Improving Fault Localization and Program Repair with Deep Semantic Features and Transferred Knowledge

Xiangxin Meng
SKLSDE Lab, Beihang University
Beijing, China
mengxx@act.buaa.edu.cn

Xu Wang*
SKLSDE Lab, Beihang University
Beijing, China
wangxu@act.buaa.edu.cn

Hongyu Zhang
The University of Newcastle
NSW, Australia
Hongyu.Zhang@newcastle.edu.au

Hailong Sun
SKLSDE Lab, Beihang University
Beijing, China
sunhl@act.buaa.edu.cn

Xudong Liu
SKLSDE Lab, Beihang University
Beijing, China
liuxd@act.buaa.edu.cn

ABSTRACT

Automatic software debugging mainly includes two tasks of fault localization and automated program repair. Compared with the traditional spectrum-based and mutation-based methods, deep learning-based methods are proposed to achieve better performance for fault localization. However, the existing methods ignore the deep semantic features or only consider simple code representations. They do not leverage the existing bug-related knowledge from large-scale open-source projects either. In addition, existing template-based program repair techniques can incorporate project specific information better than deep-learning approaches. However, they are weak in selecting the fix templates for efficient program repair. In this work, we propose a novel approach called TRANSFER, which leverages the deep semantic features and transferred knowledge from open-source data to improve fault localization and program repair. First, we build two large-scale open-source bug datasets and design 11 BiLSTM-based binary classifiers and a BiLSTM-based multi-classifier to learn deep semantic features of statements for fault localization and program repair, respectively. Second, we combine semantic-based, spectrum-based and mutation-based features and use an MLP-based model for fault localization. Third, the semantic-based features are leveraged to rank the fix templates for program repair. Our extensive experiments on widely-used benchmark Defects4J show that TRANSFER outperforms all baselines in fault localization, and is better than existing deep-learning methods in automated program repair. Compared with the typical template-based work TBar, TRANSFER can correctly repair 6 more bugs (47 in total) on Defects4J.

*Corresponding author: Xu Wang, wangxu@act.buaa.edu.cn.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510147

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Fault localization, program repair, transfer learning, neural networks, software debugging

ACM Reference Format:
Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving Fault Localization and Program Repair with Deep Semantic Features and Transferred Knowledge. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510147

1 INTRODUCTION

Fault localization (FL) and automated program repair (APR) are two consecutive tasks for automatic software debugging. Fault localization provides the suspicious fault locations and then APR tries to repair them. Since 80% of the total software cost is spent on finding software faults [46], fault localization approaches have been widely studied to reduce the heavy burden on developers [2, 5, 20, 24, 28, 37, 45, 59, 61, 63, 64]. Fault localization can be conducted at different granularities, such as class [15, 66], method [25, 26, 37], and statement [1, 28]. The finer the localization granularity is, the easier the subsequent bug repair task would be. Thus we prefer fault localization at the statement level. Based on the located faulty lines of code, many APR approaches have been proposed to fix the faults [9, 14, 16, 17, 21–23, 27, 31–33, 35, 36, 39, 50, 55, 58, 67].

Traditional fault localization approaches, such as spectrum-based and mutation-based methods, adopt manually extracted features to locate suspicious code elements [1, 2, 18, 30, 41, 43, 44, 57, 61–63]. Spectrum-based fault localization (SBFL) produces a suspicious score for each code element (i.e., class, method or statement) by performing statistical analysis of failed/passed test cases. Mutation-based fault localization (MBFL) applies specific mutation operators to the original code elements and analyzes the execution of the mutants. Besides SBFL and MBFL, other features including code complexity, code change frequency, and text similarity are also leveraged to improve fault localization [25, 28, 51]. Hybrid approaches have also been proposed to combine multiple suspicious scores from different FL techniques [26, 59].

Recently, deep learning-based approaches, such as DeepFL [25] and DeepRL4FL [28], are proposed for fault localization and have achieved promising results. DeepFL improves fault localization by integrating more than 200 features from spectrum-based, mutation-based, complexity-based, and textual similarity features. DeepRL4FL improves fault localization performance at the statement and method level by incorporating representation learning of code coverage, data dependency, and code semantic. However, these approaches have two major limitations. First, although deep semantic information has been proven to be effective in code representation [3, 4, 42, 49, 53, 54, 56, 60, 65], the deep semantic features of source code are ignored by DeepFL, and DeepRL4FL only considers simple semantic representation by the fully connected layer after the concatenation of word embeddings at the statement level; second, although the bug-related knowledge extracted from historical bug reports and bug-fix commits is helpful for bug detection [29, 47], the existing approaches largely ignore such bug-related knowledge. In this work, we consider the deep semantic features of source code and the transferred knowledge from open-source bug data to further improve fault localization.

Having obtained the suspicious code returned by FL techniques, the template-based repair methods are widely used for automated program repair [9, 14, 16, 21–23, 31, 32, 35, 36, 50, 55, 58]. These methods utilize fix templates predefined or extracted from similar code snippets to repair specific bugs. Among them, TBar [32] uses 15 manually-extracted common fix templates and selects them one by one in a predefined immutable order to generate possible patches. Recently, the encoder-decoder based neural program repair techniques have been presented to generate code repair patches beyond predefined templates [7, 17, 27, 39, 52]. However, they intend to produce the frequent repair patterns and words in the training set and ignore the project specific information [67]. Recoder [67] learns syntax-guided edits (i.e., templates) and replaces placeholders with possible project specific identifiers to optimize neural program repair. Since template-based APR methods naturally adopt local information in filling templates, we leverage the deep semantic features and transferred knowledge to help select better fix templates for efficient repair and remove plausible but incorrect patches.

In this work, we propose TRANSFER, a novel deep learning-based approach to fault localization and program repair by incorporating the deep semantic features and transferred knowledge from the large-scale open-source bug datasets. More specifically:

1) We construct two large-scale bug datasets collected from high-quality GitHub projects to learn bug-related knowledge for fault localization ($Dataset_{fl}$) and automated program repair ($Dataset_{pr}$), respectively. We choose 11 kinds of bug-fix templates from TBar [32] based on the bug popularity and treat them as the possible bug types. Based on these 11 bug types, we then build 11 datasets consisting of 785,134 samples as our fault localization dataset $Dataset_{fl}$. We also use 408,091 bug-fix commits and their bug types to construct our program repair dataset $Dataset_{pr}$. For each bug type, we train a BiLSTM-based binary classifier to predict whether or not one method contains a bug of this type in a specific location. We also train a BiLSTM-based multi-classifier model to predict which fix template should be tried to repair one suspicious statement. The learned classifiers will be reused to transfer bug-related knowledge to target projects.

2) For the fault localization task (TRANSFER-FL), we extract each statement and its contextual method of a target project to obtain its deep semantic features by performing the 11 binary classifiers trained on $Dataset_{fl}$. The semantic features, together with existing spectrum-based and mutation-based features, are further used to train an MLP-based (Multi-layer Perceptron) ranking model for all statements. In this way, we generate the suspicious score of each statement from the MLP-based ranking model for fault localization.

3) For the program repair task (TRANSFER-PR), based on the BiLSTM-based multi-classifier trained on $Dataset_{pr}$ as the transferred knowledge, we further fine-tune the model parameters on the target project. Given an unseen and faulty statement, the 11-dimension vector output by the multi-classifier represents the probabilities for selecting the corresponding fix template. The selection order of fix templates based on these probabilities is used to improve program repair.

We conduct extensive experiments on 395 real software faults from the widely used Defects4J benchmark [19] (V1.2.0) to evaluate our proposed approach. The experimental results show that, for the fault localization task, our FL method (TRANSFER-FL) significantly outperforms all baselines including 3 typical spectrum-based methods, 1 mutation-based method and 2 recent deep learning-based methods. Specifically, our approach increases the faults hit by 13/16/29 on Top-1/3/5, respectively. For the automated program repair task, our APR method (TRANSFER-PR) outperforms both the state-of-the-art template-based repair technique TBar [32] and the state-of-the-art deep learning-based repair technique CURE [17]. Compared with the strong baseline Tbar, TRANSFER (the combination of FL and PR) can work together to correctly repair 6 more bugs (47 in total) on Defects4J.

The main contributions of this paper are as follows:

- We build two large-scale open-source bug datasets, and design BiLSTM-based classifiers to learn deep semantic features of statements for fault localization and program repair.
- We propose TRANSFER, which leverages the semantic-based, spectrum-based, and mutation-based features for effective fault localization and leverages the semantic-based features for effective program repair.
- We conduct extensive experiments on widely-used benchmark Defects4J to evaluate our approach, and the experimental results confirm that our approach is effective.

2 RELATED WORK

Learning-based Fault Localization. Learning-to-Rank [34] is an important approach in information retrieval area, which utilizes supervised machine learning to solve ranking problems. Some recent studies apply the Learning-to-Rank strategy to fault localization [5, 26, 51, 59], which takes multiple features as inputs from different sources, such as suspicious scores from SBFL and MBFL techniques. Among them, the pairwise training is frequently used to rank faulty elements before correct ones. Recently, deep learning-based approaches are proposed, such as DeepFL [25] and DeepRL4FL [28], which achieve promising results for fault localization. DeepFL improves fault localization by integrating more than 200 features from four traditional feature groups but misses the deep semantic features. DeepRL4FL combines a coverage representation approach

with code representation learning for fault localization, which simply uses the fully connected layer with the concatenation matrix of word embeddings at the statement level. Both deep learning-based methods have not captured the deep semantic features of source code well, especially the sequential dependencies of tokens, which limits the effectiveness of statement-level fault localization. Grace[38] is another deep learning-based method which leverages GNNs (Gated Graph Neural Networks) to learn valuable features from the graph-based coverage representation. However, it removes all children nodes of the statement structures of ASTs (Abstract Syntax Trees), which is useful for method-level FL but misses the statement-level semantics. Moreover, the method-level fault localization effectiveness of Grace is worse than DeepRL4FL, so it is not included in our fault localization experiments.

Template-based program repair. Template-based program repair approach is a widely studied research area in APR, which utilizes predefined fix templates to fix specific bugs. The repair process can be basically divided into four steps, i.e., fault localization, fix template selection, donor code search, and patch candidate validation. The first 3 steps of them determine the final effectiveness of the corresponding repair technique. Typical template-based program repair techniques include TBar [32], Simfix [16], Avatar [31], FixMiner [21] and so on. Among them, TBar [32] is the state-of-the-art template-based repair technique, which contains 15 commonly used fix templates and achieves a good performance on the benchmark Defects4J [19]. However, the approach of fix template selection has not been well studied, which influences the entire program repair task.

Deep learning-based program repair. Recently, some studies treat the program repair task as a statistical machine translation task, and adopt the widely-used encoder-decoder architecture to learn to generate possible patches. DLFix [27] proposes an efficient way to embed the contextual information into the faulty statement. CoCoNuT [39] combines CNNs (Convolutional Neural Networks) and a new context-aware neural machine translation architecture to generate patches token by token. CURE [17] pre-trains a language model to extract bug-fix knowledge and propose a new code-aware search strategy to reduce the search space. Recoder [67] learns syntax-guided edits (i.e., templates) and replaces placeholders with possible project specific identifiers to optimize neural program repair. Nevertheless, the project specific information is hard to learn and the frequent repair patterns and words appearing in training sets are more likely to be selected. In contrast, template-based APR methods naturally adopt local information to fill templates, but they cannot effectively select fix templates. In this work, we address the template selection problem by learning useful semantic knowledge from open source code, while retaining the advantages of template-based techniques. The recent deep learning-based program repair methods including DLFix [27], CoCoNuT [39] and CURE [17] are used as baselines in this paper.

3 PROPOSED APPROACH

3.1 Overview

In order to improve the performance of statement-level fault localization and automated program repair, the deep semantic features of statements and the transferred knowledge from large-scale bug datasets are leveraged in our approach. As shown in Figure 1, our approach, TRANSFER, mainly includes three components. The first component (Section 3.2) is designed to learn transferred bug-related knowledge from our two large-scale open-source bug datasets, which includes 11 different binary classifiers to detect whether one statement has corresponding bugs and one multi-classifier to predict which fix template should be used for a faulty statement. Based on the transferred knowledge, the second component (TRANSFER-FL, Section 3.3) aims to improve fault localization with the deep semantic features, and the third component (TRANSFER-PR, Section 3.4) can improve automated program repair by predicting the order of fix templates to be used.

3.2 Learning Transfer Knowledge

3.2.1 Extraction of Bug-Fix Commits for Different Bug Types. We first collect and analyze a large number of historical bug-fix commits in open source projects. Specifically, we collect 2,000 open source Java projects with most stars on GitHub. Four projects (i.e., Joda-Time, Closure compiler, Apache commons-lang, and Apache commons-math) are removed from the collected projects because they also exist in the target benchmark (i.e., Defects4J). We utilize the approach described in [48] to extract all commits relevant to bug fix. Specifically, a commit is considered bug-relevant if its message contains keywords such as "error", "bug", “fix”, “issue”, "mistake", "incorrect", "fault", "defect", "flaw", "type", etc. We keep the commits that modify code in only one method, and in total 1,010,628 commits are collected.

We then identify bug types from the bug-fix commits. Given that the extracted commits are related to bug fix, if the code modifications of a commit match the change actions defined in a specific fix template, the code element before this commit is made is considered to contain the corresponding type of bugs. A fix template [32] defines a pattern of code modifications, which is applied to a faulty code element to help generate possible patches. If a buggy code element c is repaired after applying the fix template ft, we say that c contains bugs with a type corresponding to ft. We collect all 15 fix templates defined in TBar [32], and the corresponding bug type of each fix template is shown in Table 1.

We implement an AST-based (Abstract Syntax Tree) syntax checker through an AST-based code differencing algorithm (similar to GumTree[12]) and a rule-based matching tool that matches the edits to the fix templates shown in Table 1. On the one hand, once a fix template is identified, the code element before the commit is tagged with the corresponding bug type. Note that a commit may match multiple fix templates, because the predefined code change actions in different fix templates may overlap. As the example shown in Table 2, both *Mutate Conditional Expression* and *Mutate Variable* fix templates are matched, while the former considers the modification from $a >= a$ to $a >= b$ and the latter considers that from a to b. On the other hand, if no fix template is identified, the commit will be discarded. Simultaneously, we mark the start line of the statement where modification is located as the faulty position. Still taking Table 2 as an example, the line 2 is marked as faulty position because it is the start line of the *If Statement* which wraps the modification. To verify the correctness of the labeling

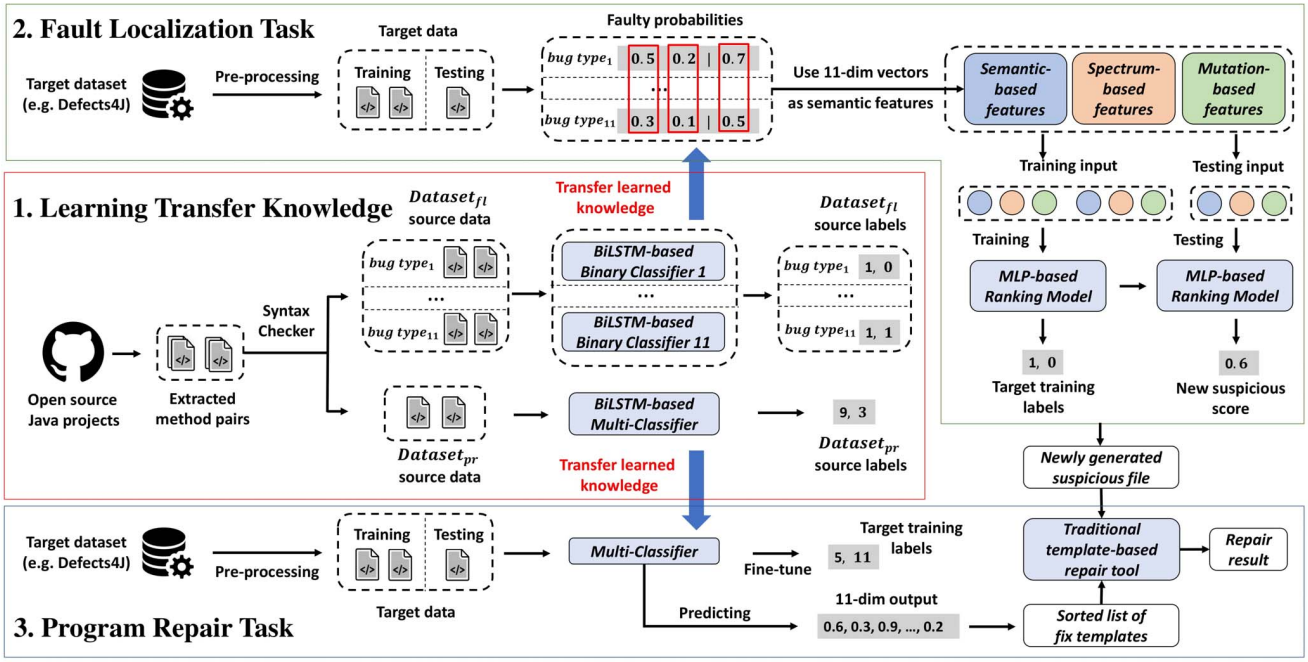

Figure 1: An Overview of TRANSFER

process using our syntax checker, we have randomly sampled 100 results, and checked that they are all accurate.

After applying the syntax checker, we find that the numbers of commits corresponding to 4 bug types are very small (less than 100), which are difficult to be utilized for the subsequent learning tasks, so we only study the remaining 11 bug types (shown in Table 4) in this paper. In total, 408,091 commits are extracted with the annotations of bug types and faulty positions.

3.2.2 *Dataset Construction.*

1) Construction of $Dataset_{fl}$. Having collected the bug-fix commits tagged with bug types, we assign each commit to one or more groups (11 in total) according to its bug type(s). Then, for each group, we further extract all the methods before the commits were made as the positive samples. Each positive sample consists of two parts: 1) the faulty statement, which marked with *<BOS>* and *<EOS>*, and 2) the contextual method of the faulty statement. The first column in Table 3 gives the positive sample to the commit in Table 2 for *Mutate Variable* fix template. In this example, since the faulty code element is the variable *a* in line 2, whose nearest statement-type ancestor is the if statement wrapping it. Thus, we add the marks <BOS> and <EOS> at the start and end position of line 2 respectively. Since the commit is also tagged with the bug type corresponding to *Mutate Conditional Expression*, for this fix template, the generated positive sample is the same as that of *Mutate Variable*, because the nearest statement-type ancestor of the faulty code element *a >= a* is also the if statement.

To collect the negative samples, we propose a new approach. We first look for another statement in the same method, which contains the same necessary syntax ingredients as the positive sample. The necessary syntax ingredients for each fix template (bug type) are underlined in the second column in Table 1. If no such statement is found in the method, we will expand the scope and continue searching in the same Java file. We collect all statements containing the necessary syntax ingredients and the corresponding methods they belong to as negative sample candidates, and then select one of them randomly. However, if no suitable negative samples are found, the corresponding positive samples will be discarded in order to construct a balanced dataset.The second column in Table 3 shows a negative sample corresponding to the already generated positive sample in the first column for *Mutate Variable* fix template. The necessary syntax ingredient of *Mutate Variable* is a *variable* code element, as long as a statement contains at least one variable, it can be selected and marked to generate a negative sample. Thus, both *return a*; and *return b*; statements can be marked. After the random selection, the former is finally marked and the negative sample is generated.

Finally, 11 datasets containing 392,567 positive samples and the same number of negative samples (785,134 in total) are constructed for the subsequent fault localization task (the detailed statistics are shown in Table 4), which are called collectively as $Dataset_{fl}$.

2) Construction of $Dataset_{pr}$. For program repair, given a faulty statement, our goal is to select the correct fix templates to repair it, which can be regarded as a multi-classification task. Thus, for each commit extracted in Section 3.2.1, we only keep the method before the commit was made and categorize it into one of the 11 predefined bug types. As a bug-fix commit can be tagged with one or more bug types, we assign it to the bug type that is deeper in the AST hierarchy. Intuitively, a deeper AST node needs a relatively simpler

Table 1: 15 Fix Templates and the Corresponding Bug Types

Fix Templates	Corresponding Bug Types
Insert Cast Checker	At least one cast expression exists without checking the compatibility of the original type and the cast type.
Insert Range Checker	At least one array (collection) access exists without checking whether the index (key) is beyond the scope.
Insert Null Pointer Checker	At least one field (expression) is used without checking if it has a *null* type.
Insert Missed Statement	Missing a specific type of statement including method invocation/return/if/try-catch statement.
Mutate Conditional Expression	A conditional expression should be added, removed or modified.
Mutate Data Type	The data type used in cast expression or variable declaration expression is incorrect.
Mutate Literal Expression	The literal expression used in the statement is incorrect.
Mutate Method Invocation Expression	The name or at least one of the parameters of the method invocation expression is incorrect.
Mutate Class Instance Creation	*super.clone*() method should be used rather creating a new class instance in an overridden clone method.
Mutate Integer Division Operation	An integer literal is used in division operation and results in the loss of accuracy.
Mutate Operators	At least one relational/arithmetic/instanceof/parentheses operator is incorrectly used.
Mutate Return Statement	The expression in return statement is incorrect.
Mutate Variable	The variable used in the statement is incorrect.
Move Statement	A statement is placed in an incorrect position.
Remove Buggy Statement	A statement should not appear in the current position and is expected to be removed.

Table 2: A Java Code Example Before and After the Commit

	Method Before Commit	Method After Commit
1	public int max(int a, int b) {	public int max(int a, int b) {
2	if (a >= **a**) {	if (a >= **b**) {
3	return a;	return a;
4	}	}
5	return b;	return b;
6	}	}

Table 3: An Example Pair of Positive and Negative samples for the fix template *Mutate Variable*

	Positive Sample	Negative Sample
1	public int max(int a, int b) {	public int max(int a, int b) {
2	<BOS> if (a >= **a**) { <EOS>	if (a >= **a**) {
3	return a;	<BOS> return a; <EOS>
4	}	}
5	return b;	return b;
6	}	}

repair template with a smaller change area, which is preferred by actual developers. For example, for the method (before the commit was made) shown in Table 2, the corresponding AST is constructed in Figure 2. We can see that the code element focused on by *Mutate Conditional Expression* is the conditional expression node whose depth is 4, while the code element focused on by *Mutate Variable* is the variable node *a* (the child of *Right Operand node*) whose depth is 7. Thus, the *Mutate Variable* should be selected as the unique label because it focuses on a deeper AST node.

In this way, we construct a dataset with 11 categories and 408,091 samples in total, which is called $Dataset_{pr}$. Each sample consists of a faulty statement with the contextual method and its corresponding bug type. The detailed statistics are shown in Table 4.

Table 4: Statistics of $Dataset_{fl}$ and $Dataset_{pr}$, where positive and negative samples are seperated by the slashes.

Fix Templates	#Samples in $Dataset_{fl}$	#Samples in $Dataset_{pr}$
Insert Null Pointer Checker	5660/5660	7600
Insert Missed Statement	47120/47120	54445
Mutate Conditional Expression	31167/31167	30530
Mutate Data Type	3293/3293	5178
Mutate Literal Expression	35536/35536	55044
Mutate Method Invocation Expr.	198754/198754	166943
Mutate Operators	1673/1673	2326
Mutate Return Statement	12511/12511	20680
Mutate Variable	29918/29918	35156
Move Statement	7313/7313	7812
Remove Buggy Statement	19622/19622	22377
Total	392567/392567	408091

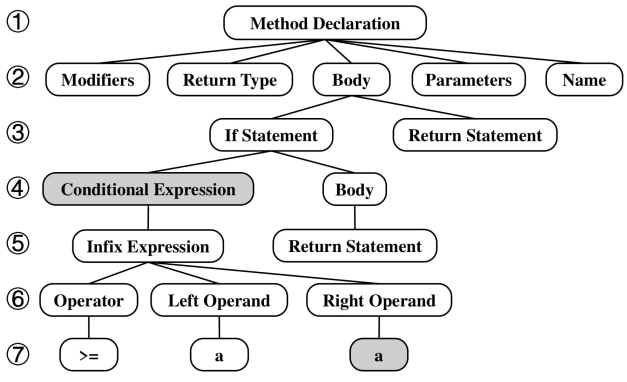

Figure 2: An Example to Illustrate the Selection of the Unique Bug Type

3.2.3 Learning Knowledge for Fault Localization.

1) The design of binary classifiers. We build 11 Bi-LSTM based binary classifiers with same structures to judge whether the bugs with corresponding bug type exist in the code or not. The overall architecture of the model is shown in Figure 3. Since we are studying statement-level fault localization, the input of the model is a token sequence $< t_1, t_2, ..., t_N >$ including the specified statement and its contextual method, where N represents the length of the token sequence. Then, the token sequence is fed into an embedding layer with the pre-trained word2vec [40] parameters $W_e \in \mathbb{R}^{|V| \times d}$ where V is the vocabulary size and d is the embedding dimension of each token, to generate a sequence of token vectors $< e_1, e_2, ..., e_N >$. The vector representation e_k of token t_k can be obtained by:

$$e_k = {W_e}^{\mathrm{T}} x_k \tag{1}$$

where x_k is the one-hot representation of token t_k. Next, we utilize LSTM [13], one type of recurrent neural network, to extract the contextual semantic features containing token sequential dependencies, while the fully connected layer after the concatenation matrix of token embeddings used in DeepRL4FL [28] in statement-level fault localization cannot extract such dependency relationships.

In order to obtain richer dependency information among tokens, we adopt a bidirectional LSTM (Bi-LSTM), the output of which is a new state generated by concatenating the hidden states from both directions at time t:

$$\overrightarrow{h_t} = \overrightarrow{LSTM}(e_t), \quad \overleftarrow{h_t} = \overleftarrow{LSTM}(e_t), \quad h_t = [\overrightarrow{h_t}, \overleftarrow{h_t}] \tag{2}$$

To extract the most important features for each dimension, we keep the hidden states of all time steps, which are then pushed into a stack and sampled by max pooling. We assume that the result after processing of max pooling is g, which is calculated as follows:

$$g = [\max_{1 \le t \le N}(h_{t1}), \max_{1 \le t \le N}(h_{t2}), ..., \max_{1 \le t \le N}(h_{t \cdot 2m})] \tag{3}$$

where m represents the dimension of hidden states. Finally, we put g into a dense layer with $softmax$ activation function, and the output indicates the probability of containing the corresponding type of bugs, which is exactly the semantic feature we need.

2) Training the binary classifiers. We divide the training task into two phases. In the first phase, for each of the 11 bug types, we use $Dataset_{fl}$ for model training, after which the optimal model parameters are saved. We call the knowledge learned in this phase as *transferred knowledge*. In the second phase, each suspicious statement with its contextual method in target datasets is input into the trained models to obtain 11 deep semantic features. In addition, as mentioned above, the annotated statements in both positive and negative samples in $Dataset_{fl}$ must contain the necessary syntax ingredients for its corresponding bug type. Therefore, in the second phase, for each suspicious statement in target projects, we first judge whether the statement contains the necessary syntax ingredients required by each bug type. If it does, this statement and its contextual method will be input into the binary classifier for the corresponding bug type to obtain the output probability, otherwise the probability is set to 0 if the necessary syntax ingredients miss. In this way, the 11-dimension deep semantic features are extracted for all suspicious statements in target projects, which will be used in the subsequent fault localization task.

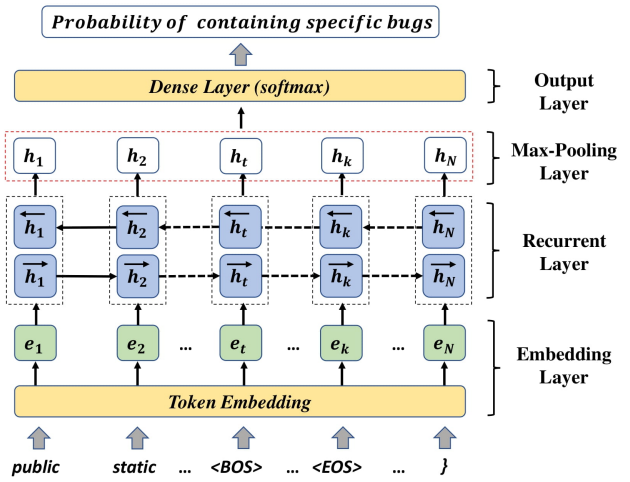

Figure 3: Bi-LSTM based Binary Classifier Model for Learning Transfer Knowledge for Fault Localization

3.2.4 Learning Knowledge for Program Repair.

For selecting correct fix templates to enhance the existing template-based program repair approaches, we build a BiLSTM-based multi-classifier, whose architecture is the same as the binary classifier described in Section 3.2.3, except the dimension of the output layer, which is changed from 2 to 11 (i.e., the number of bug types). We feed the large-scale $Dataset_{pr}$ into the multi-classifier for model training, and in this way, the transferred knowledge for judging which fix templates should be selected first to generate patches is learned, which can be used for the subsequent program repair task.

3.3 TRANSFER-FL: Effective Fault Localization based on Transferred Knowledge

3.3.1 Spectrum-based and Mutation-based Features.

1) Spectrum-based Features. Spectrum-based fault localization (SBFL) is one of the most widely studied fault localization approaches. SBFL approaches take source code and relevant test cases as inputs, and output a sorted list of code elements ordered by suspicious scores, which are calculated from the execution information of test cases. It has been found that the results of SBFL approaches can serve as part of input features for training learning-to-rank models in some recent studies [16, 21, 31, 33]. Learning-to-rank approaches have been proved to help better fault localization by optimizing the combination of features [26, 51]. In this work, we adopt the learning-to-rank method and select 3 most commonly used SBFL techniques (i.e., Tarantula [18] , Ochiai [1] and DStar [57]) to generate features for the spectrum-based feature group. The equations of calculating suspicious scores for the three SBFL techniques are as follows, where $T_f(e)/T_p(e)$ represents the number of failed/passed tests executing code element e, while $T_f(\bar{e})/T_p(\bar{e})$ represents the number of failed/passed tests that do not execute e, and T_f/T_p represents the number of all failed/passed tests.

$$Tarantula: \quad Sus(e) = \frac{T_f(e)/T_f}{T_f(e)/T_f + T_p(e)/T_p} \tag{4}$$

$$Ochiai: \quad Sus(e) = \frac{T_f(e)}{\sqrt{T_f \cdot (T_f(e) + T_p(e))}} \tag{5}$$

$$DStar: \quad Sus(e) = \frac{T_f(e)^*}{T_p(e) + (T_f(\bar{e}))} \tag{6}$$

2) Mutation-based Features. Mutation-based fault localization (MBFL) is another approach which calculates suspicious scores by analyzing the changes of execution results between the original code element and its mutants. Similar to SBFL, the outputs of MBFL approaches can also serve as input features for learning-to-rank models [25, 26]. MBFL approaches need to additionally specify the set of mutation operators as well as the granularities of failure outputs/messages. We adopt the four types of granularities proposed by TraPT [26]: (1) passed/failed information, (2) exception type information, (3) exception type and message, (4) exception type, message and the full stack trace of exception.

As described in Metallaxis [44], the formulae from SBFL approaches can be utilized to calculate suspicious scores for mutants, and the highest score among all mutants is then selected as the suspicious score of the corresponding original code element. Equation 7 takes Ochiai algorithm as an example, where $M(e)$ denotes all mutants of code element e, $T_f^{(m)}(e)$ denotes the number of originally failed tests which are impacted by mutant m, $T_p^{(m)}(e)$ denotes the number of originally passed tests which are impacted by mutant m, and so on. We choose the frequently-used Ochiai algorithm for Metallaxis to extract the mutation-based features. Finally, 4 mutation-based features are extracted because 4 types of granularities are considered which are mentioned above.

$$Sus(e) = \max_{m \in M(e)} \frac{T_f^{(m)}(e)}{\sqrt{T_f^{(m)} \cdot (T_f^{(m)}(e) + T_p^{(m)}(e))}} \tag{7}$$

3.3.2 Deep Semantic Features. Both suspicious scores generated by SBFL and MBFL approaches can be used as input features for a learning-to-rank model to improve the effectiveness of fault localization. However, the semantic features have not been included yet, which can provide useful information from a different aspect. In fault localization research area, especially when program data is compilable and executable, the dynamic execution information is still the most frequently used feature [25, 26, 51]. In contrast, the use of static semantic features is relatively preliminary, such as DeepRL4FL [28], which simply uses the fully connected layer with the concatenation matrix of word embeddings at the statement level. Therefore, our goal is to propose a more effective way of extracting and utilizing contextual semantics to further improve the effectiveness of fault localization. As described in Section 3.2.3, we have obtained the 11-dimension deep semantic feature of each suspicious statement in the target dataset by applying the transferred knowledge learned from the large-scale $Dataset_{fl}$.

3.3.3 Fault localization with transferred knowledge. Through the above steps, three feature groups are obtained, including 3 spectrum-based features, 4 mutation-based features and 11 deep semantic features. We can combine these features in a learning-to-rank model to predict the probability of being faulty for each suspicious statement in the target project. Compared with the task in Section 3.2.3 to judge whether a suspicious statement contains bugs with specific types, the current task can also be regarded as a binary classification task to judge whether bugs exist in a suspicious statement no matter what bug type it is. Thus, a model based on the MLP (Multi-Layer Perceptron) architecture is designed to achieve this goal, which is shown in Figure 4. Before fed into the model, the spectrum-based and mutation-based features are normalized by the ranking positions, because the suspicious scores before normalization are not necessarily in the range $[0, 1)$, while the deep semantic features are. The equation of calculating the normalized score of suspicious statement e is as follows:

$$Sus(e) = 1 - \frac{index(e)}{len(suspicious\_list)} \tag{8}$$

where $suspicious\_list$ is a sorted list containing all suspicious statements in the target project. It is sorted according to the suspicious score before normalization of each statement from biggest to smallest. Function $len(suspicious\_list)$ returns the length of $suspicious\_list$, and function $index(e)$ returns the position where the suspicious score of e last appears in the $suspicious\_list$. For example, if $suspicious\_list$ contains 5 statements $\{A, B, C, D, E\}$, and the corresponding suspicious scores are $\{2.0, 1.0, 0.5, 0.5, 0.2\}$, then the normalized scores of the five statements are calculated as $\{0.8, 0.6, 0.2, 0.2, 0.0\}$. Thus, the three groups of normalized features can now be fed into the model. Since the number of features in deep semantic group exceeds the other two groups a lot, it may have an unexpected greater impact on the results. Thus, we first put the 11 semantic features into an MLP (Semantic Fusion Layer in Figure 4), the output dimension of which is set as 3. Then, the new generated 3-dimension semantic feature is concatenated with other two feature groups to obtain a 10-dimension vector, which is then fed into another MLP (Multi-Source Fusion Layer in Figure 4) for automatic feature extraction. Finally, the data flows to the output layer, and a 2-dimension vector normalized by $softmax$ activation function is generated. The output vector gives the probability of being faulty for the current suspicious statement, which can be regarded as a new suspicious score. A new suspicious list can be generated according to the suspicious scores of all statements.

3.4 TRANSFER-PR: Effective Program Repair based on Transferred Knowledge

3.4.1 Template-based program repair. Template-based automated program repair is widely studied [16, 21, 31, 32], which utilizes fix templates predefined or extracted from similar code snippets to repair specific bugs. The repair process can be basically divided into four steps: fault localization, fix template selection, donor code search, and patch candidate validation, corresponding to step 1 to step 4 in Algorithm 1 respectively. The first 3 steps are closely related to patch generation, while the last step is for validation. Therefore, under the premise that a template-based repair technique has been selected, that is, the set of fix templates to be used has been determined, we can optimize the first 3 steps to tune the repair performance. Step 1 corresponds to our attempt to optimize the effectiveness of fault localization tasks in Section 3.3. For step 2, a naive way is used to traverse all fix templates (e.g., predefined

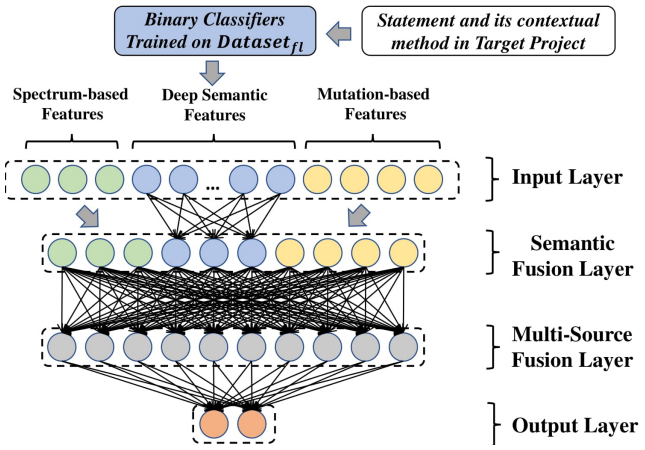

Figure 4: MLP-based Ranking Model for Fault Localization

order or random traversal) in traditional template-based program repair techniques, which is unable to give guidance for which fix template should be selected first to generate patch candidates for a specific input. This leads to a problem during bug fixing. Assuming that the real faulty statement is reached, the repair technique may still generate a plausible but incorrect patch candidate (i.e., a patch can pass all tests but is regarded as a incorrect patch after a manual check) because a wrong fix template is tried earlier. Once a plausible patch is generated, the entire repair process will be terminated according to the commonly-used repair setup, and finally the actual fix template has no chance to be selected. Therefore, we convert the fix template selection problem into a multi-classification task, which will be described in detail below.

3.4.2 Fix template selection with transferred knowledge. In Section 3.2.4, we have obtained the transferred knowledge by training the multi-classifier on large-scale $Dataset_{pr}$. For a target project, we can first extract existing bug-fix commits from its development history, and then use the syntax checker to tag them with the unique labels, just like what we have done to build $Dataset_{pr}$ in Section 3.2. Then, the newly extracted samples can be used to further fine-tune the model parameters of the multi-classifier, which is already trained on $Dataset_{pr}$. In this way, we merge the transferred knowledge learned from $Dataset_{pr}$ with the specific information of the target project. For a faulty statement with its contextual method, the fine-tuned multi-classifier is used to predict which fix templates should be selected first to generate patches, which is expected to optimize the fix template selection task at step 2 in Algorithm 1.

4 EVALUATION

In this section, we conduct the extensive experiments to evaluate the our approach on the fault localization and program repair tasks.

4.1 Benchmark Dataset

We use Defects4J (V1.2.0) benchmark [19] in our experiments, which is widely used for the evaluation of fault localization and automated program repair tasks [16, 25, 26, 28, 32]. This benchmark is composed of 6 open source projects containing 395 real faults.

Algorithm 1 The process of template-based program repair

Input: The project to be repaired P and its test cases T;
Output: The patch candidate c that can pass all test cases;

```
1:  start_time ← get_current_time();
2:  c ← NULL;
3:  suspicious_list ← fault_localization(P, T);
4:  // step 1
5:  for each pos ∈ suspicious_list do
6:    // step 2
7:    for each ft ∈ fix_templates do
8:      donor_code_list ← search_donor_code(pos, ft);
9:      // step 3
10:     for each element ∈ donor_code_list do
11:       candidate ← generate_patch(pos, ft, element);
12:       // step 4
13:       if validate(candidate, T) is True then
14:         c ← candidate;
15:         return c;
16:       end if
17:       current_time ← get_current_time();
18:       if current_time − start_time > TIME_THRESHOLD
          then
19:         return c;
20:       end if
21:     end for
22:   end for
23: end for
24: return c;
```

4.2 Experimental Settings

As shown in Table 5, columns 2 to 5 list the experimental setups for running the BiLSTM-based binary classifiers in Section 3.2.3 ($\boldsymbol{Model_{bi}}$), the MLP-based ranking model in Section 3.3 ($\boldsymbol{Model_{rk}}$), the BiLSTM-based multi-classifier trained on $Dataset_{pr}$ in Section 3.2.4 ($\boldsymbol{Model_{mu}}$), and the BiLSTM-based multi-classifier fine-tuned on Defects4J in Section 3.4 (still $\boldsymbol{Model_{mu}}$), respectively. It should be noted that all hyper-parameters are determined based on the corresponding validation set through grid search. For obtaining spectrum-based and mutation-based features for fault localization task, we utilize GZoltar [6] (V1.7.2) and PIT [8] (V1.1.5) tools. For PIT, we adopt the same modifications following TraPT [26] and DeepFL [25].

When training the MLP-based ranking model, we adopt pairwise approach because the goal of fault localization task is to rank the faulty statements higher than the correct ones, while other relations are not considered. Since most of the suspicious statements are not faulty, the number of positive samples is far less than that of negative samples. Therefore, during training, we adopt down-sampling and randomly select 10 negative samples for each positive sample to generate sample pairs like $<s_{pos}, s_{neg}>$. Then, the hinge function is utilized to calculate the loss of a sample pair. When training the multi-classifier model, due to the imbalance of samples among 11 different bug types on $Dataset_{pr}$, the down-sampling approach is also adopted. We keep up to 10,000 samples for each

Table 5: Setups for Experiments

Setups	$Model_{bi}$	$Model_{rk}$	$Model_{mu}$	$Model_{mu}$
Dataset	$Dataset_{fl}$	Defects4J	$Dataset_{pr}$	Defects4J
Batch Size	64	64	64	8
Epochs	30	30	15	30
Loss Function	CE	Hinge-loss	CE	CE
Input Dim	400	18(11+3+4)	400	400
Hidden Units	50	-	80	80
Optimizer	Adam	Adam	Adam	Adam
Learning Rate	1e-3	1e-3	1e-3	1e-4
Dropout Rate	-	0.3	0.3	-
λ for L2-Reg	-	1e-4	1e-4	-

bug type. Finally, we use 10-fold cross validation on the fault localization (column 3) and program repair tasks (column 5) to obtain the experimental results. We set the running time of each repair process to 3 hours, which is the same as TBar [32].

All the experiments are conducted on Ubuntu 18.04 server with 20 cores of 2.4GHz CPU, 384GB RAM and NVIDIA Tesla V100 GPUs with 32 GB memory.

4.3 Evaluation Metrics

We adopt the following common evaluation metrics used in the previous fault localization studies [25, 26, 28, 51]:

Top-N, which represents the number of faults with at least one faulty statement located in the top N positions. Following previous researches [25, 26, 28], Top-1, Top-3, and Top-5 are reported.

Mean First Rank (MFR): If there are multiple faulty statements in a fault, localizing the first one is important. The MFR metric of one project is the mean of the highest faulty statement's rank of each fault.

Mean Average Rank (MAR), which is computed as the average rank of all faulty statements for each fault, and the MAR metric of one project is the mean of the average rank of all its faults.

Among them, the bigger is better for Top-1/3/5 while the smaller is better for MFR and MAR. For the program repair task, we use the number of fixed faults to measure the effectiveness of a program repair technique.

4.4 Results and Discussion

RQ1: How does TRANSFER-FL perform in statement-level fault localization?

To answer this research question, we compare the effectiveness of TRANSFER-FL with three SBFL techniques (i.e., Ochiai [1], Tarantula [18] and DStar [57]) whose suspicious scores are used to form the spectrum-based feature group in this paper, and one MBFL technique (i.e., Metallaxis [44]) for mutation-based feature group, and two deep learning-based techniques (i.e., DeepFL [25] and DeepRL4FL [28]). Note that the experimental results of DeepFL are obtained after we make simple modifications (i.e., only semantic-based and mutation-based features are kept) based on its open source repository to meet the needs of statement-level fault localization tasks, which is originally designed for method-level. For DeepRL4FL, the state-of-the-art technique, since the open source repository and relevant dataset are not publicly available, we are not able to reproduce its results. Thus, we directly cite the experimental results reported in their paper [28][1].

Table 6: Fault Localization Results

Techniques	Top-1	Top-3	Top-5	MFR	MAR
Ochiai [1]	19	65	99	183.78	233.14
Tarantula [18]	19	63	97	189.28	241.58
Dstar [57]	20	65	99	183.75	233.52
Metallaxis [44]	13	36	63	512.28	649.41
DeepFL [25]	60	122	140	128.02	170.46
DeepRL4FL [28]	71	128	142	-	-
TRANSFER-FL	**84**	**144**	**171**	**79.97**	**120.47**

Table 6 presents the detailed experimental results. From the table, we can see that TRANSFER-FL can perform all compared techniques in all metrics. More specifically, compared with the three SBFL techniques, the increased numbers of the faults hit on Top-1,3,5 are 64+, 79+ and 72+, while the improvements on MFR and MAR are 56.5% and 48.3%. When compared with the MBFL technique Metallaxis, the increased numbers on Top-1, 3, 5 are 71, 108 and 108, while the improvements on MFR and MAR are 84.4% and 81.4%. SBFL and MBFL methods only consider the dynamic information generated by executing test cases, and lack the analyses for program semantics. When compared with the two deep learning-based methods, the increased numbers on Top-1,3,5 are 13+, 16+ and 29+, while the improvements on MFR and MAR are 37.5% and 29.3%, showing that TRANSFER-FL significantly outperforms the state-of-the-art deep learning-based methods. Since semantic features are not used or just simply used by concatenating the word embeddings in the existing deep learning-based methods, the transferred knowledge learned from our built large-scale dataset and the deep semantic features generated by 11 binary classifiers can significantly enhance the effectiveness of fault localization.

RQ2: How effective are the main components of TRANSFER-FL?

In this RQ, we explore the effectiveness of main components in TRANSFER-FL. Since TRANSFER-FL is based on the fusion of 3 feature groups, we design 6 model variants, each of which represents a specific combination of features groups. As shown in Table 7, the first 3 variants retain only one feature group, and the following 2 variants retain the combinations {spectrum, semantic} and {mutation, semantic}, respectively. The last variant retains all 3 feature groups (i.e., TRANSFER). Through the experimental results, we find that under the restriction of using only one feature group, $variant_{mutation}$ performs best on all 5 metrics, and $variant_{semantic}$ performs better than $varaint_{spectrum}$ on Top-1, while the results are opposite on other 4 metrics. Next, compared with $variant_{spectrum}$ ($variant_{mutation}$), $variant_{spec+sem}$ ($variant_{mut+sem}$) achieves significant improvement, indicating that the semantic features can be effectively integrated with existing traditional features to generate richer information, which is beneficial for fault localization. Finally, the experimental results of $variant_{all}$ on all 5 evaluation metrics

[1]Note that we find that the MFR and MAR results reported in DeepRL4FL [28] are strangely very low (20.32 and 28.63, respectively). After checking the code and consulting the authors of DeepRL4FL, we have to choose not to compare these two metrics with DeepRL4FL.

Table 7: Comparative analysis with different feature groups

Techniques	Top-1	Top-3	Top-5	MFR	MAR
$variant_{spectrum}$	20	63	98	180.53	232.32
$variant_{semantic}$	25	51	71	253.27	350.03
$variant_{mutation}$	55	109	138	110.10	173.96
$variant_{spec+sem}$	49	91	120	118.80	174.64
$variant_{mut+sem}$	70	121	152	93.98	148.41
$\boldsymbol{variant_{all}}$	**84**	**144**	**171**	**79.97**	**120.47**

Table 8: Repair Results under Perfect Fault Localization

Project	DLFix	CoCoNuT	CURE	TBar	TRANSFER-PR
Chart	5	7	**10**	**10**	**10**
Closure	11	9	14	16	**18**
Lang	8	7	9	10	**13**
Math	13	16	19	**20**	**20**
Time	2	1	1	**3**	**3**
Mockito	1	**4**	3	3	3
Total	40	44	56	62	**67**

are better than those of all other variants, proving that each feature group has a positive impact on the fault localization task.

RQ3: How does TRANSFER-PR perform in program repair under perfect fault localization?

In order to evaluate whether TRANSFER-PR is effective or not, we conduct experiments under perfect fault localization setup (i.e., the actual faulty statements are given). We then compare the repair results of different methods. As shown in Table 8, DLFix [27], CoCoNuT [39], CURE [17] are encoder-decoder based deep learning methods, while TBar [32] is the state-of-the-art template-based method. TRANSFER-PR is implemented based on TBar, because the predefined fix templates used in this paper are derived from it. The experimental results show that TRANSFER-PR can fix 5 more bugs (67 in total) than TBar, which reflects the improvement gained from the optimized fix template selection mechanism. When compared with deep learning-based repair methods, TRANSFER-PR can fix 11 more bugs than the state-of-the-art method CURE. Furthermore, TRANSFER-PR achieves the best results on 5 out 6 projects. The results show that using predefined high-quality fix templates and optimizing the selection of fix templates can improve the repair performance.

RQ4: Can TRANSFER, as a whole, improve the performance of program repair?

In this RQ, we evaluate the effectiveness of TRANSFER (both FL and PR) in automated program repair. As TRANSFER-PR is a template-based approach derived from TBar [32], and TBar is the best-performing repair method as shown in Table 8, we use TBar as the baseline in this RQ. As described in TBar [32], there are 71 bug versions that can be fixed when the faulty statements and the fix templates are both directly given. However, in actual repair scenario, these two premises are not satisfied. Thus, we preform a set of comparison experiments to explore how TRANSFER-FL and TRANSFER-PR improve program repair in actual repair scenario (both faulty statements and fix templates are unknown). Table 9 shows the repair results in different {FL Approach × PR Approach} setups. We find that the number of fixed bugs under {Ochiai, TBar} setup (which is the default setup in TBar [32]) is 41, while the number under {Ochiai, TRANSFER-PR} setup is 42, showing the effectiveness of TRANSFER-PR method. In addition, the number of fixed bugs under {TRANSFER-FL, TBar} steup is 43, indicating that TRANSFER-FL can also help improve the repair performance. Then, when both TRANSFER-FL and TRANSFER-PR are used (i.e., TRANSFER), there are 47 bugs can be fixed, which achieves the best result. The detailed statistics of the fixed bugs corresponding to 4 setups are shown in Figure 5-(a). Figure 5-(b) shows the bugs which can be fixed in all 4 setups.

Table 9: Repair Experiments in Different Localization and Repair Setups

Localization Approaches	Repair Approaches	#Not-fixed Bugs[1]			#Fixed Bugs
		Time	Pos.	Pat.	
Ochiai	TBar	18	9	3	41
	TRANSFER-PR	18	9	2	42
TRANSFER-FL	TBar	10	12	6	43
	TRANSFER-PR	10	12	2	47

[1] Reasons for not-fixed bugs: Time (Timeout), Pos (Position) and Pat (Pattern).

Table 9 also shows the number of not-fixed bugs and the corresponding reasons. We divide the reasons why a bug cannot be successfully fixed into three categories: 1) Timeout: the faulty statements are ranked too low in suspicious list to be found within a limited time. 2) Position: a plausible but incorrect patch is generated in another suspicious statement, when the real faulty statement has not been reached. 3) Pattern: a plausible but incorrect patch is generated due to a wrong fix template being selected. Table 9 also shows that, after adopting TRANSFER-FL, the number of not-fixed bugs with the reason of Timeout is significantly reduced (from 18 to 10). This result shows that many bugs that previously failed to be located within a limited time have now been successfully found. After adopting TRANSFER, there are still 24 not-fixed bugs, caused by Timeout (10), Position (12), and Pattern (2). In our future work, we will address these not-fixed bugs and further improve the repair performance.

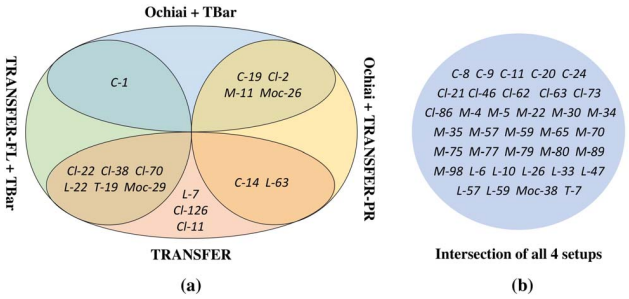

Figure 5: Overlapping Analysis for Program Repair Experiment (C:Chart, Cl:Closure, L:Lang, M:Math, Moc:Mockito, T:Time)

5 DISCUSSION

5.1 Why Does it Work?

For the fault localization task, previous spectrum-based, mutation-based, and deep learning-based methods either do not consider deep semantic features, or just use some simple features by concatenating the word embeddings, while our approach incorporates the deep semantic features and transferred knowledge from the large-scale open-source bug dataset. Specifically, we design 11 binary classifiers to extract deep semantic features for predicting the probabilities of containing bugs of different bug types.

For the program repair task, selecting the correct fix template for the statement to be repaired is important, because if an incorrect fix template is selected, a plausible but incorrect patch may be generated and the whole fix process will be terminated. Our approach includes a multi-classifier, which learns deep semantic features from historical data that contains knowledge about which fix template should be selected. In this way, the number of plausible but incorrect patches decreases and the repair performance is improved.

5.2 Threats to Validity

One threat to external validity is the target programming language we use, i.e., the selected fix templates and generated datasets are all for Java language. However, most of the fix templates can be generalized to other languages because of the generic representation of AST. On the other hand, there are sufficient projects in open source communities (e.g., GitHub) to build datasets for other languages. The second threat to external validity is that we use one defect benchmark (Defects4J-V1.2.0) in our study. Although it is a widely-used benchmark, Durieux et al. [10] showed that fault localization and program repair techniques may overfit on this benchmark. To reduce this threat, we have conducted a preliminary experiment on a recent benchmark Defects4J-V2.0.0. The results (given in our project page) show similar trend as that on Defects4J-V1.2.0 and confirm the effectiveness of the proposed approach. In our future work, we will conduct more comprehensive experiments to further evaluate the generality of our methods on more defect benchmarks.

One threat to internal validity is the implementation of the syntax analyzer developed by ourselves. In order to reduce the threat, the analyzer is developed based on the widely used JDT framework [11]. Another internal threat is the manual annotation for faulty statements in Defects4J, because the standards to judge whether a statement is faulty or not may be different. To reduce this threat, we choose the standard given by the authors of Defects4J [45].

6 CONCLUSION

In this paper, we propose a fault localization method (TRANSFER-FL) that incorporates the deep semantic-based features extracted by learning the transferred knowledge from large-scale open-source data. We also propose a program repair method (TRANSFER-PR) for optimizing the selection of fix templates, which can be used together with the fault localization method to improve the performance of existing template-based program repair techniques. Our approach TRANSFER (the combination of FL and PR) can fix 47 bugs on Defects4J dataset, which is 6 more than that of the state-of-the-art template-based repair technique TBar [32].

Our source code and experimental data are publicly available at: **https://github.com/mxx1219/TRANSFER**. The code and data can facilitate replication of our study. Furthermore, the large-scale datasets $Dataset_{fl}$ and $Dataset_{pr}$ built by us can be utilized in future automatic software debugging research.

ACKNOWLEDGMENTS

This work was supported partly by National Key Research and Development Program of China (No.2018YFB1306000), partly by National Natural Science Foundation of China (No. 62072017, 62141209) and Australian Research Council Discovery Projects (DP200102940, DP220103044), and the Ministry of Industry and Information Technology of the PRC.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.
[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.
[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[5] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 177–188.
[6] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 378–381.
[7] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. Codit: Code editing with tree-based neural machine translation. *arXiv preprint arXiv:1810.00314* (2018).
[8] Henry Coles. 2021. PIT. https://pitest.org/
[9] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 349–358.
[10] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 302–313.
[11] Eclipse. 2021. JDT. https://www.eclipse.org/jdt/
[12] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.
[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[14] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th international conference on software engineering*. 12–23.
[15] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2020. Control flow graph embedding based on multi-instance decomposition for bug localization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4223–4230.
[16] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.
[17] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.
[18] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.
[19] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[20] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[21] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* (2020), 1–45.

[22] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[23] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[24] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2016. Iterative user-driven fault localization. In *Haifa Verification Conference*. Springer, 82–98.

[25] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 169–180.

[26] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[27] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[28] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.

[29] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[30] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *Acm Sigplan Notices* 40, 6 (2005), 15–26.

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[32] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[33] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 658–662.

[34] Tie-Yan Liu. 2011. Learning to rank for information retrieval. (2011).

[35] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 118–129.

[36] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.

[37] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[38] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[40] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[41] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[42] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[43] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[44] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[45] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[46] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002).

[47] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[48] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

[49] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[50] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.

[51] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[52] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 832–837.

[53] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308.

[54] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[55] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1–11.

[56] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[57] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 21–30.

[58] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670.

[59] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 191–200.

[60] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[61] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 23–32.

[62] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Notices* 48, 10 (2013), 765–784.

[63] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2007. Locating faulty code by multiple points slicing. *Software: Practice and Experience* 37, 9 (2007), 935–961.

[64] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. 2005. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. 33–42.

[65] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.

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

[67] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.