$ represent the forward and backward hidden states for timestep i, $\boldsymbol{O_i} \in \mathbb{R}^d$ is the output of BRNN for token t_i, and $\tanh$ is an activation function of the BRNN.

Finally, all the hidden states are fed into an output layer to get the vector representation of the sentence T for a specific task. A typical output layer is a maxpooling function [15]:

$$\boldsymbol{m} = \text{maxpooling}([\boldsymbol{O_1}, ..., \boldsymbol{O_{N_M}}]). \tag{3}$$

B. Soft Attention Mechanism for Text Expansion

Attention mechanism has become an important component in deep neural networks for diverse application domains. The intuition of attention mechanism comes from the human biological systems: the visual processing system of humans tends to focus selectively on parts of the image while ignoring other irrelevant information [16]. Among various attention mechanism, soft attention captures the global latent information of a sequence by using a weighted average of all hidden states of the input sequence. This approach is widely used in NLP tasks such as machine translation, text classification, and text expansion.

Take the text expansion as an example. Given a short text Q and a collection of texts $C = C_1, C_2, ..., C_{N_C}$, where each text C_i is relevant to Q. Text expansion aims to expand Q into a richer representation according to C. Fig.1 shows an example of text expansion using soft attention. The short text Q and each text in C are first mapped into two d-dimensional vectors by sequence embedding, and then are matched to get an attention score s_i:

$$s_i = \tanh(\boldsymbol{W_q}Q + \boldsymbol{W_c}C_i), \tag{4}$$

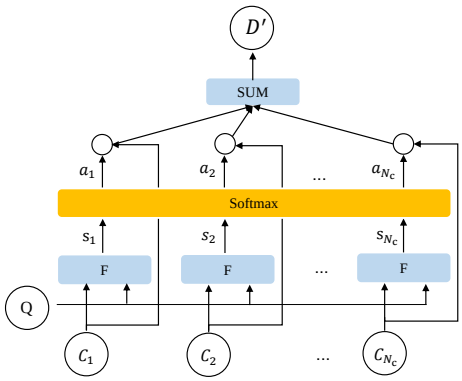


Fig. 1: A soft attention mechanism for text expansion

[1]https://github.com/OrthrusFL/MRAM

TABLE I: Our dataset

Project	Bug report	Valid report	Bug report type			Fixed method		Method			Short method		
			fully	partially	not	all	short*	max	med	min	max	avg	min
AspectJ	530	324	157	69	304	5.296	1.868	32816	19713	7607	20613	14370	5913
Birt	4009	2792	301	224	3484	7.075	4.797	100625	60432	14441	73615	22080	11145
JDT	6058	3278	1101	401	4556	5.023	1.668	49152	24714	7916	34516	17969	6244
SWT	3936	928	1876	472	1588	5.320	0.712	16165	12196	6088	11653	8017	4532
Tomcat	979	607	9	10	960	5.174	2.040	32812	19314	11531	23162	15135	8692

Valid report: the number of bug report that its fixed methods exist in latest code revision; fixed method: the average of fixed methods per bug report; method: the number of methods per code revision; short method: the number of methods whose statements are fewer than 5 per code revision.

where $\boldsymbol{W_q}$ and $\boldsymbol{W_c}$ are trainable parameters in soft attention mechanism. Next, s_i is passed to a softmax function for normalization to compute the attention weight a_i:

$$a_i = \frac{e^{s_i}}{\sum_{j=1}^{N_c} e^{s_j}}. \tag{5}$$

With softmax, the information retrieved from C is computed as a weighted average for $C_1, C_2, ..., C_{N_C}$:

$$D^{'} = \sum_{i=1}^{N_C} (a_i C_i). \tag{6}$$

Finally, the retrieved information $D^{'}$ is used to augment the representation of Q. A typical way is to concatenate the vector of Q and $D^{'}$ as a new Q.

III. Preliminary Study

In this section, we conduct a preliminary study to analyze the problems of locating faulty methods.

A. Setting

The benchmark dataset created by Ye *et al.* [3] from open-source Java projects is used for our experiments. This dataset is widely used in evaluating approaches that locate faulty files, and we make it adapt to the faulty methods locating task. On this dataset, we conduct a preliminary analysis of three problems: the semantic gap problem, the representation sparseness problem, and the single revision problem. We notice that the discussions of some bug reports already identify faulty files or methods [17], [18], and we call such bug reports *localized bug reports*. As Kochhar *et al.* [17] did, we categorize the bug reports in our dataset into three categories:

1) fully localized: faulty methods are identified;

2) partially localized: faulty methods are partially identified;

3) not localized: faulty methods are not identified.

Column "Bug report type" of Table I lists their numbers.

B. Result

An analysis of our dataset can reveal the main problems of faulty methods locating task:

1) The semantic gap problem. The terms used in bug reports written in natural languages evidently differ from the tokens used in methods written in programming languages. We adopt TF-IDF to measure their textual similarity. The result shows that the average textual similarity between bug reports and their *fixed* methods is 0.0153, while that between bug reports and their *irrelevant* methods is 0.0149. Obviously, there is a big semantic gap between bug reports and methods, and a typical IR-based approach is not able to identify those faulty methods by matching textual similarity.

2) The single revision problem. As shown in column 2-3 of Table I, a lot of bug reports can not be localized when only using the latest code revision. For example, there are only 928 bug reports in SWT that all fixed methods exist in the latest code revision. One potential reason is that code changes occur more frequently in method level, and a great number of methods are deleted or changed across neighboring code revisions.

3) The representation sparseness problem. As shown in column 6-7 of Table I, nearly 2/3 of all methods are short. It is difficult to handle the sparse representation when localizing short methods. Moreover, these short methods cannot be simply filtered out for their relatively large proportion of faulty methods.

IV. Code Revision Graph

To learn more faulty localization knowledge from project historical data, we construct code revision graphs from multi-revision code, commits, and past bug reports.

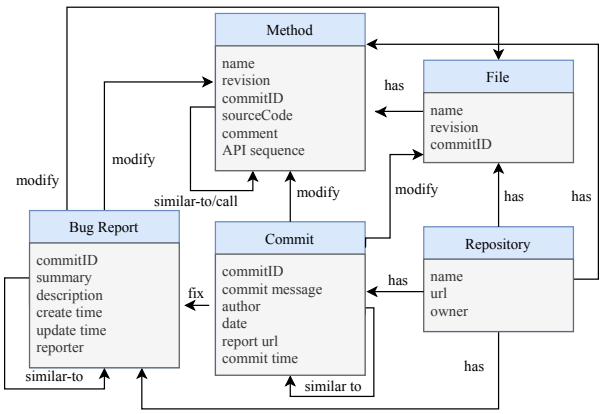


Fig. 2: The schema of our code revision graphs

As shown in Fig. 2, code revision graphs define the following entities and relations between them: repository, bug report, commit, file, and method. The code entities, file and method, have a "revision" attribute. The code revision graphs offer the following benefits:

1) efficiently providing all revisions (not just the latest revision) of code, bug reports, and commits to train the model;

2) alleviating the representation sparseness problem by deducting related methods to expand short methods;

3) calculating multi-revision bug-fixing features to improve the performance of fault localization.

A. Constructing Graph

Given a repository, we use Spoon [19], an open-source library to parse Java source code to get code entities: files and methods, and extract *has*, *modify* and *call* relations. The approach presented by Dallmeier *et al.* [20] is used to obtain the *fix* relations between code commits and bug reports. In particular, we apply a structural-context similarity measurement, *SimRank* [21], to identify the *similar-to* relations. Let S^b_{ij} denote the similarity between bug reports b_i and b_j, S^m_{ij} denotes the similarity between methods m_i and m_j; $M(b_i)$ denote the set of methods modified by b_i; and $B(m_i)$ denote the set of bug reports that modified m_i. *SimRank* has the following three steps: In step 1, S^b_{ii} and S^m_{ii} are set as 1 and will not be updated later. All other similarity scores are initialized as 0. In step 2, similarity scores are updated iteratively (5 rounds of iteration) by the following equations:

$$S^b_{ij} = \frac{C}{|M(b_i)||M(b_j)|} \sum_{k=1}^{|M(b_i)|} \sum_{l=1}^{|M(b_j)|} S^m_{kl} \,, \tag{7}$$

$$S^m_{ij} = \frac{C}{|B(m_i)||B(m_j)|} \sum_{k=1}^{|B(m_i)|} \sum_{l=1}^{|B(m_j)|} S^b_{kl} \,, \tag{8}$$

where C is the rate of decay as similarity flows in the code revision graphs, which is set 0.8. In step 3, we pick the pairs that have a similarity score larger than 0.001 (except S^b_{ii} and S^m_{ii}) as similar bug reports and methods. Compared to content-based similarity measurements, *SimRank* works better for short methods. Here, the *similar-to* relations are computed directly from revision graphs. As these relations define the similarity of the bug fixing history, they are useful to locate faulty methods.

It is expensive to build our graphs from scratch every time a new version is submitted. Instead, we update our graphs incrementally. In particular, the modifications of a revision are classified into three types: *additions*, *deletions*, and *modifications*. The code entity of an addition is added into code revision graphs; that of an deletion is marked as deleted; and that of a modification is linked to its new revision with an *update* relation.

Fig. 3 illustrates how we merge code entities from two neighboring revisions. In revision 1, file F contains three methods: A, B, C; in revision 2, method B is deleted from F, method D is added into F while method C is modified. In the code revision graph, a method before and after modification are treated as two different entities and identified by the "revision" attribute. This solution is limited to a linear history of revisions. In this way, it takes only several minutes to maintain and update code revision graphs.

B. Calculating Bug-Fixing Feature

Code change history is an important metric in detecting defects [22]. Therefore, we select three types of

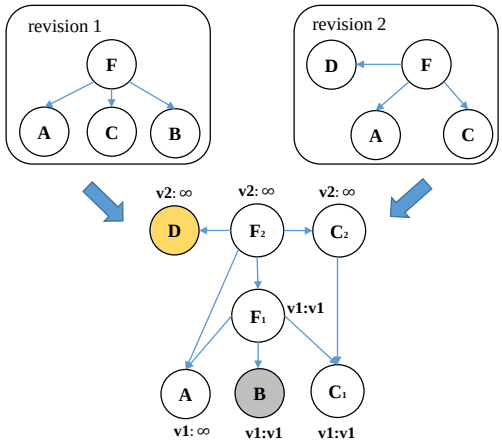


Fig. 3: An example of merging code entities from two neighboring revisions. The yellow circle represents an added entity and the gray circle represents a deleted entity.

bug-fixing features which are widely used in fault prediction models to improve the performance of fault localization: collaborative filtering score, bug fixing frequency score, and bug fixing recency score. Further, we revise the collaborative filtering score to fit the faulty methods locating task, since it heavily depends on the fix relations which may not be sufficient in a software repository. The algorithms to calculate the bug-fixing features are introduced as follows.

1) Revised Collaborative Filtering Score (rcfs): Collaborative filtering score calculates the relevant values between a new given bug report b_i and all source methods M in its before-fix code revision according to the history of previous code revision [2]. Its calculation relies on *similar-to* relations between b_i and its previous bug reports $B_{prev}(b_i)$, and *fix* relations between $B_{prev}(b_i)$ and M. However, there are no existing *similar-to* relations of b_i because *SimRank* could only be used for localized bug reports. Moreover, our code revision graphs do not have many *fix* relations, and may underestimate the similarity values. Therefore, we propose Algorithm 1 to compute *rcfs* for each m in M when given a new bug report b_n. In Step 1, we revise the cosine similarity values between b_n and $b_i \in B_{prev}$ with S^b. In Step 3, *cfs* is revised into *rcfs* via S^m.

2) Bug Fixing Recency Score (bfrs): We calculate *bfrs* based on the assumption that changing code may introduce more faults, which has been proven correct at locating faulty files [23]. That is to say, a method that fixed recently has a high probability to be a faulty method in the near future. Given a bug report b_i and a method m_i, b^p_i is the bug report that is most recently fixed in m_i, then *bfrs* of m_i for b_i can be calculated as follows.

$$bfrs(m_i) = \frac{1}{k+1} \,, \tag{9}$$

where k counts the number of the month between b_i and b^p_i. If m_i has never been fixed before b_i is reported, then *bfrs* of m_i is 0.

Algorithm 1: The procedure of calculating *rcfs*

Input: b_n, M, S^b_{ij}, S^m_{ij}, $B_{prev}(b_i)$
Output: $rcfs_{m_i}$, for $m_i \in M$

/* Step 1: get S^b_{ni}, for $b_i \in B_{prev}(b_n)$ */
for $b_i \in B_{prev}(b_n)$ **do**
  $S^b_{ni} = \sum_{j=1}^{|B_{prev}(b_n)|} S^b_{ij} * cosSim(b_j, b_n) + cosSim(b_i, b_n)$

/* Step 2: get cfs_{m_i}, for $m_i \in M$ */
M_{b_i} denotes methods that are modified by b_i
for $m_i \in M$ **do**
  for $b_j \in B^{prev}$ **do**
    if $m_i \in M_{b_j}$ **then**
      add b_j into B_{m_i}
  $cfs_{m_i} = \sum_{j=1}^{|B_{m_i}|} S^b_{nj} * \frac{1}{|M_{b_j}|}, b_j \in B_{m_i}$

/* Step 3: revise cfs_{m_i} with S^m */
for $m_i \in M$ **do**
  $rcfs_{m_i} = \sum_{j=1}^{|M|} cfs_{m_j} * S^m_{ij} + cfs_{m_i}$

3) Bug Fixing Frequency Score (bffs): bffs measures how often a method is fixed. It is based on the assumption that the more frequently a method has been fixed, the more likely it is the method causing the fault. Given a bug report b_i and a source method m_i, the *bffs* of m_i is scored as the number of times that m_i has been fixed before b_i is reported.

V. *MRAM*

We propose a neural network named *MRAM* for locating faulty methods. This section describes its network architecture, detailed design of main components, and model training.

A. Overall Architecture

Fig. 4 shows the overall architecture of *MRAM*, consisting of three main components:

1) SMNN (semantic matching network), which captures both semantic and structural information of a source method with bidirectional RNNs and soft attention. In this way, the source method can be matched with the bug report accurately in a unified vector space.

2) MENN (method expansion network), which enriches the representation of a method with short length by retrieving the information from its relevant methods;

3) FLNN (fault localization network), which predicts the fault probability of a method by combining both its implicit reference and explicit relevance to the bug report.

B. Semantic Matching Network

To bridge the gap between natural language and programming language, semantic matching network (SMNN) embeds bug reports and source methods into a unified vector space. Different from bug reports, methods provide additional structural information besides the lexical terms and shouldn't be treated as plain texts. Therefore SMNN represents the source method by its three method structured features: the token sequence, the API invocation sequence, and the method comment. Inspired by the attention mechanism for text expansion, we use it to embed multi-sources of information jointly. Specifically, the three features along with the bug report are first embedded as a vector representation by bidirectional RNN respectively; then, taking the bug report as a reference, a soft attention mechanism is used to retrieve crucial information from vectors of three features to represent the whole method. Thereafter the vector of the bug report is matched against the vector of the method by an MLP.

Consider a method $\mathcal{S}$=$[M, A, C]$ and a bug report $\mathcal{R}$, where M=$m_1, ..., m_{N_M}$ is the source method represented as a sequence of N_M split tokens; A=$a_1, ..., a_{N_A}$ is the API sequence with N_A consecutive API method invocations; C=$c_1, ..., c_{N_C}$ is the method comment represented as a sequence of N_C split tokens; and $\mathcal{R}$=$r_1, ..., r_{N_R}$ is the bug report represented as a sequence of N_R split tokens. They are first embedded into a d-dimensional vector individually by a bidirectional RNN [24] with a fully connected layer and a maxpooling layer:

$$\begin{aligned}
\vec{h}_t &= \tanh(\boldsymbol{W}^{\boldsymbol{f}}_{\boldsymbol{M}}[\vec{h}_{t-1}; \boldsymbol{m_t}]),\\
\overleftarrow{h}_t &= \tanh(\boldsymbol{W}^{\boldsymbol{b}}_{\boldsymbol{M}}[\overleftarrow{h}_{t+1}; \boldsymbol{m_t}]),\\
\boldsymbol{O_t} &= \boldsymbol{W_M}[\vec{h}_t; \overleftarrow{h}_t] + b,\\
\boldsymbol{m} &= \text{maxpooling}([\boldsymbol{O_1}, ..., \boldsymbol{O_{N_M}}]),
\end{aligned} \tag{10}$$

where $\boldsymbol{m_t} \in \mathbb{R}^d$ is the embedding vector of token m_t, $[a; b] \in \mathbb{R}^{2d}$ is the concatenation of two vectors, $\vec{h}_t \in \mathbb{R}^d$ and $\overleftarrow{h}_t \in \mathbb{R}^d$ are the forward and backward hidden states for timestamp t, $\boldsymbol{O_t} \in \mathbb{R}^d$ is the output of bidirectional RNN with a fully connected layer for token m_t, $\boldsymbol{W^f_M} \in \mathbb{R}^{2d\times d}$, $\boldsymbol{W^b_M} \in \mathbb{R}^{2d\times d}$ and $\boldsymbol{W_M} \in \mathbb{R}^{2d\times d}$ are the matrices of trainable parameters, $tanh$ is an activation function of the bidirectional RNN, and b is the bias for the fully connected layer. The token sequence M is embedded as a d-dimensional vector $\boldsymbol{m}$. Likewise, the API sequence A is mapped to a vector $\boldsymbol{a}$ in the same vector space, the method comment C is mapped to a vector $\boldsymbol{c}$, and the bug report R is also mapped into a vector $\boldsymbol{r}$.

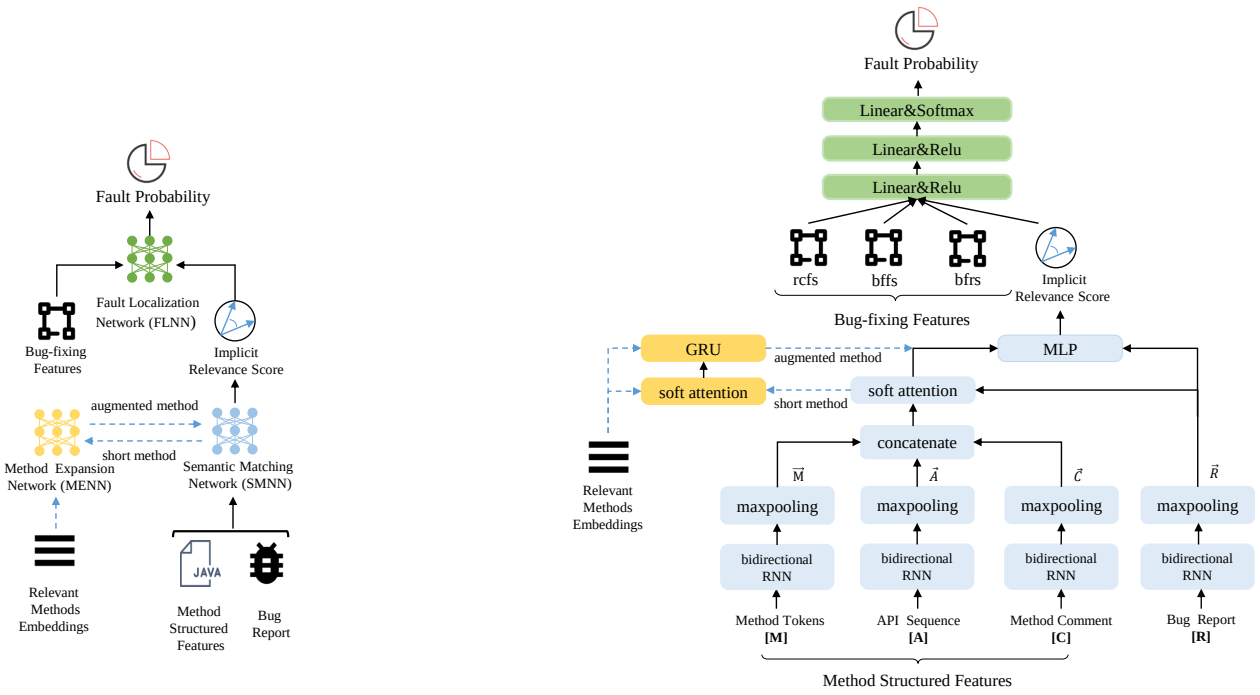
Then, with reference to the vector of bug report $\boldsymbol{r}$, a soft attention is used to dynamically select the key information from $\boldsymbol{m}$, $\boldsymbol{a}$, and $\boldsymbol{c}$ to represent the source method:

$$\begin{aligned}
s_i &= \tanh(\boldsymbol{W_v}\boldsymbol{v_i} + \boldsymbol{W_r}\boldsymbol{r}), \forall \boldsymbol{v_i} \in \{\boldsymbol{m}, \boldsymbol{a}, \boldsymbol{c}\},\\
a_i &= \frac{e^{s_i}}{\sum_{j=1}^{3} e^{s_j}},\\
\boldsymbol{s} &= a_1\boldsymbol{v_1} + a_2\boldsymbol{v_2} + a_3\boldsymbol{v_3},
\end{aligned} \tag{11}$$

where $\boldsymbol{W_v}$ and $\boldsymbol{W_r}$ are the matrixes of trainable parameters in soft attention. The output vector $\boldsymbol{s}$ represents the final embedding of the source method. Finally, the vector of source method $\boldsymbol{s}$ will be matched against the vector of bug report $\boldsymbol{r}$ by an MLP:

$$e = \boldsymbol{W_2}\sigma(\boldsymbol{W_1}[\boldsymbol{s}; \boldsymbol{r}] + b_1) + b_2, \tag{12}$$

where W_i and b_i are the weight and bias for i^{th} MLP layer. e is the explicit matching score between $\boldsymbol{s}$ and $\boldsymbol{r}$. The higher e is, the more closely $\boldsymbol{s}$ is related to $\boldsymbol{r}$.



(a) Overall Architecture

(b) Detailed Architecture

Fig. 4: The structure of our mixed RNN and attention model

C. Method Expansion Network

A common practice to expand a short method is to enrich it with external information, which can be harvested from its relevant methods. Therefore if the method $\mathcal{S}$ is short of length, MENN will enrich its representation by integrating the retrieved information from its relevant methods $M_{rel}(\mathcal{S})$. $M_{rel}(\mathcal{S})$ includes two types of methods: $M_{sim}(\mathcal{S})$ – a set of methods with high SimRank similarity scores for $\mathcal{S}$, and $M_{call}(\mathcal{S})$ – a set of methods that $\mathcal{S}$ calls. Both $M_{sim}(\mathcal{S})$ and $M_{call}(\mathcal{S})$ are collected from code revision graphs. Each $N_i \in M_{rel}(\mathcal{S})$ is also embedded into a vector just like S.

MENN first uses a soft attention mechanism to retrieve the relevant information for S from $M_{rel}(\mathcal{S})$ automatically:

$$\begin{aligned} s_i &= \tanh(\boldsymbol{W_n} \boldsymbol{n_i} + \boldsymbol{W_s} \boldsymbol{s}), \\ a_i &= \frac{e^{s_i}}{\sum_{j=1}^{|M_{rel(S)}|} e^{s_j}}, \\ \boldsymbol{u} &= \sum_{i=1}^{|M_{rel(m)}|} a_i \boldsymbol{n_i}, \end{aligned} \tag{13}$$

where $\boldsymbol{s}$ is the vector of $\mathcal{S}$, $\boldsymbol{n_i}$ is the vector of N_i, $\boldsymbol{W_n}$ and $\boldsymbol{W_s}$ are the matrices of trainable parameters in soft attention, s_i is the attention score for n_i, a_i denotes the attention weight over n_i, and $\boldsymbol{u}$ is the relevant information retrieved from $M_{rel}(\mathcal{S})$ for $\mathcal{S}$.

Then, different from directly adding $\boldsymbol{s}$ and $\boldsymbol{u}$ [5], which may bring about noisy data, a gated recurrent unit (GRU) is employed to integrate the retrieved information $\boldsymbol{u}$ into $\boldsymbol{s}$. As a result, the embedding of expanded method $\hat{\boldsymbol{s}}$ becomes richer and denser than the original embedding $\boldsymbol{s}$.

$$\begin{aligned} \boldsymbol{q} &= \sigma(\boldsymbol{W_q} \boldsymbol{s} + \boldsymbol{U_q} \boldsymbol{u}), \\ \boldsymbol{r} &= \sigma(\boldsymbol{W_r} \boldsymbol{s} + \boldsymbol{U_r} \boldsymbol{u}), \\ \hat{\boldsymbol{u}} &= \tanh(\boldsymbol{W_{\hat{u}}} \boldsymbol{s} + \boldsymbol{r} \circ \boldsymbol{U_{\hat{u}}} \boldsymbol{u}), \\ \hat{\boldsymbol{s}} &= (1 - \boldsymbol{q}) \circ \boldsymbol{s} + \boldsymbol{q} \circ \hat{\boldsymbol{u}}, \end{aligned} \tag{14}$$

where $\circ$ denotes element-wise multiplication, σ is the logistic sigmoid activation function, $\boldsymbol{W_q}$, $\boldsymbol{U_q}$, $\boldsymbol{W_r}$, $\boldsymbol{U_r}$, $\boldsymbol{W_{\hat{u}}}$, $\boldsymbol{U_{\hat{u}}}$ are the weight matrices to learn, $\boldsymbol{q}$ is the weighting vector between $\boldsymbol{s}$ and $\hat{\boldsymbol{u}}$, and $\hat{\boldsymbol{u}}$ denotes the information used to expand $\boldsymbol{s}$. The output $\hat{\boldsymbol{s}}$ replaces the original vector $\boldsymbol{s}$ to represent the source method $\mathcal{S}$ in SMNN.

D. Fault Localization Network

Given a bug report $\mathcal{R}$, FLNN combines the explicit relevance score e obtained from method structured features and the three bug-fixing features (*rcfs*, *bffs* and *bfrs*) to predict the probability that method $\mathcal{S}$ causes the faults described in bug report $\mathcal{R}$. Specifically, these four inputs are first concatenated as a feature vector $\boldsymbol{z}$. For better feature interaction, $\boldsymbol{z}$ is projected into a continuous vector by MLPs.

$$\hat{y_{sr}} = \text{Softmax}(\boldsymbol{W_2} \sigma(\boldsymbol{W_1} \boldsymbol{z}) + b_1) + b_2), \tag{15}$$

where $\boldsymbol{W_i}$ and b_i are the weight of the i^{th} MLP layer, σ is an activation function, and $\hat{y_{sr}}$ represents a prediction probability.

E. Model Training

Now we present how to train the *MRAM* to locate the faulty methods for a given bug report. The high-level goal is: given a bug report $\mathcal{R}$ and an arbitrary method $\mathcal{S}$ with its bug-fixing features $\mathcal{F}$, we want *MRAM* to predict a high relevance score if $\mathcal{S}$ is a faulty method for $\mathcal{R}$, and a low relevance score otherwise.

We construct each training instance as a triple $\langle \mathcal{S}, \mathcal{F}, \mathcal{R} \rangle$ and reduce the task to a binary classification problem. The binary cross-entropy loss function is defined as follows:

$$\mathcal{L} = -\frac{1}{n} \sum_{(s,r)\in\mathbb{D}} (y_{sr}\ log\ y_{sr} - (1-y_{sr}) log(1-y_{sr})), \quad (16)$$

where $\mathbb{D}$ is the training dataset and $y_{sr} \in \{0, 1\}$ represents whether the source method $\mathcal{S}$ is fixed by the bug report $\mathcal{R}$. Intuitively, the ranking loss encourages the relevance between a bug report and its fixed methods to go up, and the relevance between a bug report and methods not fixed by it to go down.

VI. Experiment

We conduct a series of experiments on five open source Java projects, with the aims of answering the following questions:

- **RQ1:** How effective is *MRAM* in faulty methods localization task, compared with state-of-the-art techniques?
- **RQ2:** How do different components contribute to the fault localization performance of *MRAM*?
- **RQ3:** How does *MRAM* perform in cross-project fault localization?

A. Experimental Setup

Baselines. We compare *MRAM* with the following state-of-the-art techniques.

1) *FineLocator* [5] : It is a state-of-the-art method level fault localization approach proposed recently. It augments short methods by their neighboring methods according to three query expansion scores.
2) BLIA 1.5 [4] : It integrates stack trace feature, collaborative filtering score, and commit history feature with structural VSM and achieves better results than previous IR-based approaches like BugLocator [2].
3) Blizzard [11] : It locates faults by employing context-aware query reformulation and information retrieval. It applies appropriate reformulation techniques to refine the bug report in poor quality. Then, the improved bug report is used for the fault localization with Lucene.

Dataset. We reuse the benchmark dataset of Ye *et al.* [3]. Pendlebury *et al.* [25] criticize that it introduces bias, if a model is trained from future data and tested on past data. To resolve the problem, we sort the bug reports chronologically, and divide the bug reports of each project into 10 folds with equal sizes. We train the model on $fold_i, fold_{i+1}, fold_{i+2}$ and test it on $fold_{i+3}$ to ensure that we have sufficient data for training. In addition, when we collect the buggy commit of a bug report, we select its before-fix commit rather than the commit when a report is reported as used by [6], since a found bug shall appear at a before-fix commit.

What's more, we notice that it is impossible to use all the negative samples for training as negative samples are much more than positive ones. For example, the latest revision of Birt has more than 100,000 methods (see table I for detail), and each bug report can have the same amount of negative samples. To reduce the bias, we randomly select 300 negative samples for each bug report when we train the model, but we use all the samples when we test our model.

Metrics. We use three metrics to evaluate the performance of all approaches, namely, *top-ranked accuracy (Top@k)*, *mean average precision (MAP)*, and *mean reciprocal rank (MRR)*, which have been widely used in fault localization and information retrieval [3]–[6].

Top@k indicates the percentage of bug reports for which at least one of its fixed method in the *Top-k* ranked results. It is calculated as follows:

$$Top@k = \frac{1}{|R|} \sum_{r=1}^{|R|} \delta(FRank_r \leq k), \quad (17)$$

where $|R|$ is the number of bug reports, $FRank_r$ is the rank of the first hit faulty method r in the result list [26], and δ is a function which returns 1 if the input is true and 0 otherwise.

MAP measures the ability of fault localization approach to consider the bug report with multiple fixed methods. *MAP* is calculated as follows:

$$\begin{aligned} avgPre &= \frac{1}{|M|} \sum_{m=1}^{|M|} \frac{m}{Rank_m}, \\ MAP &= \sum_{r=1}^{|R|} \frac{avgPre_r}{|R|}, \end{aligned} \quad (18)$$

where $|M|$ presents the number of fixed methods for a bug report, and $rank_m$ is defined as rank of the m^{th} faulty method.

MRR concentrates on the rank of the first fixed method in the ranked list, and it is calculated as follows:

$$MRR = \sum_{r=1}^{|R|} \frac{1}{FRank_r}, \quad (19)$$

where $|R|$ is the number of bug reports, $FRank_r$ is the rank of the first fixed method in the rank list of the r^{th} bug report.

B. RQ1: The Improvements

To answer this RQ, we compare the effectiveness of *MRAM* with state-of-the-art fault localization techniques on the dataset with all bug reports and only *not localized* bug reports. Table II shows the overall performance of *MRAM* and baselines, measured in terms of Top@1/5/10, MAP and MRR.

We have observed that *MRAM* achieves more promising overall fault localization results than other techniques. For example, the average Top@1 value of *MRAM* is 5.134 much higher than *BLIA 1.5*'s 1.815, *Blizzard*'s 1.321, and *FineLocator*'s 1.882. Also, the MAP and MRR values of *MRAM*

TABLE II: The Top@k values of *MRAM* and the baselines

Project	Approach	Top@1		Top@5		Top@10		MAP		MRR	
		w(%)	w/o(%)	w(%)	w/o(%)	w(%)	w/o(%)	w	w/o	w	w/o
Tomcat	*BLIA 1.5*	0	0	0.165	0	0.165	0	0.0004	0.0001	0.0004	0.0001
	Blizzard	0	0	0.228	0	0.457	0.248	0.0032	0.0003	0.0031	0.0003
	FineLocator	0.667	0.324	2.667	1.831	4.833	4.167	0.0204	0.0192	0.0261	0.0239
	MRAM	**1.667**	**1.167**	**5.333**	**5.132**	**9.367**	**8.501**	**0.0378**	**0.0356**	**0.0492**	**0.0462**
AspectJ	*BLIA 1.5*	1.773	3.200	6.028	5.600	7.447	7.200	0.0343	0.0419	0.0391	0.0480
	Blizzard	0.781	0.800	3.385	2.400	4.687	3.200	0.0173	0.0154	0.0183	0.0154
	FineLocator	1.563	1.035	7.188	6.863	9.688	8.753	0.0347	0.0402	0.0503	0.0471
	MRAM	**5.937**	**4.438**	**13.563**	**11.751**	**16.123**	**15.753**	**0.0742**	**0.0686**	**0.0921**	**0.0847**
SWT	*BLIA 1.5*	3.409	2.465	5.681	5.343	6.818	6.849	0.0455	0.0383	0.0457	0.0399
	Blizzard	5.825	2.336	15.426	7.476	19.633	9.345	0.1015	0.0463	0.1063	0.0497
	FineLocator	4.632	3.082	7.634	5.733	9.232	6.032	0.0502	0.0456	0.0533	0.0483
	MRAM	**9.698**	**7.113**	11.422	**8.763**	14.116	**11.321**	0.0987	**0.0765**	**0.1114**	**0.0838**
JDT	*BLIA 1.5*	2.319	1.076	6.714	2.929	9.155	5.020	0.0422	0.0214	0.0461	0.0236
	Blizzard	0	0	0.095	0.060	0.151	0.179	0.0005	0.0004	0.0005	0.0004
	FineLocator	0.917	0.341	3.511	2.843	6.710	4.672	0.0238	0.0183	0.0335	0.0205
	MRAM	**4.613**	**3.672**	**7.951**	**6.932**	**12.920**	**12.513**	**0.0491**	**0.0391**	**0.0702**	**0.0627**
Birt	*BLIA 1.5*	1.576	0.883	5.086	3.092	6.698	4.290	0.0299	0.0189	0.0331	0.0205
	Blizzard	0	0	0.028	0.063	0.028	0.063	0.0002	0.0003	0.0002	0.0003
	FineLocator	1.632	0.023	3.447	1.313	5.324	2.221	0.0266	0.0083	0.0293	0.0095
	MRAM	**3.755**	**3.322**	**6.321**	**7.631**	**10.443**	**9.572**	**0.0434**	**0.0423**	**0.0592**	**0.0561**
Average	*BLIA 1.5*	1.815	1.525	4.735	3.393	6.057	4.672	0.0305	0.0241	0.0329	0.0264
	Blizzard	1.321	0.627	3.832	2.000	4.991	2.596	0.0249	0.0125	0.0257	0.0132
	FineLocator	1.882	0.961	4.889	3.717	7.157	5.169	0.0311	0.0263	0.0385	0.0299
	MRAM	**5.134**	**3.942**	**8.918**	**8.042**	**12.594**	**11.532**	**0.0606**	**0.0524**	**0.0764**	**0.0667**

* w stands for using all bug report for evaluating.
* w/o stands for using only *not localized* bug report for evaluating.

are the best among all studied techniques. One reason for this is that *MRAM* considers both the implicit and explicit relevance between bug reports and source methods, providing more effective fault-diagnosis information analysis.

We have also observed that the performance of all techniques is decreased after filtering out localized bug reports, which could lead a fault localization technique to be not as effective in practice as in experiments. It is worth noting that although the performance of our *MRAM* is also decreased, it only drops slightly compared to other techniques. For example, the MRR value of *MRAM* drops 12.696%, compared to *BLIA 1.5*'s 19.757%, *Blizzard*'s 48.638%, and *FineLocator*'s 22.338%. One potential reason is that when localized bug reports are filtered out, it is difficult for *BLIA 1.5* and *Blizzard* to utilize the keywords (*i.e.* method name) existed both in bug reports and their fixed methods to compute a high textual similarity. *Blizzard* which relies extremely on those keywords drops the most. For example, in SWT, nearly 60% bug reports (see Table I) are localized. Therefore, it is easy to locate their faulty methods by matching their method names with *Blizzard*. However, a new bug report may not contain so many method names, and its effectiveness is thus reduced.

BLIA 1.5 integrates other sources information besides textual similarity, and therefore reduces the impacts brought by filtering keywords compared to *Blizzard*. Both *MRAM* and *FineLocator* rely on embedding technique, and thus elegantly avoid this problem. Moreover, compared to *FineLocator*, which treats methods as plain texts, *MRAM* considers method structured features and incorporates the relevant information by focusing on the representation of bug reports, and thus performs better. Therefore, we have the first finding:

> *Finding 1*: Comparing with state-of-the-art approaches, *MRAM* improves MRR values by 3.8-5.1% (3.7-5.4%), under the settings with (without) localized bug reports.

It is worth noting that the performance of all models in Tomcat is not promising. With manual analysis on bug reports of the Tomcat project, we found that, different from other projects, there are a certain number of bug reports that are raised by the developers themselves to record new features. They are actually not "bug reports", and thus could not be localized by fault localization techniques. We consider it as future works to identify these bug reports automatically, and analyze the influence of these bug reports during the training and evaluation of models.

To investigate our improvements, we further use the Mann-Whitney U Test [12] to compare our Top@k values with those of the prior approaches. The results show that *MRAM* is significantly better than all compared approaches in terms of Top@k at significance level of 0.05. In particular, the p-values varies from 0.00110 to 0.00367 when the dataset contains localized bug reports and p value varies from 0.00013 to 0.00162 when the dataset does not contain such reports.

TABLE III: The impacts of the internal techniques of *MRAM* (the SWT project)

Model	Top@1(%)	△	Top@5(%)	△	Top@10(%)	△	MAP	△	MRR	△
MRAM	9.698	-	11.422	-	14.116	-	0.0984	-	0.1114	-
MRAM-rcfs	5.567	-4.131	9.897	-1.525	10.619	-3.497	0.0660	-0.0324	0.0738	-0.0376
MRAM-bffs	7.113	-2.585	8.763	-2.659	9.381	-4.735	0.0674	-0.0310	0.0816	-0.0298
MRAM-bfrs	6.907	-2.791	7.732	-3.690	8.557	-5.559	0.0646	-0.0338	0.0778	-0.0336
MRAM-SMNN	8.632	-1.066	10.413	-1.009	12.124	-1.992	0.0893	-0.0091	0.1052	-0.0062
MRAM-graph	6.846	-2.852	8.631	-2.791	10.577	-3.593	0.0713	-0.0271	0.0769	-0.0345

C. RQ2: Our Internal Techniques

To get a better insight into *MRAM*, an in-depth ablation study is conducted on SWT project. The main goal is to validate the effectiveness of the critical features or components in our architecture including revised collaborative filtering score (*rcfs*), bug-fixing frequency score (*bffs*), bug-fixing recency score (*bfrs*), method expansion network (MENN), and code revision graph. When without the code revision graph, we use only the latest code revision of SWT to compute the features of methods.

As Table III shows, we can find that:

1) All the investigated components in *MRAM* are helpful in locating faulty methods, and among them *rcfs* contributes mostly on top@1. The benefit of *rcfs* lies in two parts: a) it gets more accurate explicit relevance between bug reports and source methods by revising original collaborative filtering score; b) it improves the localization of short methods significantly by exploiting the relations in code revision graphs.

2) Both *bffs* and *bfrs* are vital for the final performance. This indicates the pattern that the frequently fixed methods and recently fixed methods in the past are the ones that are likely to be fixed in the near future.

3) MENN also benefits for the performance of *MRAM* to some extent, by alleviating the representation sparseness problem. However, it seems that it works not as well as other components. The possible reason is that relations between methods have already been exploited in *rcfs* more explicitly.

4) Code revision graph plays an important part in locating faulty methods. Using only the latest code revision may result in the loss of method historical information, which is caused by code changes (*i.e.* change method signature, deleting method) across code revisions. Thus the calculated bug-fixing features are not complete and accurate to train a model. Then we have the second finding:

> *Finding 2*: Code revision graphs are useful to resolve the single revision problem, and *rcfs* and MENN are useful to handle the sparseness representation problem.

D. RQ3: Cross-project Localization

The localization in previous RQs are within-project. To further investigate the effectiveness of *MRAM*, we extend the fault localization to cross-project scenarios, which will benefit a startup project without enough training data to locate faulty code precisely. Since the experiment is time-sensitive, instead of using k-fold cross validation like prior work [27], the data of one project is used as the training data while the data from another project is treated as the testing data.

TABLE IV: The results of learning from other projects

Task	Top@1(%)	Top@5(%)	Top@10(%)	MAP(%)	MRR(%)
B→A	4.063	6.563	11.875	0.0437	0.0600
J→B	2.926	5.535	8.067	0.0353	0.0502
S→J	3.676	7.021	9.176	0.0396	0.0556
T→S	5.872	8.313	11.072	0.0631	0.0783
A→T	1.332	3.434	7.591	0.0215	0.0306
Average	3.574	6.173	9.556	0.0406	0.0549

Table IV presents the experimental results of *MRAM* for the cross-project localization. The results show that the average MAP and MRR values of cross-project localization are 0.0406 and 0.0549, respectively, slightly worse than within-project localization. This is as expected: in within-project localization, the training data and test data tend to have similar data distribution since they come from the same project, while in cross-project localization, the data distributions are rather different. We also observe that the performance of our *MRAM* in cross-project scenario still outperforms the compared techniques in within-project scenario. This indicates that our model can effectively utilize the data from a project to train a model for another project which has insufficient labelled bug reports. So we have the final finding:

> *Finding 3*: In the cross-project setting, the effectiveness of *MRAM* is only slightly worse than that of the within-project setting.

E. Threats to Validity

For internal validity, the main threat is the potential mistake in our technique implementation and features collection. To reduce this threat, we implement our approach by utilizing state-of-the-art frameworks and tools, such as Pytorch [28], Spoon [19], and Neo4j [29]. For external validity, the main threat is the selection of the studied projects. To mitigate this threat, we evaluate all approaches on five open-source project, which are used in the evaluations of the prior papers [3], [6]. These projects are not only developed by different developers with different programming idioms but also different in size, thus can reflect the real-world situations. In the future, we will extend the scale and scope of studied projects. For construct validity, the main threat is that the metrics used may not

fully reflect real-world situations. To reduce this threat, we use Top@1/5/10, MAP, and MRR metrics, which have been widely used in the previous works [2], [4]–[6]. Our measures consider only the top 10 recommendations, while a lower recommendation can still be a real faulty method. However, we believe that the setting is reasonable. As developers ofte do not inspect more than top 10 results [30], it does not make much difference if a method appears at rank 11 or later.

VII. Related Work

IR-based Fault Localization. One line of work take bug reports as their inputs. Most of them adopt various machine learning techniques to detect the similarity between bug reports and source code [31]–[34]. Kim *et al.* [31] transformed the bug report into a feature vector and use naive Bayes to match it with source files. Lukins *et al.* [32] showed the performance of LDA-based fault localization model is not affected by the size of subject software system. Rao *et al.* [33] concluded simple text models such as VSM are more effective at correctly retrieving faulty files as compared to more sophisticated models such as LDA. These works mostly treated code snippets as the natural language by ignoring the structural information of the program. More recent studies [35] extracted structured information (*i.e.* class name, method name, and comment) from source code to improve fault localization. Wang *et al.* [36] showed these keyword-based techniques rely heavily on the code names that appear in both bug reports and source code, which may lead to a marked decline in performance when localized bug reports are filtered out [17]. Zhang *et al.* [37] introduced deep learning networks to fault localization. Besides matching textual similarity between bug reports and source code, other data source (*i.e.* stack traces, code complexity) are also investigated. For projects with a shorter maintenance history, Huo *et al.* [38] use deep transfer learning to learn a fault-localization model from projects with longer maintenance histories. Akbar *et al.* [39] proposed a code retrieval framework that can be used to locate faults.

Differing from existing techniques, *MRAM* measures both implicit relevance and explicit relevance between bug reports and source methods. To calculate implicit relevance, it does not rely on information retrieval techniques, but uses deep learning techniques to cleverly integrate program structure to represent source methods and match them with bug reports in the same vector space. Thus, it could better understand the semantics of methods and bug reports. It calculates bug-fixing features on all code revisions rather than the latest code revision used in previous techniques. Therefore, the calculated explicit relevance is more accurate and effective.

Spectra-based Fault Localization. The approaches in another line calculate the suspicious scores of program elements using the program's execution information. They require an executable system with many high-quality test cases, which are often unavailable when a bug was reported [40]. Failed test cases and program behavior traces are used for fault localization in [41], [42], [43]. Renieres *et al.* [44] concerned on the source code only executed by the failed test or by all the successful tests, which are considered more suspicious. Liblit *et al.* [45] proposed a statistical debugging technique to isolate bugs in programs with instrumented predicates. Zimmermann *et al.* [46] presented a program state-based technique that contrast program states between executions of a successful test and a failed test using memory graphs. Lo *et al.* [27] applied deep learning techniques to analyze the program execution information of passed and failed test cases. However, test cases are always not available when bugs are reported, which makes those *spectra-based fault localization* approaches not suitable for time-critical debugging tasks [40].

Source Code Representation in Deep Learning. Recently, researchers have investigated a series of techniques to represent source code by using deep learning techniques to improve the performance of various code intelligence tasks [47]–[50]. Zhang *et al.* [47] utilized split abstract syntax tree (AST) to capture lexical and syntactical knowledge of code fragment and achieved the best performance on two common program comprehension tasks: source code classification and code clone detection. White *et al.* [50] proposed a recursive neural network (RvNN) to link patterns mined at the lexical level and a recurrent neural network (RtNN) to link patterns mined at the syntactic level for code clone detection. Gu *et al.* jointly embedded source code and natural language queries into a unified space by using RNN to measure their semantic similarity for code search task. Instead of fusing multiple structured information directly, our proposed *MRAM* takes the bug reports as a reference to integrate token sequence, API invocation sequence and comment to represent the source methods, by using a soft attention mechanism. Moreover, we construct code revision graphs from code, commits and past bug reports, and reveal the latent relationships among methods to augment short methods before method embedding, and thus alleviate the representation sparseness problem.

VIII. Conclusion

In this paper, we propose a mixed deep neural network named *MRAM* for method level fault localization task. *MRAM* learns a unified vector representation of both source methods and natural language bug reports and matches them by vector similarities. Instead of only using the latest code revision, *MRAM* constructs code revision graphs from multiple code revisions and past fixes. Based on such graphs, *MRAM* further expands short methods and calculates bug-fixing features to improve the effectiveness of locating faults at method level. As a proof-of-concept application, we implement a faulty method locating tool based on *MRAM*. We compare *MRAM* with three state-of-the-art approaches on a widely used dataset. Our experimental results show that *MRAM* is effective and outperforms the compared approaches.

In the future, we will investigate to capture program structure (*e.g.* AST, control flow or data flow) to better represent the source methods. To improve the evaluation, we plan to apply *MRAM* to locate real open bug reports in both open-source projects and industry projects.

IX. Acknowledgment

This research is supported by National Natural Science Foundation of China (Grant No. 62032004) and National Key Research and Development Program of China (Grant No. 2018YFB1003903).

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[2] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*. IEEE Computer Society, 2012, pp. 14–24.

[3] X. Ye, R. C. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *SIGSOFT FSE*. ACM, 2014, pp. 689–699.

[4] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Inf. Softw. Technol.*, vol. 82, pp. 177–192, 2017.

[5] W. Zhang, Z. Li, Q. Wang, and J. Li, "Finelocator: A novel approach to method-level fine-grained bug localization by query expansion," *Inf. Softw. Technol.*, vol. 110, pp. 121–135, 2019.

[6] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *ICPC*. IEEE Computer Society, 2017, pp. 218–229.

[7] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[8] S. Kombrink, T. Mikolov, M. Karafiát, and L. Burget, "Recurrent neural network based language modeling in meeting recognition," in *Twelfth annual conference of the international speech communication association*, 2011.

[9] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[10] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[11] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 621–632.

[12] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[13] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[15] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[16] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[17] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: do they matter?" in *ASE*. ACM, 2014, pp. 803–814.

[18] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are bug reports enough for text retrieval-based bug localization?" in *ICSME*. IEEE Computer Society, 2018, pp. 381–392.

[19] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[20] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 433–436.

[21] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *KDD*. ACM, 2002, pp. 538–543.

[22] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 432–441.

[23] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.

[24] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014.

[25] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *Proc. {USENIX}*, 2019, pp. 729–746.

[26] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 357–367.

[27] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *ISSTA*. ACM, 2019, pp. 169–180.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[29] neo4j website. [Online]. Available: https://neo4j.com/

[30] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *ISSTA*. ACM, 2016, pp. 165–176.

[31] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[32] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[33] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.

[34] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 171–180.

[35] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[36] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 1–11.

[37] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 445–455.

[38] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," *IEEE Transactions on Software Engineering*, 2019.

[39] S. Akbar and A. Kak, "Scor: source code retrieval with semantics and order," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 1–12.

[40] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. L. Traon, "ifixr: bug report driven program repair," in *ESEC/SIGSOFT FSE*. ACM, 2019, pp. 314–325.

[41] B. Korel and J. Laski, "Stad-a system for testing and debugging: User perspective," in *Workshop on Software Testing, Verification, and Analysis*. IEEE Computer Society, 1988, pp. 13–14.

[42] A.-B. Taha, S. M. Thebaut, and S.-S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," in *[1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*. IEEE, 1989, pp. 527–534.

[43] H. Agrawal, R. A. De Millo, and E. H. Spafford, "An execution-backtracking approach to debugging," *IEEE Software*, vol. 8, no. 3, pp. 21–26, 1991.

[44] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[45] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[46] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Software Visualization*. Springer, 2002, pp. 191–204.

[47] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *ICSE*. IEEE / ACM, 2019, pp. 783–794.

[48] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[49] X. Huo, M. Li, Z.-H. Zhou *et al.*, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, 2016, pp. 1606–1612.

[50] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.