Multi-View Feature Fusion Model for Software Bug Repair Pattern Prediction

XU Yong¹, CHENG Ming²

1. School of Mathematics and Statistics, Zhaoqing University, Zhaoqing 526040, Guangdong, China;
2. Department of Medical Information, The First Affiliated Hospital of Zhengzhou University, Zhengzhou 450000, Henan, China

© Wuhan University 2023

Abstract: Many search-based Automatic Program Repair (APR) techniques employ a set of repair patterns to generate candidate patches. Regarding repair pattern selection, existing search-based APR techniques either randomly select a repair pattern from the repair pattern set to apply or prioritize all repair patterns based on the bug’s context information. In this paper, we introduce PatternNet, a multi-view feature fusion model capable of predicting the repair pattern for a reported software bug. To accomplish this task, PatternNet first extracts multi-view features from the pair of buggy code and bug report using different models. Specifically, a transformer-based model (i.e., UniXcoder) is utilized to obtain the bimodal feature representation of the buggy code and bug report. Additionally, an Abstract Syntax Tree (AST)-based neural model (i.e., ASTNN) is employed to learn the feature representation of the buggy code. Second, a co-attention mechanism is adopted to capture the dependencies between the statement trees in the AST of the buggy code and the textual tokens of the reported bug, resulting in co-attentive features between statement trees and reported bug’s textual tokens. Finally, these multi-view features are combined into a unified representation using a feature fusion network. We quantitatively demonstrate the effectiveness of PatternNet and the feature fusion network for predicting software bug repair patterns.

Key words: Automatic Program Repair (APR); bug repair pattern prediction; Recurrent Neural Network (RNN); transformer; co-attention

CLC number: TP391

0 Introduction

It is widely recognized that software bugs are inevitable and pervasive. Bug fixing constitutes a significant aspect of computer programming and has become a routine activity for software developers. However, manual bug fixing is a laborious, error-prone, and costly task, accounting for up to 90% of the total cost of a software project. Consequently, to assist developers in debugging programs more effectively and efficiently, Automatic Program Repair (APR) techniques have garnered considerable attention from industry practitioners and academic researchers, making noteworthy contributions to the field.

For example, the generate-and-validate and program synthesis-based approaches are two well-known design philosophies followed by many APR techniques (e.g., Refs. [2-5]). The basic idea behind the first approach is to generate a candidate patch space using predefined strategies and then validate all repair candidates...
(e.g., Refs. [2-4]), aiming to find one that can make the buggy programs behave as expected. On the other hand, the latter approach utilizes a program synthesis engine to synthesize patches that adhere to the given specification (e.g., Ref. [5]). Despite achieving promising results, existing APR techniques suffer from limitations such as the search space explosion problem and the generation of low-quality patches\(^6\), which hinders their practical application.

To reduce the search space and enhance the quality of generated patches, some APR techniques utilize fix templates to generate patches\(^5\). Fix templates are common repair patterns, such as modifying a branch condition or wrapping code with an "if" statement, and can be either manually crafted or automatically inferred from human-written patches collected from open source repositories\(^8\). Some researchers also explored the mining and extraction of repair models from software repositories\(^{16-19}\). For example, in Ref. [10], the extracted repair models consist of probabilistic repair actions, which are modifications or mutators related to bug fixes. A subset of these probabilistic repair actions includes the probabilities of inserting assignments (4.1%) and "if" statements (6.6%). Well-designed fix templates empower APR techniques, such as Genesis\(^9\), to generate repairs tailored to specific types of bugs, such as null pointer and class cast (CC) defects. Additionally, APR techniques can efficiently traverse the search space using repair models to prioritize potentially correct patches among all generated candidate patches\(^4\).

While the literature has demonstrated the effectiveness of applying repair patterns in APR, there is limited work on automatically annotating a bug with the most relevant repair patterns. In this paper, we aim to investigate the feasibility of training a machine learning model to identify a bug’s repair pattern automatically. This is achieved through the training of a single model, PatternNet, for bug repair pattern prediction.

Training PatternNet, however, is far from trivial. First, apart from the syntax and semantic information in buggy code, bug reports also contain valuable data that can aid in predicting a bug’s repair patterns. For illustration, consider a real BIRT bug \#293006, as listed in Listing 1 (Fig. 1), taken from the BIRT project. Evidently, the code will crash if the argument "parent" is NULL. Repairing the bug involves applying a single pattern, i.e., adding a null check to the variable "parent". Simultaneously, its bug report recorded in a bug tracking system is presented in Listing 2 (Fig. 1). The bug report’s summary and textual description provide information regarding the code crash. Specifically, the exception occurs when the variable ‘parent’ is NULL. This example demonstrates that, in addition to the faulty source code, bug report texts often offer valuable insights, which can assist in predicting repair patterns. Therefore, to achieve superior performance, PatternNet should possess the capability to process multimodal information. We further contend that such multimodal information could prove advantageous in the bug repair pattern prediction model.

One challenge of training PatternNet is how to extract features from code and bug reports. Recently, several deep learning approaches have been proposed, aiming to learn source code embeddings. Among the existing techniques for source code representation learning, transformer-based pre-trained models\(^{14,15}\) and Recurrent Neural Networks (RNN)-based approaches\(^{16}\) represent state of the art. For example, UniXcoder\(^{14}\) is based on a multi-layer transformer, utilizing multi-modal content including the abstract syntax tree of source code and comments, to learn the representation of code fragments through contrastive learning. While this approach is appealing, it is important to note that the model’s maximum input token sequence length is 512, which necessitates truncation for longer input sequences, potentially leading to the loss of information from the source code. Therefore, the model’s performance may degrade when handling datasets with source code token lengths exceeding 512. ASTNN\(^{16}\), another method for source code representation, splits the abstract syntax tree of source code into a sequence of statement trees. A Recursive Neural

---

**Listing 1: A Bug**

```
public void setParent(IEElement parent) {
    super.setParent(parent);
    // if (parent != null) {
    //    added if
    while (((IContent) parent).setHasChildren(true));
    //    added block
}
```

**Listing 2: Bug Report**

```
Summary: Bug 293006 setParent() should exclude the condition when parent is null.
Description: setParent() should exclude the condition when parent is null. Unit test failed.
if (parent != null) {
    ((IContent) parent) .setHasChildren(true); }
```

---

![Fig. 1] Example of bug and its bug report taken from commit be373ea of BIRT project and issue tracking system
Network (RvNN) encodes each statement tree into a vector. A bidirectional Gated Recurrent Unit (Bi-GRU) is applied to iterate through each statement tree vector to obtain the vector representation of an entire code fragment. Since RNNs are designed to handle input sequences of variable length, the ASTNN model can effectively manage long sequences of code statements. However, it exclusively considers the purely unimodal content, i.e., source code, for learning its representation. In order to further enhance the effectiveness of code representation learning, this paper proposes a hybrid code representation learning for repair pattern prediction, drawing inspiration from recent success in multi-view deep learning\(^{17}\). Specifically, PatternNet takes the buggy source code and its bug report as input, performing feature extraction from two different modalities: a transformer-based model and a RNN-based model. The pre-trained transformer-based models (i.e., UniXcoder) are used to extract features from the buggy code-bug report pair, while the RNN-based model (i.e., ASTNN) is employed to obtain features from the abstract syntax tree of the buggy code. Finally, the feature vectors from both views are combined before being utilized in the repair pattern prediction.

Another challenge for training PatternNet lies in the lack of a commonly used standard dataset for bug repair pattern prediction. Creating a comprehensive dataset of all types of bug repair patterns is difficult, since the numbers and types of repair patterns used during bug fixing vary widely across different bugs. Furthermore, the probability distribution of change actions occurring is very imbalanced\(^{19}\). Thus, we restrict our attention to those bugs that require the application of single frequent repair pattern to fix. For brevity, these bugs are called single repair pattern bugs, otherwise, that is, bugs that fixed by applying more than one repair pattern, are referred to as multi-repair pattern bugs. In fact, as our empirical analysis shows, most bugs in Defects4J\(^{10}\) fixed successfully by state-of-art pattern-based APR belong to single repair pattern bugs (Section 4).

The contributions of this paper are as follows: first, we introduce PatternNet, a model for automatically predicting a reported bug’s repair pattern. To the best of our knowledge, PatternNet is the first model that predicts a bug’s repair pattern based on the semantic features learned from bug reports and source code, demonstrating that a reported bug’s repair pattern can be predicted rather than mined\(^7\) or inferred\(^{19}\). Second, a multi-view features learning and co-attention feature fusion model are developed in PatternNet to get the high-level representation of buggy code and bug report. Multi-view features learning enables PatternNet to both reap the benefit of different source code representation learning models described above and alleviate the drawback of single view feature learning models. Furthermore, the co-attention feature fusion model allows PatternNet to attend to different statements of buggy code and different tokens of reported bug. Third, we conduct experiments on our newly created dataset for bug repair pattern prediction, and the experimental results quantitatively demonstrate the effectiveness of PatternNet and feature fusion network for software bug repair pattern prediction.

1 Related Work

1.1 Automated Program Repair

A rich body of work has been done on developing automated methods of bug fixing since automated program repair promises to reduce the burden of software debugging on developers. According to a recent work\(^{19}\), these techniques can be divided into three categories: generate-and-validate methodology\(^{4,7,9}\), constraint-based repair\(^{20,21}\) and deep learning-based repair\(^{22-24}\). In order to reduce the repair search space and improve the generated patches’ quality, bug repair patterns have been widely adopted in these approaches. The adopted repair patterns can be a set of patterns manually pre-defined\(^{25}\), automatically mined\(^7\) or inferred\(^{19}\) from the past bug fixes. Compared to previous work on APR techniques, our work uses a learned prediction model to provide general guidelines for selecting the most appropriate bug repair patterns required in bug fixing. Therefore, our work is complementary to previous studies on template-based APR techniques. The most closely related to our work is M3V\(^{26}\). M3V is an APR tool based on repair operator prediction. Our work differs from M3V in that M3V only can predict repair operators for bugs caused by null pointer exceptions and index out bounds, PatternNet, however, can predict 25 classes of repair patterns.

1.2 Bug Fixing Mining and Analysis

To obtain an in-depth understanding of how real world bugs are fixed, a large number of empirical studies have been conducted\(^{10,13,27-29}\). Specifically, the mining and analyses are carried out based on the data collected either directly from bug datasets\(^{27}\) or from bug
fixing commits of project repositories\textsuperscript{10-12}. The findings of these empirical studies mainly contain the common bug repair patterns identified among the human-written patches and the distribution of these patterns. Obviously, our work follows this line of research, but goes further and presents a bug repair pattern prediction model.

1.3 Source Code Representation Learning

As for source code representation, one kind of deep learning models consists of several pre-trained transformer-based models such as CodeBERT\textsuperscript{15}, GraphCodeBERT\textsuperscript{16} and UniXcoder\textsuperscript{14}. UniXcoder has been presented recently representing current state of the art techniques for code understanding and generation. Though these models have been shown the powerful source code representation, long code snippets are hard to handle\textsuperscript{31}. ASTNN\textsuperscript{18} is another kind of source code representation based on deep RNN. The key feature of ASTNN is that it can encode long code since RNN is used to model the sequence of statements of code. Our work combines these two types of source code representation approaches and demonstrates the benefits of the combination.

2 Background

2.1 Multi-View Deep Learning

Deep multi-view learning is a deep feature representation learning technique by exploiting complementary information of multiple features or modalities\textsuperscript{17}. To combine multiple distinct features and utilize the complementary information of each view, several methods have been proposed, such as Canonical Correlation Analysis (CCA)\textsuperscript{12}, co-attention mechanism\textsuperscript{13}. The former intends to discover the common information that is mostly correlated between views, however it may ignore the complementary information in the data, which yields sub-optimal quality of the representation resulting in poor model performance\textsuperscript{13}. The latter tries to simultaneously attend multi-view information by weighing the importance of multi-view features for improving representation learning. In fact, co-attention mechanism can be considered a feature fusion method, since it is designed to obtain a better representation of paired views\textsuperscript{14}. Moreover, co-attention mechanism has been successfully applied in many fields such as computer vision\textsuperscript{12} and software code search\textsuperscript{15}, and has achieved remarkable performance in multi-view feature representation.

2.2 Tree-Based Neural Network for Source Code Encoder

Source code representation learning has emerged as a fundamental technique supporting various downstream software engineering tasks, including clone detection and code translation. Predicting software bug repair patterns is no exception. Notably, source code exhibits formal syntax and semantics, in contrast to natural language. Moreover, the syntactic usage of code elements is highly contextual\textsuperscript{13}. Intuitively, capturing the source code’s structural and contextual information as comprehensively as possible can result in superior source code representation, a notion supported by recent finding\textsuperscript{16}. As a result, one approach to source code representation focuses on modeling code at the AST level. The AST is a recursive tree model representing an entire program or a specific program structure (e.g., statement or expression), encompassing lexical information and the syntactic structure of the source code. Given the inherently recursive nature of the AST, several recursive source code encoders have been introduced, including recursive Neural Networks\textsuperscript{17}, Tree-LSTM\textsuperscript{18}, and ASTNN\textsuperscript{16}. These source code encoders recursively compute node embeddings within the AST in a bottom-up manner, and the vector associated with the root node of the AST is employed to represent the source code. ASTNN represents the state-of-the-art in this regard, as it divides each large AST into a sequence of smaller statement trees and encodes these statement trees into vectors. Building on this, ASTNN further captures the sequential information among statement trees using a bidirectional RNN model, ultimately producing a vector representation of a code fragment.

2.3 Transformer Model for Source Code Embedding

Inspired by the success of BERT\textsuperscript{19} in natural language processing, the transformer’s transfer learning strategy has recently been extended to multi-modal learning for source code understanding and generation. Among the most notable transformer-based models are CodeBERT\textsuperscript{15}, GraphCodeBERT\textsuperscript{16}, and UniXcoder\textsuperscript{14}. These models have been developed using a pre-training and fine-tuning workflow. Specifically, CodeBERT, a bi-modal pre-trained language model for both natural and programming languages, is trained with two objectives: Masked Language Modeling (MLM) and Replaced Token Detection (RTD).

In contrast to CodeBERT, GraphCodeBERT takes
source code paired with comments and the corresponding data flow as input. It is pre-trained using standard masked language modeling [39] and two structure-aware tasks. One structure-aware task involves predicting where a variable is identified, while the other focuses on predicting data flow edges between variables.

To support various types of software engineering downstream tasks, UniXcoder, however, is pre-trained on large amounts of multi-modal content, including code, code comments, and AST, using self-supervised objectives such as masked language modeling, unidirectional language modeling [40], and denoising objectives. In addition to these self-supervised objectives, UniXcoder also employs two supervised pre-training tasks: multi-modal contrastive learning and cross-modal generation to learn embeddings that represent the semantics of a code fragment.

3 PatternNet: Multi-View Feature Fusion Model for Repair Pattern Prediction

3.1 Overall Architecture

In this section, we introduce the deep neural network model PatternNet for repair pattern prediction, as shown in Fig. 2. This model comprises three components: the top branch captures features from the semantics of the buggy code’s AST. The bottom branch is a transformer-based model that simultaneously represents the tokens of the buggy code and the bug report. The outputs of these two subnetworks are combined to form a single latent representation using a feature fusion network. Finally, the resulting features are sent to the softmax layer for classification. We provide detailed descriptions of the various subnetworks in the following sections.

3.2 Representation Learning for Buggy Code AST

Splitting ASTs and Creating Sequences of Statement Trees. In general, AST is a tree data structure in which each node represents an abstract syntactic structure of the source code. As mentioned earlier, ASTNN creates a sequence of statement trees from a code fragment at the statement level. An AST node is referred to as a statement node if it represents a statement in the source code. Essentially, a statement tree is a rooted tree in which one node is designated as the tree’s root, while the remaining nodes are the root’s descendants. Given a code fragment, it is first parsed to build an AST Z. Then, a statement tree sequence is created by using a pre-order traversal of Z, with the ordering determined by the sequence in which the statement nodes are visited. Let s be the statement node currently visited during the AST traversal. If s corresponds to a simple statement, a statement tree can be directly constructed from the subtree rooted at s in Z. In other words, the statement tree, whose root is s, consists of all children of s. On the other hand, if s corresponds to a nested statement, the node s is split into two parts: the header and the body of nested statements. For the header, constructing a statement tree rooted at s is straightforward. Additionally, a recursive process of splitting AST nodes and constructing statement trees is performed for each statement node inside the body of nested statements. Finally, all the constructed statement trees are sequentially added to the statement tree sequence. For more details, we refer the reader to Ref. [16].

Statement Tree Encoder. Since a statement tree is a recursive data structure consisting of AST nodes, ASTNN employs a recursive neural network-based statement tree encoder to learn a representation of statements. On the other hand, ASTs contain a variety of symbols representing either syntactic constructs or lexical information, such as identifiers. Hence, ASTNN utilizes word2vec [42] to learn vector representations of symbols. The trained symbol embeddings are then used to initialize the parameters in the encoder. Specifically, suppose we have a statement tree t, and it has C children. Let $W_s \in \mathbb{R}^{V \times d}$ denote the pre-trained embedding parameters, where V is the vocabulary size and d is the embedding dimension of symbols. Let n be an internal node. The lexical vector representation of node n can be computed by:

$$v_n = W_s x_n$$

(1)

where $x_n, v_n$ is the one-hot representation of the symbol n and the embedding, respectively. Next, the node n’s vector representation can be obtained by the following equation:

$$h = \sigma \left( W_n^\top v_n + \sum_{i \in C} h_i + b_s \right)$$

(2)

where $W_n \in \mathbb{R}^{d \times u}$ is the weight matrix, and the dimension size is u, $b_s$ is a biased term, $h_i$ is the hidden state for each child i, $h$ denotes the updated hidden state, and $\sigma$ is the identity activation function. In a similar fashion, the vectors of all nodes in Z can be recursively com-
computed and optimized in a bottom-up way. Additionally, the encoder applies the max pooling operation on each node to extract the most significant features. Therefore, the final representation of the statement tree \( i \) with \( N \) nodes can be obtained by:

\[
e_i = \left[ \max(h_1), \ldots, \max(h_n) \right], i = 1, \ldots, N \tag{3}
\]

Note that since different nodes have a different number of child nodes, directly calculating equation (2) in a batch manner is impossible. Therefore, a dynamic batch processing algorithm is presented to facilitate batch computations of equation (2). For more details about the algorithm, please refer to Ref. [16].

Representing the Sequence of Statement Trees. Suppose we have a code fragment; a sequence of \( T \) statement trees is extracted from its AST. After the statement tree encoder described above is applied to all statement trees, the code fragment can be represented in a sequence of statement tree vectors \( p = [e_1, \ldots, e_i, \ldots, e_T] \). To gain some dependency between adjacent statement trees within the code fragment, ASTNN uses a Bi-GRU network to process the sequence of statement tree vectors \( p \). Specifically, the Bi-GRU contains a forward GRU which reads \( p \) from \( e_i \) to \( e_j \), and a backward GRU which reads from \( e_j \) to \( e_i \):

\[
\begin{align*}
\vec{h}_i &= \text{GRU} \left( e_j \right), \tau \in \{1, T\} \\
\vec{h}_i &= \text{GRU} \left( e_i \right), \tau \in \{1, T\}
\end{align*}
\]

At time step \( \tau \), the forward hidden state \( \vec{h}_\tau \) and backward hidden state \( \vec{h}_\tau \) are concatenated to form the annotation of statement tree \( \tau \), i.e., \( v_\tau = [\vec{h}_\tau, \vec{h}_\tau] \), which summarizes the context information surrounding the statement tree \( \tau \). After iteratively calculating hidden states of all time steps, all hidden states are denoted as \( V = \{v_1, v_2, \ldots, v_T\} \). Following ASTNN, the max pooling operation over \( v_\tau \) is applied, producing a vector \( \tilde{v} \), which is the vector representation of the buggy code.

3.3 Representation Learning for Buggy Code-Bug Report Pair

To capture the semantics of both the buggy code and the bug report, PatternNet leverages UniXcoder to obtain the features representing buggy code and bug report. In particular, given a pair of buggy code and bug report, they are tokenized separately into \( m \) code tokens and \( n \) text tokens using RobertaTokenizer, which are denoted as \( [\text{tok}_1^c, \text{tok}_2^c, \ldots, \text{tok}_m^c] \) and \( [\text{tok}_1^t, \text{tok}_2^t, \ldots, \text{tok}_n^t] \), respectively. Since UniXcoder takes a sequence of tokens as input, these two sequences of tokens need to be concatenated into one sequence. Therefore, we add the unique token [SEP] in the middle of two sequences and the unique token [CLS] and [EOS] at the start and the end of the whole sequence, respectively. The embedding of each token is obtained by feeding the concatenated token sequence to UniXcoder, as shown in Fig. 3. These token embeddings are denoted as \( Q = [q_1^c, q_2^c, q_3^c, \ldots, q_n^t, q_1^t, q_2^t, \ldots, q_m^c] \).
where $W_s \in \mathbb{R}^{d_v \times d}$ are learnable parameters. The element $c_y$ of $C$ represents the similarity between the $i$-th bimodal token of $Q$ and the $j$-th statement tree of $V$.

Then the attention weights of each statement tree and bimodal token denoted by $a' \in \mathbb{R}^T$ and $a'' \in \mathbb{R}^N$ are calculated as follows:

\[
\begin{align*}
H' &= \tanh(W_s V + (W_s, Q)c) \\
H'' &= \tanh(W_s Q + (W_s, V)c') \\
a' &= \text{softmax}(w^a H') \\
a'' &= \text{softmax}(w'' H'')
\end{align*}
\]  

where $W_s \in \mathbb{R}^{d_v \times d}$, $W_e \in \mathbb{R}^{d_v \times d}$, $w_a$, and $w''$ are the learnable weight parameters.

In what follows, the statement tree and bimodal token attention vectors are calculated as the weighted sum of statement tree features and bimodal token features as follows,

\[
\hat{v} = \sum_{i=1}^{T} a'_i v_i, \quad \hat{q} = \sum_{s=1}^{N} a''_s q_s
\]  

Finally, inspired by the work in Ref. [44], we use a multi-layer perceptron to encode the multi-view features, that is,

\[
\begin{align*}
h' &= \tanh(W_s (\hat{v} + \hat{q})) \\
h'' &= \tanh(W_e (\hat{v}, h')) \\
e &= \tanh(W_e [\hat{q}, h''])
\end{align*}
\]

where $W_s$, $W_e$, and $W_e$ are the learnable weight parameters. $[\cdot]$ denotes the concatenation operation.

3.5 Bug Repair Pattern Prediction

As said before, this paper aims to develop a model that can automatically perform bug repair pattern prediction. We cast the prediction problem as a multi-class classification problem. Thus, given the fused feature vector $e$ of the $i$-th instance, i.e., a pair of buggy code and its bug report, the probability $\hat{p}_a$ that the $i$-th instance belongs to the repair pattern $a$ can be obtained as follows:

\[
\hat{p}_a = \text{softmax}(W_s e + b_a)
\]

where $W_s$ and $b_a$ denote the weight matrix and bias term.

In our model, we use cross entropy as the loss function, which is defined as:

\[
L = -\sum_{i=1}^{M} \sum_{a=1}^{D} p'_a \log \hat{p}_a
\]  

where $M$ denotes a set of bug repair pattern types, $p'_a$ is equal to 1 if the target bug repair pattern type for the $i$-th instance is $a$; otherwise, it is equal to 0, and $D$ represents the number of training instances.
4 Dataset

4.1 Bug Repair Pattern Types

Developing a general-purpose prediction model capable of forecasting repair pattern types for bug fixing is exceedingly challenging. One of the reasons for this challenge is that previous research [27] has revealed the recurring nature of certain repair patterns in bug fixing, while others are seldom employed. The infrequent occurrence of some repair patterns makes data collection a challenging task. As a result, our model is tailored to focus on commonly recurring repair patterns, leaving the rarely occurring ones for future exploration. Based on the bug repair pattern taxonomies outlined in Ref. [27], we initially construct a real-world bug dataset, each comprising a triple (buggy file, bug report, repair patterns). Subsequently, we perform an analysis of the dataset and identify the top 25 repair pattern categories most frequently involved in bug fixes, as presented in Table 1. It is worth noting that certain repair pattern categories mentioned in Ref. [27] have been further divided into more precise subcategories. For instance, the repair pattern `mcParValChange` denotes method call parameter value modification, which is further subdivided into more refined categories like `constChangeArgument` and `wrongVarRefArgument`.

4.2 Empirical Analysis of the Distribution of the Number of Repair Patterns Used by FixMiner

FixMiner[7] is considered one of the state-of-the-art tools for applying mined repair patterns to fix bugs, and its performance is evaluated using Defects4J[13]. Based on the experimental results, we compiled a bug dataset consisting of all bugs that FixMiner can successfully fix. For each bug in the dataset, a line-based edit script can be generated by applying GNU diff to the buggy version.

<table>
<thead>
<tr>
<th>ID</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>expLogicExpand</td>
<td>Logic expression expansion</td>
</tr>
<tr>
<td>2</td>
<td>mdAdd</td>
<td>Method call addition</td>
</tr>
<tr>
<td>3</td>
<td>condBlockOthersAdd</td>
<td>Conditional block addition</td>
</tr>
<tr>
<td>4</td>
<td>missNullCheckN</td>
<td>Missing non-null check addition</td>
</tr>
<tr>
<td>5</td>
<td>wrongMethodRef</td>
<td>Wrong method reference</td>
</tr>
<tr>
<td>6</td>
<td>condBlockRetAdd</td>
<td>Conditional block addition with return statement</td>
</tr>
<tr>
<td>7</td>
<td>missNullCheckP</td>
<td>Missing null check addition</td>
</tr>
<tr>
<td>8</td>
<td>wrapsIf</td>
<td>Wraps with if statement</td>
</tr>
<tr>
<td>9</td>
<td>condBlockRem</td>
<td>Conditional block removal</td>
</tr>
<tr>
<td>10</td>
<td>wrongVarRef</td>
<td>Wrong variable reference</td>
</tr>
<tr>
<td>11</td>
<td>constChangeArgument</td>
<td>Method call with different constant parameter</td>
</tr>
<tr>
<td>12</td>
<td>wrongVarRefArgument</td>
<td>Method call with different actual parameter variable</td>
</tr>
<tr>
<td>13</td>
<td>mcRepl</td>
<td>Method call replacement</td>
</tr>
<tr>
<td>14</td>
<td>missAssignment</td>
<td>Assignment addition</td>
</tr>
<tr>
<td>15</td>
<td>condExpMod</td>
<td>Conditional expression modification</td>
</tr>
<tr>
<td>16</td>
<td>expLogicReduce</td>
<td>Logic expression reduction</td>
</tr>
<tr>
<td>17</td>
<td>expArithMod</td>
<td>Arithmetic expression modification</td>
</tr>
<tr>
<td>18</td>
<td>wrapsTryCatch</td>
<td>Wraps with try catch block</td>
</tr>
<tr>
<td>19</td>
<td>constChange</td>
<td>Constant change</td>
</tr>
<tr>
<td>20</td>
<td>condBlockOthersAddElse</td>
<td>Conditional (else) branch addition</td>
</tr>
<tr>
<td>21</td>
<td>unwrapIfElse</td>
<td>Unwraps from if-else statement</td>
</tr>
<tr>
<td>22</td>
<td>expLogicMod</td>
<td>Logic expression modification</td>
</tr>
<tr>
<td>23</td>
<td>wrapsMethod</td>
<td>Wraps with method call</td>
</tr>
<tr>
<td>24</td>
<td>assignExpChange</td>
<td>Assignment expression modification</td>
</tr>
<tr>
<td>25</td>
<td>wrapsIfElse</td>
<td>Wraps with if-else statement</td>
</tr>
</tbody>
</table>
and the patched version of the program. Using these edit scripts, each bug is manually annotated with repair patterns. Figure 4 illustrates the distribution of the number of repair patterns applied in these successful bug fixes. The figure reveals that the current state-of-the-art APR approach, which relies on repair patterns, demonstrates strong efficacy in addressing simple bugs that affect a single statement, and their corresponding fixes require the application of a single repair pattern.

4.3 Bug Repair Pattern DataSet

Several publicly available datasets exist regarding real-world software bugs and bug reports, including Bugs.jar\(^{46,47}\). Bugs.jar encompasses 1 158 real bugs extracted from 8 prominent, open-source Java projects. Additionally, Bench4BL\(^{49}\) comprises bug reports and their corresponding buggy source code files, featuring 10 017 bug reports gathered from 51 open-source projects. We constructed a new bug repair pattern dataset by leveraging these two existing datasets to expedite the data curation process.

Building our bug repair pattern dataset involves two key steps: bug data aggregation and bug annotation. For each bug report in Bench4BL, the corresponding Git commit ID of the bug-fixed version is provided. In contrast, Bugs.jar utilizes the distributed version control system Git to manage and store bug data. This means that within each branch of every project repository, one can find the buggy version of the source code, the associated bug report, the developer’s patch to fix the bug, and the test suite. For Bugs.jar, we systematically iterate through all bugs. During each iteration, we apply the developer’s patch to the original buggy source file, resulting in the fixed version of the source file. This process allows us to collect (buggy file, fixed file, bug report) triples for Bugs.jar. At the end of these iterations, we have gathered a comprehensive collection of (buggy file, fixed file, bug report) triples for Bugs.jar. As for Bench4BL, we first clone the projects’ Git repositories locally. Similarly, we iterate through each bug-fixing commit ID, using the Git checkout command to extract both the pre- and post-fix versions of the buggy file. This process enables us to compile a set of (buggy file, fixed file, bug report) triples for Bench4BL. Finally, we combine the two sets of (buggy file, fixed file, bug report) triples into one unified dataset. Notably, bugs that affect more than one Java file are discarded, and any duplicate bugs are eliminated. Furthermore, we address missing values in bug reports through manual data completion.

Upon acquiring the set of (buggy file, fixed file, bug report)-triples, each bug requires annotation with repair patterns. We employed a semi-automatic method for bug annotation. We initially performed automated annotation for each pair of buggy and fixed files using the PPD tool\(^{46}\). PPD is a tool proficient in the automatic extraction of repair patterns between the faulty and patched code, utilizing GumTree\(^{49}\) and Spoon\(^{49}\). It is worth noting that we made slight modifications to PPD in two ways. First, we extended PPD by incorporating additional repair pattern detections, enhancing its ability to identify more valuable repair patterns, for instance, the wrongVarRefArgument pattern. Secondly, we selectively disabled the tool from detecting certain repair patterns, as some of them offered minimal benefit to APR techniques. An example is the Single Line pattern. Subsequently, we verified the accuracy of the annotation results and rectified any errors based on the edit scripts generated by the GNU diff tool between the buggy file and the fixed file. In the end, we meticulously curated a dataset comprising 7 891 (buggy file, bug report, repair patterns)-triples drawn from Bugs.jar and Bench4BL. Given the emphasis of this paper on single repair pattern bugs and frequently occurring repair patterns, we refined the dataset by filtering out multi-repair pattern bugs and eliminating repair patterns with probabilities below 0.57%. This process yielded a final dataset of 4 242 (buggy file, bug report, repair pattern)-triples, which we refer to as sPATTERN.

As previously mentioned, some repair patterns occur frequently, while others are infrequent. Consequently, the sPATTERN dataset exhibits an inherent imbalance. The distribution of repair patterns in the sPATTERN dataset is presented in Table 2, affirming earlier
findings\cite{27}. Basic descriptive statistics pertaining to the sPATTERN dataset are summarized in Table 3.

<table>
<thead>
<tr>
<th>Repair pattern ID</th>
<th>Repair pattern distribution/%</th>
<th>Repair pattern ID</th>
<th>Repair pattern distribution/%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.20</td>
<td>14</td>
<td>4.72</td>
</tr>
<tr>
<td>2</td>
<td>19.68</td>
<td>15</td>
<td>4.48</td>
</tr>
<tr>
<td>3</td>
<td>15.80</td>
<td>16</td>
<td>3.08</td>
</tr>
<tr>
<td>4</td>
<td>13.20</td>
<td>17</td>
<td>2.92</td>
</tr>
<tr>
<td>5</td>
<td>12.76</td>
<td>18</td>
<td>2.56</td>
</tr>
<tr>
<td>6</td>
<td>11.00</td>
<td>19</td>
<td>2.52</td>
</tr>
<tr>
<td>7</td>
<td>10.88</td>
<td>20</td>
<td>1.76</td>
</tr>
<tr>
<td>8</td>
<td>8.40</td>
<td>21</td>
<td>1.52</td>
</tr>
<tr>
<td>9</td>
<td>6.12</td>
<td>22</td>
<td>1.52</td>
</tr>
<tr>
<td>10</td>
<td>5.60</td>
<td>23</td>
<td>1.44</td>
</tr>
<tr>
<td>11</td>
<td>5.36</td>
<td>24</td>
<td>1.40</td>
</tr>
<tr>
<td>12</td>
<td>5.32</td>
<td>25</td>
<td>1.24</td>
</tr>
<tr>
<td>13</td>
<td>5.20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Bug repair pattern types and their percentages in sPATTERN

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. Code Tokens</td>
<td>565.3</td>
</tr>
<tr>
<td>Max Code Tokens</td>
<td>13 869</td>
</tr>
<tr>
<td>Avg. Summary Tokens</td>
<td>16.8</td>
</tr>
<tr>
<td>Max Summary Tokens</td>
<td>52</td>
</tr>
<tr>
<td>Avg. Description Tokens</td>
<td>382.6</td>
</tr>
<tr>
<td>Max Description Tokens</td>
<td>11 071</td>
</tr>
<tr>
<td>Avg. AST Depth</td>
<td>15.1</td>
</tr>
<tr>
<td>Max AST Depth</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 3 Descriptive statistics of sPATTERN

5 Empirical Evaluation

5.1 Research Questions

PatternNet comprises several design components: 1) the utilization of multi-view information, 2) adopting a hybrid model that integrates a transformer-based and an RNNs-based model, 3) incorporating feature fusion networks. Hence, our evaluation not only assesses the efficiency and effectiveness of each individual model to justify their hybrid integration but also examines how these design choices impact PatternNet’s performance. Note that there are two strategies for feature fusion, i.e., concatenation fusion and co-attention fusion mentioned above. By default, PatternNet uses co-attention fusion strategy to combine different features, which is also called as PatternNet-CoAtt. Similarly, we name PatternNet with concatenation fusion strategy as PatternNet-Concat. Therefore, we examine the following research questions:

- RQ1: To what extent does the length of code affect the prediction performance of UniXcoder?
- RQ2: How accurately does PatternNet predict repair patterns for a reported bug with respect to other techniques?
- RQ3: How do feature fusion methods affect the model performance?
- RQ4: How much longer does it take to train PatternNet compared to the baseline models?

5.2 Experiment Implementation and Setup

We employed PyTorch\cite{31} to implement PatternNet, and it was trained and validated using the sPATTERN dataset at the method level. The ASTs of Java buggy methods were extracted using javalang. During the training of PatternNet, we set the maximum sequence length to 512. In alignment with ASTNN, we utilized the Skip-gram algorithm with word2vec\cite{42} to generate distributed representations of AST symbols, configuring the embedding size to be 128. The hidden dimensions of the ST-tree encoder and the Bi-GRU were both set to 128 and 384, respectively. Furthermore, the embedding dimensions of transformer-based models were all set to 768. As for the co-attention network, the size of the hidden layer $W_c$ was set to be 512 and 768×3, and we set $k=1$.

As the task of bug repair pattern prediction can be formulated as a multi-class classification problem, and given the imbalanced nature of our dataset, all the subsequent experiments employ accuracy and F1-macro as evaluation metrics. We utilize Adam as the optimizer and cross-entropy as the loss function for the model. To mitigate overfitting, we incorporate dropout in the hidden layer with a probability of $p = 0.5$, and during training, we employ early stopping based on the validation set. Our hybrid model is fine-tuned using various combinations of hyperparameters to ensure optimal performance. The hyperparameters are tuned through a grid search procedure with the following parameters and values: we explore learning rates from the set {2E−4, 3E−5, 5E−5, 2E−5}, and batch sizes from the set {8, 16, 24, 32, 64}.
The best model identified through grid search is ultimately tested, and the resulting performance metric values are reported.

**Data used.** While we have introduced sPATTERN, a bug repair pattern prediction dataset, its size remains limited. To enhance the generalization ability of deep learning models, it is necessary to perform code data augmentation by employing semantic and syntax-preserving program transformations. Since each example in sPATTERN consists of a (buggy method, bug report, repair pattern)-triple, we need to apply both code data augmentation and text data augmentation concurrently. Specifically, for each example in sPATTERN, we utilize a Java program transformer and Eda to generate two synthetic buggy methods and their corresponding bug reports. Following data augmentation, the dataset is randomly divided into three subsets for training, testing, and validation, with proportions of 60%, 20%, and 20%, respectively, in a stratified manner. This stratification accounts for the uneven distribution of bug repair patterns. Subsequently, oversampling is applied to balance the training sets concerning the various bug repair pattern classes. Ultimately, the training set, validation set, and test set comprise 25,425, 2,036, and 2,545 (buggy method, bug report, repair pattern)-triples, respectively.

On the other hand, we partition sPATTERN into two subsets: one subset where each buggy method contains 512 tokens or more, and another subset where the code token length of each buggy method is below 512. Similarly, we apply data augmentation, splitting, and oversampling to these two subsets, following the same procedures as those employed for sPATTERN. This results in two datasets, referred to as LongCode-sPATTERN and ShortCode-sPATTERN. For LongCode-sPATTERN, the sizes of the training set, validation set, and testing set are 8,425, 684, and 854, respectively. ShortCode-sPATTERN consists of a training dataset with 17,000 samples, a validation dataset with 1,353 samples, and a testing dataset with 1,691 samples.

**Baseline models.** We compare PatternNet with five types of baseline models utilizing different input modalities. These baseline models are Bi-GRU, ASTNN, UniXcoder, CodeBERT, and GraphCodeBERT. To elaborate, we use these baseline models to learn feature representations of reported bug for repair pattern prediction. Note that neither our proposed models nor baseline models use any hand-crafted features since previous work indicates that hand-crafted features usually fail to capture the semantic characteristics of source code.

Bi-GRU processes the entire bug report as a single sequence, and it employs the average of the hidden states of all words as a feature for bug repair pattern prediction.

ASTNN is an AST-based neural network used for source code representation and is a subnetwork of PatternNet. It is employed to extract the features of buggy code for bug repair pattern prediction.

UniXcoder is a cross-modal pre-trained model for source code representation, currently represents the state of the art and serves as a subnetwork of PatternNet. UniXcoder can take multi-modal data as inputs and output the embeddings of the inputs. As mentioned earlier, each example in the dataset contains buggy method code and its bug report. A bug report typically consists of the bug summary and textual description. In our evaluation of UniXcoder, we consider all possible combinations of input modalities. Specifically, we feed the following combinations into UniXcoder to obtain representations of reported bugs: buggy code and bug report summary, buggy code and bug report description, buggy code alone, and buggy code with complete bug report (i.e., summary and description). These feature representation learning methods are denoted as UniXcoder (code+summary), UniXcoder (code+description), UniXcoder (code), UniXcoder (code+summary+description), UniXcoder (summary), and UniXcoder (description), respectively.

CodeBERT is a bimodal pre-trained model based on transformer architecture for code understanding and generation. We use CodeBERT as a reported bug encoder to encode a pair of buggy code and bug report summary into its CodeBERT embedding.

GraphCodeBERT is a pre-trained model for programming languages that leverages semantic-level information of code, particularly data flow. We utilize GraphCodeBERT to encode a reported bug, which includes a buggy code paired with a bug report summary and the corresponding data flow. This information is fed into GraphCodeBERT to obtain the representation of the reported bug, which is subsequently used to predict bug repair pattern.

### 5.3 Experiment Results

To address the first research question regarding the impact of code length on UniXcoder’s performance, we conducted evaluations on two datasets with varying
lengths of buggy code to investigate differences in bug repair pattern prediction performance. Specifically, we tested UniXcoder on LongCode-sPATTERN and ShortCode-sPATTERN; the results are presented in Table 4. As expected, when the token length of buggy code increases, UniXcoder’s performance deteriorates. This decline in performance for the long code dataset is due to UniXcoder’s practice of simply truncating all code tokens that exceed the 512-token length limit, resulting in the loss of valuable code information.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Accuracy</th>
<th>F1-macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>LongCode-sPATTERN</td>
<td>0.27</td>
<td>0.33</td>
</tr>
<tr>
<td>ShortCode-sPATTERN</td>
<td>0.60</td>
<td>0.65</td>
</tr>
</tbody>
</table>

Table 4 The bug repair prediction performance of different code token lengths based on UniXcoder

For the second research question regarding PatternNet’s accuracy in predicting repair pattern for reported bugs compared to other techniques, we evaluated PatternNet and baseline models trained on the sPATTERN dataset using different input modalities. The experimental results are summarized in Table 5. The table provides three key observations. First, models trained with bi-modal data exhibits slightly superior performance compared to those trained with unimodal data. For instance, UniXcoder (code+summary) achieves an F1-macro score of 0.80, which is 0.02 higher than UniXcoder (code) alone. Second, due to the maximum input length constraint of 512 tokens, longer code or more detailed bug reports do not necessarily result in improved UniXcoder performance. Third, our multi-view and co-attentive representation learning model, PatternNet-CoAtt, outperforms the state-of-the-art UniXcoder by 0.05 in terms of F1-macro.

To address the third research question, which investigates the impact of feature fusion methods on model performance, we modified PatternNet-CoAtt by replacing its feature fusion module with the concatenation fusion strategy, resulting in what we refer to as PatternNet-Concat. Subsequently, to test the performance of PatternNet-Concat, we conducted the same experiment that evaluated PatternNet-CoAtt using identical parameter settings. Table 5 shows that PatternNet-CoAtt outperforms PatternNet-Concat in terms of F1-macro by 0.02. This suggests that the co-attention mechanism, as utilized in PatternNet-CoAtt, can enhance feature extraction performance in bug repair pattern prediction when compared with the concatenation fusion strategy.

To address the fourth research question, we assessed the training efficiency of PatternNet and compared the training times among PatternNet and the baseline models. Specifically, we reported the training times per epoch for PatternNet and all the baseline models, as illustrated in Fig. 5. It is evident from the figure that PatternNet takes approximately 1-2 times longer than UniXcoder to complete one epoch of training. In other words, the co-attention mechanism not only enhances PatternNet’s performance, but also extends the training duration.

![Fig. 5 Training times per epoch of PatternNet and baseline models](image-url)
6 Conclusion

In this paper, we introduce a novel deep neural network named PatternNet for predicting repair patterns of reported bugs. PatternNet utilizes a multi-view deep learning framework to extract high-level multi-view features from both buggy code and bug reports, employing different source code representation learning models. Furthermore, it captures co-attentive features between different statements within the buggy source code and the words in the bug report text. These multi-view features are then integrated into a unified representation using a feature fusion network. We provide quantitative evidence of the effectiveness of PatternNet and the feature fusion network for software bug repair pattern prediction.

In the future, we plan to expand our sPATTERN dataset by including more examples of single repair pattern bugs, which will necessitate further evaluations to assess the generalizability of PatternNet. Additionally, future research could explore how effectively PatternNet can enhance the performance of search-based APR techniques.

References


437-440.


XU Yong et al: Multi-View Feature Fusion Model for Software Bug Repair Pattern Prediction