Automatic Semantic Code Repair Based on Deep Learning for Programs with Single Defect

***Abstract*—Semantic code repair refers to automatically fixing bugs where the actual program code is compiled and executed successfully but fails to generate the output that the programmer intends. This is a challenging problem since fixing a semantic error may require deep analysis of the entire code. As the scale of the project increases, it will cost more time for programmers to debug programs. Most past techniques require unit tests to check whether the generated candidate repairs are valid. To our knowledge, little work has been done to address the problem of automatic semantic code repair. In this paper, we present a semantic code repair method which uses a deep attentional sequence-to-sequence model to predict related information about bugs and generate potential fixes without actually running the program. The model is trained on a large corpus of real-world Java source code that is widely used in software industry, and the source code has been randomly injected with semantic errors. We also prove the effectiveness of our model by evaluating the model on both the test part of dataset and a real-world test dataset. Finally, an automatic semantic defect repair system for Java defects is designed and implemented.**

***Keywords*—Automatic semantic code repair, Deep learning, Chrome plugin**

I. INTRODUCTION

Software has gradually penetrated into the lives of people and has promoted economic growth and industry development from multiple dimensions. However, even large companies' software often suffers from various types of errors, which lead to software runtime errors and even system failures and crashes. A semantic error is different from a syntax error and cannot be directly recognized by a compiler. Fixing semantic errors can cost a lot of manpower, which can seriously affect the efficiency of software development. There exists a large amount of data in the growing open source community. The data can be combined with machine learning methods to better assist with software development and problem fixing. Therefore, automatic defect repair has become a very important research field in software engineering. The potential value of solving this problem is also immeasurable.

There are two types of software defects: syntax errors and semantic errors. Syntax errors refer to the errors caused by the expressions that do not conform to grammatical requirements and can usually be checked by a compiler or by a specific grammar checker. Semantic errors generally refer to the situations in which the actual program code is compiled and executed successfully but fails to generate the output that the programmer intends. This is a challenging problem as fixing a semantic error may require not only deep analysis of the entire code but also the programmer's actual intention and a careful analysis of the program's behavior. Syntax errors may be found easily by compilers, so they seldom exist in the software released to the market. In contrast, semantic errors may be discovered usually through running the program and observing the actual results, and thus they have more potential hazards.

Semantic errors usually refer to the errors that the behaviors of software do not match the programmer's actual intention. That definition covers a wide range of code errors. So we limit our work to a part of simple semantic errors, which is defined as: "Bugs that can be identified and fixed by an experienced human programmer without running the program or having a deep understanding of the context of the code snippet [18]." Although there are some restrictions on the scope of the initial problem, it does not mean that simple semantic errors could be easily repaired. Because the process also needs some time and efforts to read the code snippet and infer the programmer's intent, and it requires some deep programming experience.

According to our study, there are currently few academic studies on the automatic repair of semantic errors. Most of traditional methods need to generate a large number of candidate patches, and then verify and select valid patches based on unit tests. In this paper, we develop a deep attentional sequence-to-sequence model to predict the related information about bugs and generate potential fixes without actually running the program. The model is trained on a large corpus of real-world Java source code that has been randomly injected with seven types of semantic errors. Based on the model, we design and implement a semantic defect repair system.

To summarize, the major contributions of this paper lie in the following aspects:

1) **We propose a new method for constructing large-scale training data based on a large number of high-quality source code through static code analysis, code desensitization and mutation.** The method is proposed mainly to solve the problem that existing semantic defect databases are not large enough to

satisfy the requirement of training a deep learning model. Static code analysis is used to obtain the structure information of code. A desensitization method is designed and implemented to reduce the vocabulary involved in the model inferring stage. Also we design and implement a method for generating mutated data by self-defined mutation operators. Finally, a large-scale dataset that can be used for training semantic defect repair model is generated.

2) **A method for constructing an automatic semantic defect repair model based on deep learning is proposed.** This method simplifies a semantic defect repair problem into a common sequence generation problem in natural language processing. The process of sequence generation implicitly contains the identification of error types, the prediction of defect locations and the generation of repair patches. The model could fix 68.39% samples in the test data set of the mutation data and 36.25% samples in the ground-truth data with semantic defects. **Our model proves that there is great potential for sequence-to-sequence model to solve the semantic defect repair problem.**

3) **An automatic semantic defect repair system for Java defects is designed and implemented. The system is the first to solve the automatic semantic code repair problem based on deep learning for Java defects, to the best of our knowledge.** The system is designed and implemented as a plugin for the Chrome browser based on the deep learning model. The plugin provides defect repair suggestions based on code segments input by users.

II. Related Works

This section covers existing literature in traditional research on automatic code repair, deep learning on automatic code repair, and sequence generation in NLP (Natural Language Processing) applying deep learning techniques.

A. Traditional research on Automatic Code Repair

The traditional problem of automatic defect repair is defined as: given a program and a set of tests, including failed tests, the automatic code repair technology repeatedly modifies the program until it passes all the tests. GenProg proposed by Le Goues et al. in 2009 regards the program as a set of statements [1]. The mutation, insertion, deletion and replacement of the source program are performed through a genetic algorithm to perform loop mutation and test until all defects were fixed. Weimer et al. further propose an improved method out of GenProg called AE [2]. Based on the previous work, a series of rules are set to quickly check equivalent transformations to avoid time cost caused by these meaningless transformations, which greatly reduce the overhead in the mutation stage.

GenProg, as a representative work for the emergence of automatic software defect repair technology, has a great impact on subsequent research. Kim et al. propose the PAR method by replacing the mutated template in GenProg with an artificial template [3]. The method first digs out the most common errors from a large database with artificial patches, and then creates corresponding artificial repair templates. They perform defect repair by template matching on the test program and validate the method on Java. Qi et al. propose RSRepair by replacing the genetic algorithm in GenProg with random search and obtain significant better effects than GenProg [4,5,6].

Some of the research work limit the object of research and focus on solving certain types of defect repair problems. Nopol, proposed by Xuan et al., is the first to fix the If condition in a specialized way [7]. They locate defects through predicate switching and collect all the constraints that pass the tests including fail tests. SMT method is used to solve constraints and synthesize sentences. SemFix proposed by Nguyen et al. is similar to Nopol but supports the assignment statement [8].

As the research moves along, it’s recognized that there exists something seriously wrong in automatic code repair. The goal of defect repair technology should be changed to generate the same patches that a developer actually modified, not just pass the tests [9]. Mechtaev et al. propose DirectFix and Angelix based on SemFix. They adjust the target to generate the least syntactical different fixes and define the difference of patches as the number of elements that are changed in the syntax tree of the program. Then the problem can be solved with MaxSMT [10,11]. Long et al. propose SPR, which inherits the templates of Nopol and PAR, and Prophet, a method to use machine learning to sort the candidate patches generated by SPR [12]. Xiong et al. propose the method ACS to accurately repair the defects of the If condition [13]. Locality principle of variables is used to select the variables in the If statement. Then the operations on variables are selected through a statistical method.

Some limitations do exist in the present work. Traditional defect repair method can only fix a small part of defects in the test data. Since the number of tests in the experimental environment is limited, it’s improper that repair methods aim at passing the tests. So passing all tests does not mean that the program is correct. Some advanced researches on automatic code repair are able to achieve high fix rate on some types of defects. But most of them are under strong constraint conditions.

B. Deep Learning on Automatic Code Repair

As deep learning becomes more and more popular in some research fields, some studies using deep learning to solve automatic code repair problem begin to appear in recent years. Gupta et al. propose DeepFix, which uses a neural network with attention mechanism to repair syntactic errors in programs written by C. 6971 programs about 93 programming tasks are used to train and test the model [14]. The model finally could get complete repair rate of 27% and partial repair rate of 19%. Choi et al. use memory neural network model to construct an end-to-end method which is the first to solve buffer overflow problem without any artificial rules [15]. Allamanis et al. propose to use a deep neural network model to solve the SmartPaste task which means to rationally modify the given piece of code and paste it to the existing code around it [16]. The evaluation of this research shows that the deep neural network model can learn the contextual expressions of variables at different locations and usages in a data-flow sensitive manner. The model finally achieves an accuracy of 58.6%. White et al. propose a deep learning method named DeepRepair to learn the similarity between programs which will be used to discover the redundant statement candidates in search-based code repair method and generate candidate patches by converting the sentence [17]. Correct patches that cannot be discovered by existing search-based repair techniques can be found by this method. Devlin et al. propose a three-phase repair framework named SSC for semantic code repair which uses a deep neural network to model the original code and make scores for candidate repairs [18]. Finally, the candidate repairs are compared to each other to determine the optimal one. The method is tested in the programs written by Python and 41% can be fixed exactly.

Automatic code repair methods based on deep learning have gradually been proved to have great potential. However, deep learning generally requires a very large training data set to obtain a stable and reliable model. However, existing defect databases are difficult to meet the requirement. Gupta et al. train their neural network in 6971 programs, while Choi et al. use 10,000 programs, which are not big enough to get a stable deep learning model. In addition, the current research on automatic repair of semantic defects is inadequate. To our knowledge, Devlin's work is the only one to use deep learning to do research on semantic defect repair.

C. *Sequence Generation in Deep Learning*

Sequence learning is an important research area in natural language processing including machine translation, speech recognition, dialog generation, and text abstraction. Sutskever et al. apply deep learning algorithm to machine translation. They propose Seq2Seq model which uses a multi-layer LSTM (Long Short-Term Memory) to map the input sequence to a fixed-length vector, and then decode and output the target sequence from the vector [19]. Bahdanau et al. introduce the attention mechanism to break the restriction that inputs with different lengths must be encoded as fixed-length vectors [20]. A soft alignment model is conducted by allowing the model to perform appropriate search to focus on the information in the input sequence which is more related to the predicted target. That allows the model to handle long-term dependency in the input sequence better. Luong et al. further extend the calculation of the attention mechanism and propose a new local attention mechanism [21]. The Seq2Seq model with attention mechanism has reached a very advanced level in many sequence generation tasks, such as syntactic parsing [22] and image captioning [23]. Xie et al. use the character-level neural network model with attention mechanism to assist the modification with natural language and improve the language learners' writing skills [24].

In this paper, an automatic semantic code repair problem is regarded as a problem of sequence generation, and we attempt to solve it with Seq2Seq model with attention mechanism.

III. Approach Overview

In this section, we discuss the overall design of our approach to automatically repairing semantic defects. The overall framework is shown in Figure 1.

A. *Static Program Analysis*

Static program analysis is a method that performs program analysis without running the program. Different from a dynamic program analysis which needs to run the program, a static program analysis conducts a program analysis through lexical analysis, syntax analysis, control flow analysis etc. to scan and analyze the whole program code. In this paper, the main purpose of using static program analysis is to parse source code to an

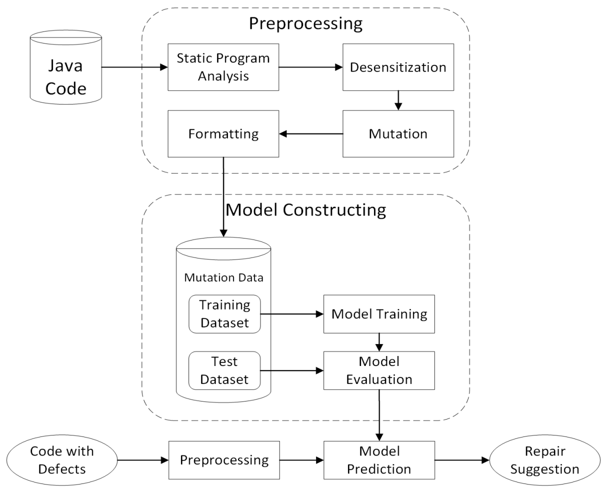


Fig. 1. Overall framework of automatic semantic code repair approach

abstract syntax tree to understand the structure of a program as well as the semantics of each code snippet. Based on the result of the static program analysis, we can implement desensitization and process mutation to generate training data simply by replacing the nodes of abstract syntax tree with suitable ones.

B. *Desensitization* 脱敏

When programmers write code, they often need to name variables and functions properly. In order to ensure the readability of a program, naming is generally considered as the program's functionality, or context and programmers' usual practice. Although there are some general naming notations in programming field, for example, the Hungarian Convention, naming still has great uncertainty. Different naming conventions lead each programmer to have different sets of common vocabularies while a word is the basic unit for natural language processing. If we do not consider the vocabulary problem brought by different naming conventions, the problem will cause the scale of the vocabulary that deep learning model needs to deal with to be too large, which will seriously reduce the quality of deep learning models.

Based on the above discussion, this paper proposes a desensitization method whose target is to eliminate the influence of unique naming conventions of different programmers on the vocabulary involved in a program.

Since naming variables and functions does not have any effect on the problems studied in this paper, the algorithm maps them to the fixed strings in the order in which they appear in the program. The name of the function itself is uniformly mapped to "Main". String constants appearing in the program are mapped to the fixed value "STR", and numerical constants are mapped to the fixed value "0".

The desensitization algorithm introduced above does not take into consideration practical strings that are common to each program, such as variable types, keywords, special characters etc.

运用了哪些变异算子

C. *Mutation*

This section will introduce a mutation-based approach to generating large-scale programs with semantic defects since an end-to-end deep learning model requires a large amount of Java code with semantic defects.

Firstly, we need to identify several semantic defect types as mutation operators. These mutation operators will define the range of defects that our methods would fix. They are abstracted from a large number of actual defect programs and can be simply applied to high-quality correct programs to generate wrong programs injected with specific defects.

Then we can define a mutation process as follows. For each function segment, the mutation-based approach checks whether a mutation operator can be applied to an abstract syntax tree. If so, a mutation operation will be performed on it, and the operator type and the line number where the mutation performed will be recorded.

In this paper, we only discuss the automatic code repair of single mutations. We will try to find the solution which can solve the automatic repair problem for multiple mutations in our future work.

D. *Formatting*

After completing the desensitization and mutation, for each source code snippet, we get:

- Original source code snippet
- The map applied in the desensitization operation
- Code injected with defects after the desensitization and mutation
- Mutation operator type
- Location of performed mutation operation (line number of the source program)

In this paper, the problem of automatic semantic code repair is simplified into the sequence generation in natural language processing. Our defect repair model can learn the statistic features of programs with various semantic defects from a large number of programs injected with mutation defects and their corresponding correct patches, and be able to predict repair patches. The input is the sequence of the code snippet with defects after the mutation, and the target sequence is the corresponding repair patch. We also format the original sequence and target sequence of the model to finally get a dataset that can be used directly for training and testing.

First of all, there will be different writing styles which do not break syntax rules in the program, for example, the left curly braces ("{") used in a function declaration and a loop condition. Second, since a word is the basic unit of the language model processing, word segmentation should be conducted. Different character sequences can be cut apart to reduce the size of the vocabulary and make the structure of a program clearer.

In addition, we also perform operations on the source and target sequence as follows. Each sequence S is a function fragment composed of multiple lines of code, and the lines are separated by a line break character. Let $s_1, s_2, s_3, \dots, s_n$ denote each line of source sequence, $l_1, l_2, l_3, \dots, l_n$ denote the line number, then S is represented as "$(s_1, l_1), (s_2, l_2), (s_3, l_3), \dots, (s_n, l_n)$ ". Such a representation method can provide the information related to the structure of code snippets and improve the model's quality compared to the method using original input sequence.

Target sequence consists of the type of the mutation operator, the line number of the mutation, and a related patch. Let m denote the type of a mutation operator, l denote the line number of a mutation, and p denote the related patch sequence. We represent the target sequence T as " $(m : l, p)$". If the input sequence has not been randomly mutated, the corresponding target sequence of the correct code snippet is "(0)". We expect the model can distinguish the correct code snippet from those who have simple semantic defects by adding Type 0. We do not directly use the complete correct program as the target sequence but only use the line where the mutation is performed. Since only a single mutation is performed, other rows of the source program are not changed. This representation reduces the length of the target sequence, helps to reduce the influence of long-term dependence in model prediction phase and improves the prediction quality. It can also reduce time cost while training and testing the data.

E. *Model Constructing for Single Defect*

This model is based on the Seq2Seq neural network model proposed by Bahdanau et al. [20]. They use a RNN (Recurrent Neural Network) encoder to process the input and a RNN decoder with attention to generate the output sequence. The basic unit of both the encoder and decoder is GRU (Gated Recurrent Unit). The GRU is an extension of LSTM unit. It consists of two gate calculations, including update gate and reset gate. Update gate is used to control how much state information is brought into current state. The larger the value of update gate is, the more previous the state information will be retained. Reset gate is used to control how much the state information of the previous moment will be ignored. The smaller the value of reset gate is, the more the discarded information will be.

In a one-way GRU, if data is input in order, the jth hidden state can only carry the information of the jth word itself and words in previous sequence. If data is input in reverse order, only the information of the jth word and words in the subsequent sequence are included. In order to further extract the information before and after current timestamp, our model uses a bidirectional GRU layer, i.e. BiGRU as an encoder. A BiGRU unit is a combination of two single GRU units. At each moment, the input will be provided to both GRUs at the same time, and the output will be a combination of two single GRU's output.

Both the encoder and decoder are stacked by N basic units. The encoder layer maps each token in input sequence to a real vector. For an input sequence, $x_1, x_2, \dots, x_{T_x}$, the hidden state at time t is determined by the input of the last time state and the current time, which is calculated as follows.

$$h_t^{(1)} = BiGRU(h_{t-1}^{(1)}, x_t)$$

$$h_t^{(n)} = BiGRU\left(h_{t-1}^{(n)}, h_t^{(n-1)}\right), \forall n \in \{2, \dots, N\}$$

After summarizing the hidden state at each time point, the final context vector is generated and delivered to the decoder.

$$C = q(h_1, h_2, h_3, \ldots, h_{T_x})$$

To avoid encoding all input sequences into a fixed-length context vector which represents the information of the input sequences not so well, attention mechanism is introduced to compute a context vector for each token in the output sequence. The context vector is generated to represent part of the input sequence which should be focused on next prediction. Then the model can automatically select most suitable context information for current output token when predicting the target sequence. The context vector c_t is computed as follows.

$$a_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T_x} \exp(e_{tk})}, j \in \{1,2,3,T_x\}$$

$$e_{tk} = \emptyset(s_{t-1}, h_k^{(N)})$$

$$c_t = \sum_{j=1}^{T_x} a_{ij} h_j^{(N)}$$

a_{ij} represents attention distribution when the ith token in the output sequence is generated. The larger the number is, the more important the jth input token is to the ith output token.

The hidden states of the decoder are updated as follows.

$$s_t^{(n)} = GRU(s_{t-1}^{(n)}, s_t^{(n-1)}), \forall n \in \{2, \ldots, N\}$$

$$s_t^{(1)} = GRU(s_{t-1}^{(1)}, z_t)$$

Among them, z_t is the concatenation of the previous output $\hat{y}_{t-1}$ and the context vector c_t. Finally, the output of decoder $s_t^{(n)}$ is concatenated with c_t. A softmax layer is followed to compute the distribution of next output token. We calculate cross-entropy over outputs of the softmax layer at each step and add them together to compute the final loss.

IV. Experiments

In this section we apply our approach to solve 7 types of semantics errors in Java program and prove the effectiveness of our model by evaluating the model on both the test part of dataset and a real-world test dataset. Our approach could also be used to solve other semantic errors.

To present experiments results, we implement an automatic semantic code repair system for Java defects based on the approach we propose above. The system provides automatically semantic defect checking and bug fixing for Java programs in some extent, without compiling and any external environment configuration.

A. Dataset

The open source community has grown rapidly in recent years. Many companies open some of their source code and maintain it by developers around the world. Many excellent open source projects are also widely used in commercial projects. To create the dataset for training and evaluation, we download first 1% (about 500) repositories sequencing in star numbers under the Java topic, which have permissive licenses. We extract the Java files in these repositories which amount to 3.5 million files and extract every function from each Java file as a code snippet. Each code snippet will be treated as a single research object. This requires that each code snippet should be able to parse to an AST (Abstract Syntax Tree) by a static program analyzer for subsequent desensitization, mutation, and other operations. There is no need for code snippets to be runnable.

Finally, we measure the length of each code snippet. It requires that the length should not be too short, because the code snippet needs to contain enough information for model inference. It should not be too long to avoid the existence of long-term dependences reducing the strength of the model. Approximately 450,000 code snippets with 8-15 lines of code, which represent general code snippets in the repositories we choose, are kept for experiments finally according to the limitations above.

B. Preprocessing

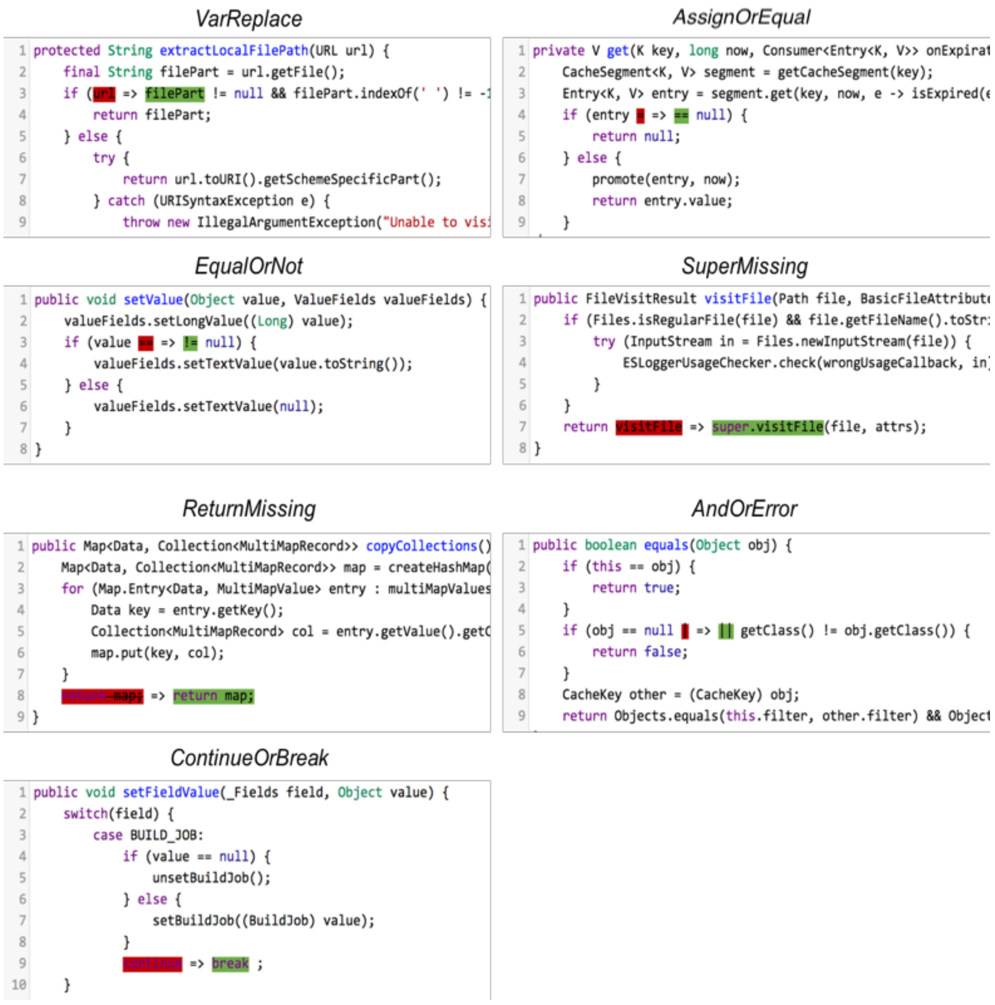
The static program analysis tool used is JavaParser[1]. JavaParser is a light static program analysis tool that provides the functions including parsing, analyzing and modifying the abstract syntax trees for the code written in Java.

Before desensitization, we have a vocabulary whose size is about 10^6. The desensitization algorithm helps to reduce the size of vocabulary by more than 10 times, which effectively reduces the complexity of prediction on the probability distribution over the vocabulary. At the same time, in order to facilitate the recovery of the desensitized process after the experiment, it is necessary to record and store the desensitization map used for each code snippet in the desensitization process.

In this experiment, we mainly deal with Java semantic defects of 7 types which are described as follows.

- ***VarReplace***: A variable in the code snippet is used incorrectly and should be replaced with another variable that appears before it.
- ***AssignOrEqual***: An assignment operator ("=") is incorrectly used at a particular location and should be replaced by an equality operator ("==").
- ***EqualOrNot***: An equality operator ("==") is incorrectly used at a particular location and should be replaced by an inequality operator ("!=").
- ***SuperMissing***: The "super" accessor is missing from a call to the superclass function at a particular location.

[1] http://javaparser.org



VarReplace

```java
protected String extractLocalFilePath(URL url) {
    final String filePart = url.getFile();
    if (url => filePart != null && filePart.indexOf(' ') != -1
        return filePart;
    } else {
        try {
            return url.toURI().getSchemeSpecificPart();
        } catch (URISyntaxException e) {
            throw new IllegalArgumentException("Unable to visi
```

AssignOrEqual

```java
private V get(K key, long now, Consumer<Entry<K, V>> onExpirat
    CacheSegment<K, V> segment = getCacheSegment(key);
    Entry<K, V> entry = segment.get(key, now, e -> isExpired(e
    if (entry = => == null) {
        return null;
    } else {
        promote(entry, now);
        return entry.value;
    }
```

EqualOrNot

```java
public void setValue(Object value, ValueFields valueFields) {
    valueFields.setLongValue((Long) value);
    if (value == => != null) {
        valueFields.setTextValue(value.toString());
    } else {
        valueFields.setTextValue(null);
    }
}
```

SuperMissing

```java
public FileVisitResult visitFile(Path file, BasicFileAttribute
    if (Files.isRegularFile(file) && file.getFileName().toStr
        try (InputStream in = Files.newInputStream(file)) {
            ESLoggerUsageChecker.check(wrongUsageCallback, in
        }
    }
    return visitFile => super.visitFile(file, attrs);
}
```

ReturnMissing

```java
public Map<Data, Collection<MultiMapRecord>> copyCollections()
    Map<Data, Collection<MultiMapRecord>> map = createHashMap(
    for (Map.Entry<Data, MultiMapValue> entry : multiMapValues
        Data key = entry.getKey();
        Collection<MultiMapRecord> col = entry.getValue().getC
        map.put(key, col);
    }
    ~~return map;~~ => return map;
}
```

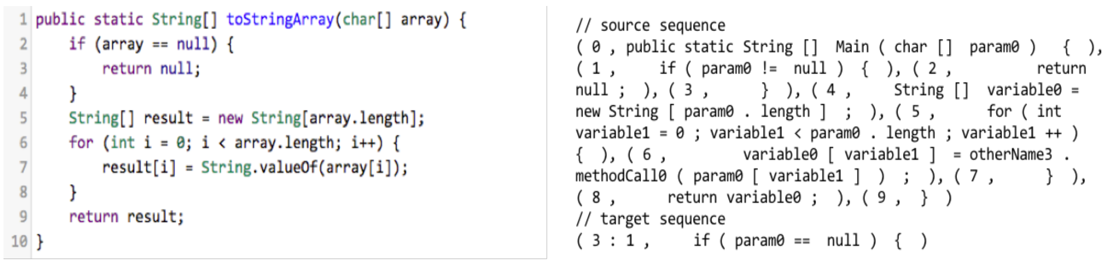
AndOrError

```java
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null | => || getClass() != obj.getClass()) {
        return false;
    }
    CacheKey other = (CacheKey) obj;
    return Objects.equals(this.filter, other.filter) && Object
```

ContinueOrBreak

```java
public void setFieldValue(_Fields field, Object value) {
    switch(field) {
        case BUILD_JOB:
            if (value == null) {
                unsetBuildJob();
            } else {
                setBuildJob((BuildJob) value);
            }
            continue => break ;
    }
```

Fig. 2. Example of 7 kinds of simple Java semantic defects



```java
public static String[] toStringArray(char[] array) {
    if (array == null) {
        return null;
    }
    String[] result = new String[array.length];
    for (int i = 0; i < array.length; i++) {
        result[i] = String.valueOf(array[i]);
    }
    return result;
}
```

```
// source sequence
( 0 , public static String []  Main ( char []  param0 )   {  ),
( 1 ,     if ( param0 !=  null )  {  ), ( 2 ,          return
null ;  ), ( 3 ,      }  ), ( 4 ,      String []  variable0 =
new String [ param0 . length ]  ;  ), ( 5 ,      for ( int
variable1 = 0 ; variable1 < param0 . length ; variable1 ++ )
{  ), ( 6 ,          variable0 [ variable1 ]  = otherName3 .
methodCall0 ( param0 [ variable1 ]  )  ;  ), ( 7 ,      }  ),
( 8 ,     return variable0 ;  ), ( 9 ,  }  )
// target sequence
( 3 : 1 ,     if ( param0 ==  null )  {  )
```

Fig. 3. A code example before and after preprocessing

- ***ReturnMissing***: Control stream reaches the end of functions with non-null return value but no value is returned.
- ***AndOrError***: The use of logical operators And ("&&") and Or ("||") is confused with bitwise operator And ("&") and Or ("|").
- ***ContinueOrBreak***: The keyword "continue" is used instead of "break", or vice versa.

See Figure 2 for a real code example of these 7 type defects.

By implementing a single mutation, we finally obtain around 1500,000 simple mutated code snippets for model training and evaluation. In addition, we also add some original code snippets that have not been randomly mutated to the dataset. Therefore, the model may get the ability to distinguish the correct code snippet from those who have a semantic defect.

See Figure 3 for a code example before and after preprocessing.

TABLE I. DATASET AND EVALUATION RESULT

Error Type	**Size**	**Type Accuracy**	**Location Accuracy**	**Exact Accuracy**
Correct	45423	88.11%(40024)	/	/
VarReplace	31716	61.80% (19601)	52.71% (16720)	39.83% (12633)
AssignOrEqual	12174	99.42% (12104)	99.31% (12091)	89.60% (10908)
EqualOrNot	18421	90.91% (16747)	89.58% (16503)	82.72% (15239)
SuperMissing	1680	88.69% (1490)	88.33% (1484)	85.29% (1433)
ReturnMissing	14414	99.34% (14320)	98.45% (14191)	87.60% (12628)
AndOrError	5393	96.55% (5207)	96.16% (5186)	81.49% (4395)
ContinueOrBreak	543	84.53% (459)	83.24% (452)	82.50% (448)

C. *Training*

We implement the sequence-to-sequence model described in this paper by Tensorflow. Each layer of the network contains 256 units. The model limits the length of the input sequence to 150 according to the most general length of the code snippets and the length of the target sequence to 30. The size of mini-batch during training is 64. We use Adam to train the network and the learning rate is a fixed value 0.0004. Dropout of a rate of 0.2 is used. The parameters described above are the best-performed parameters that we have obtained.

D. *Results*

We evaluate our model on the test set of the data generated by the mutation. We first test whether the model can output a correctly formatted prediction sequence with the regular expression method. In fact, 99.55% of the model's output sequences have correct format.

For those correctly formatted prediction sequences, we calculate three evaluation indicators as follows.

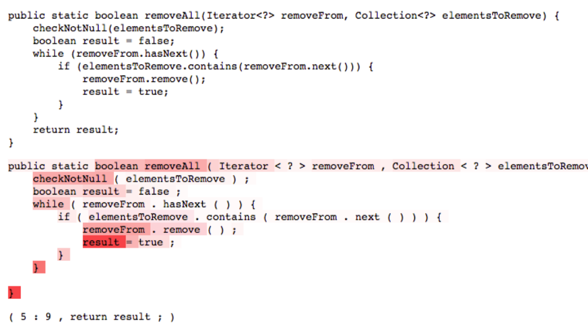
- ***Type Accuracy***: accuracy of the model predicting correct defect type (including type 0).
- ***Location Accuracy***: On the basis of 1, the model also predicts correct location (line number) in original code snippet where the mutation is performed.
- ***Exact Accuracy***: On the basis of 2, the model fixes the defect exactly (the output sequence contains exactly the same patch as the corresponding part of the original code). For samples belong to type 0, only ***Type Accuracy*** needs to be calculated. The evaluation results are shown in Table I.

Based on the test results, we conclude our summary about the model for semantic code repair as follows.

- The model can predict whether a code snippet has semantic defects with a high accuracy. According to the test results, it can be calculated that the model can get a F1-Score of 0.8473 on the binary classification problem that whether there is a semantic defect in the input sequence.
- The model has better predictions for the mutated data of three semantic defect types: ***AssignOrEqual***, ***SuperMissing*** and ***ReturnMissing***. Their accuracies are relatively high. Consistent with human intuition, we believe that these three problems are three simpler types of mutation mentioned compared with other four ones in this paper. Only a small amount of contextual information is needed to make prediction. For example, when we observe that an assignment statement appears in a condition statement or there are logical operators around it, then we may speculate that the assignment statement probably has semantic errors and should be replaced with an equality statement.
- The model has relatively low predictions for repairing the mutated data of three semantic defect types: ***EqualOrNot***, ***ContinueOrBreak***, and ***AndOrError***. We speculate that the main reason is that these three types of defects need to be solved by understanding more contextual information.
- The model has a low prediction on ***VarReplace*** defects and the accuracy rate is less than 40%. We think that checking whether a program has improper used variables and then repairing them is a very difficult problem. It requires a detailed understanding of the meaning and usage of the variables in the whole code snippet.

E. *Attention Visualization*

We visualize the model's attention weight which is assigned to each token in the input sequence when predicting the output sequence to gain an insight how the prediction is performed. We first choose one key token, for example, the token that the mutation is performed, then we visualize the attention weights when predicting this token. The visualization contains three parts: the original unmuted code snippet, the attentional weights on the input sequence and the output sequence of the model. The background color represents the weight of attention. The heavier the background color is, the bigger the attention weight assigned to that token is. At the same time, we perform an inverse desensitization operation on the input sequence and output sequence to facilitate correspondence with the original sequence.



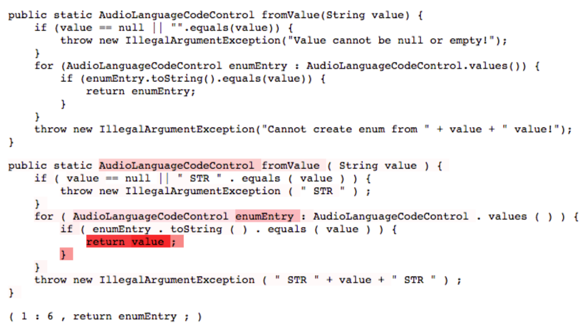
```
public static boolean removeAll(Iterator<?> removeFrom, Collection<?> elementsToRemove) {
    checkNotNull(elementsToRemove);
    boolean result = false;
    while (removeFrom.hasNext()) {
        if (elementsToRemove.contains(removeFrom.next())) {
            removeFrom.remove();
            result = true;
        }
    }
    return result;
}

public static boolean removeAll ( Iterator < ? > removeFrom , Collection < ? > elementsToRemove ) {
    checkNotNull ( elementsToRemove ) ;
    boolean result = false ;
    while ( removeFrom . hasNext ( ) ) {
        if ( elementsToRemove . contains ( removeFrom . next ( ) ) ) {
            removeFrom . remove ( ) ;
            result = true ;
        }
    }

}

( 5 : 9 , return result ; )
```

Fig. 4. A visualization example of attention weights for ***ReturnMissing***

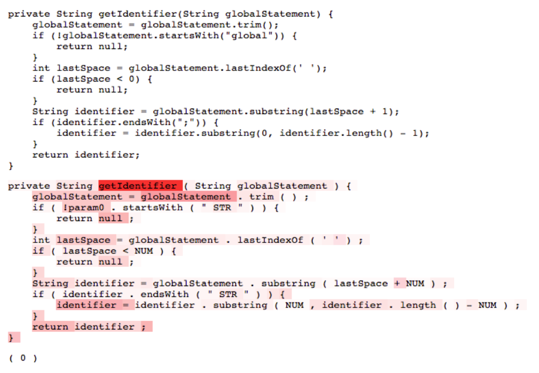


```
public static AudioLanguageCodeControl fromValue(String value) {
    if (value == null || "".equals(value)) {
        throw new IllegalArgumentException("Value cannot be null or empty!");
    }
    for (AudioLanguageCodeControl enumEntry : AudioLanguageCodeControl.values()) {
        if (enumEntry.toString().equals(value)) {
            return enumEntry;
        }
    }
    throw new IllegalArgumentException("Cannot create enum from " + value + " value!");
}

public static AudioLanguageCodeControl fromValue ( String value ) {
    if ( value == null || " STR " . equals ( value ) ) {
        throw new IllegalArgumentException ( " STR " ) ;
    }
    for ( AudioLanguageCodeControl enumEntry : AudioLanguageCodeControl . values ( ) ) {
        if ( enumEntry . toString ( ) . equals ( value ) ) {
            return value ;
        }
    }
    throw new IllegalArgumentException ( " STR " + value + " STR " ) ;
}

( 1 : 6 , return enumEntry ; )
```

Fig. 5. A visualization example of attention weights for ***VarReplace***



```
private String getIdentifier(String globalStatement) {
    globalStatement = globalStatement.trim();
    if (!globalStatement.startsWith("global")) {
        return null;
    }
    int lastSpace = globalStatement.lastIndexOf(' ');
    if (lastSpace < 0) {
        return null;
    }
    String identifier = globalStatement.substring(lastSpace + 1);
    if (identifier.endsWith(";")) {
        identifier = identifier.substring(0, identifier.length() - 1);
    }
    return identifier;
}

private String getIdentifier ( String globalStatement ) {
    globalStatement = globalStatement . trim ( ) ;
    if ( !param0 . startsWith ( " STR " ) ) {
        return null ;
    }
    int lastSpace = globalStatement . lastIndexOf ( ' ' ) ;
    if ( lastSpace < NUM ) {
        return null ;
    }
    String identifier = globalStatement . substring ( lastSpace + NUM ) ;
    if ( identifier . endsWith ( " STR " ) ) {
        identifier = identifier . substring ( NUM , identifier . length ( ) - NUM ) ;
    }
    return identifier ;
}

( 0 )
```

Fig. 6. A visualization example of attention weights for correct input

Figure 4 shows an example of ***ReturnMissing*** when predicting the return value. It's easy to find that the model observes a large amount of contextual information when deciding the return value of this function. The attention weights focus on the type of return value in function declaration, the type of the real return value and expression of the real return value "result" when generating correct prediction of the return value.

Figure 5 shows an example of ***VarReplace***, a complex defect type, when predicting the correct variable. The input sequence incorrectly replaces the variable p_3 in line 7 with a meaningless variable. The model exactly predicts the repair patch by observing the statements of variable p_1, p_2 which have similar behaviors and the nearest variable declaration of p_3.

Figure 6 shows an example for correct input sequence. We find that the model assigns a lot of attention to many tokens in the input sequence when the error type is 0. We think that it may be because the model needs to observe the whole code snippet to confirm that the input sequence does not have the 7 kinds of semantic defects introduced in this paper.

TABLE II. REAL DATASET AND TEST RESULT

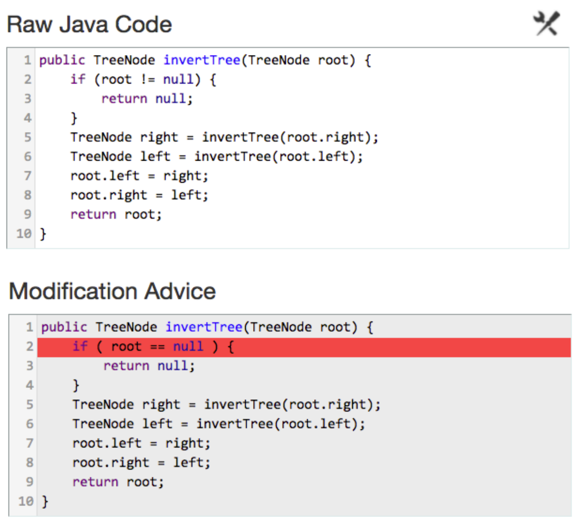
Error Type	Erroneous Code Snippets	Exactly Fixed Code Snippets
EqualOrNot	24	9
ContinueOrBreak	3	2
AndOrError	5	5
SuperMissing	2	2
AssignOrEqual	1	1
VarRepalce	34	9
Total	80	29

F. Real-Defect Test

Now we have obtained an effective deep learning model that can be used for Java semantic defect repair. When used to check and repair Java semantic defects in a code snippet of a Java function, the code also needs to be preprocessed according to the method proposed and then the result can be used as an input to the model for prediction.

We further test this model in a ground-truth dataset where the code samples have Java semantic defects. The source of the test data is the commits submitted by users to github. However, automatically extracting Java semantic defects from commits is a difficult task. Therefore, we have to limit the extraction. First of all, in order to ensure that the commits are submitted for fixing a certain defect in the original code, we filter the message of the commits and only retain the commits that contain a word in the list "bug / error / exception / fix / issue". In order to ensure that the faulty code snippet can be obtained accurately, we limit the extraction to commits that only have one file changed and only one modification in the file. After manual scanning, we finally obtain 80 defective data from the repositories and their patches that fall within the 7 kinds of semantic defect types introduced in this paper. Then the previous model is used to test and the results are shown in Table II.

We finally obtain ground-truth data which contains six kinds of simple semantic defects. For the ***ReturnMissing*** type, we think that it may rarely appear in the programming of skilled programmers. But this type of defects still exists and has serious effect. It has research value. Since the error code in the real dataset may involve multiple or more complex potential errors, the actual accuracy rate will be slightly lower than the training data set. For example, for ***EqualOrNot*** type, in the real data set, there are many complex cases such as multiple judgment statements in the conditional expression. The model proposed in this paper achieves an average repair rate of 36.25%. Since there are many samples of ***VarReplace*** type that have a low repair rate in the dataset, we believe that the accuracy of the model may be underestimated. In the future we will continue to test our model on larger scale real datasets.



```java
public TreeNode invertTree(TreeNode root) {
    if (root != null) {
        return null;
    }
    TreeNode right = invertTree(root.right);
    TreeNode left = invertTree(root.left);
    root.left = right;
    root.right = left;
    return root;
}
```

```java
public TreeNode invertTree(TreeNode root) {
    if ( root == null ) {
        return null;
    }
    TreeNode right = invertTree(root.right);
    TreeNode left = invertTree(root.left);
    root.left = right;
    root.right = left;
    return root;
}
```

Fig. 7. Main page of the plugin

V. System Implementation and Case Study

We implement an automatic semantic code repair system for Java defects. The system can provide some functions like automatically semantic defect checking and bug fixing at a certain degree of accuracy for Java function code without compiling any external environment configuration to help programmers in daily Java programming. We want to make it as lightweight as possible, so we implement the system as a Chrome plugin with a client and a server service.

The client needs to complete the following functional modules:

- Code acquisition module. It is mainly responsible for obtaining the user-supply raw output code snippet. The system provides two ways of code acquisition: obtain the raw text from other web pages opened by the browser or directly input the text by users in the web page displayed by the plugin.
- Code management module. It is mainly responsible for the display of program texts and provides functions like code editing.
- Code repair module. It is mainly responsible for providing users with interfaces for repairing the input sequences, displaying the results and showing the alert messages that occur during the repair process, for example, the input code snippet cannot be parsed to AST.

The server is mainly responsible for receiving the original code snippet input by users and preprocessing to get the input sequence that can be used by the deep learning model. The model is used to generate the predictions for defect types, locations and patches. Finally, a prediction result is generated and returned to the client for processing.

Then we will show the system through a case study. We select a scenario where a user is coding in Java on the online programming website Leetcode[2]. Our system is used to help to check whether a semantic defect exists in the code and generate a patch to repair the defect automatically.

When a user runs into a wrong answer when submitting the written code in the website and then hopes to check whether his code has a semantic defect. The code snippet that has entered on the site is selected and a Chrome menu is called up by right-clicking on the selected text. In the menu, there is an option to check for defects in the selected code. By clicking on this option, the main page of the plugin will be opened as Figure 7.

The main page is divided into two parts. The left side of the page is responsible for displaying the input source code snippet. In addition to displaying the input code snippet, it also supports editing operations. So the user can also input the code snippet here and click on the icon in the upper left corner to perform the automatic check and repair the operations. The right side is responsible for displaying the result of the automatic repair by the system. The two-column display in this way helps the user to find out the specific patch details through a comparison. We also modify the background color of the line that has different contents on two sides to facilitate the user's observation.

With this system, developers can check the code for the location of the error that the algorithm gives, before spending a lot of time sorting out the code. It can help developers find defects more accurately and faster. At the same time, some errors may not be exposed until the actual use due to insufficient unit tests, which has serious impact. The system can also analyze and prompt these undetected semantic errors, and spend less time reducing the possibility of semantic errors. This has a crucial impact on the development of large software projects in industry.

VI. Conclusion

In this paper, we present an end-to-end approach based on deep learning to automatically repairing programs with single defect. In order to get a large scale dataset that can meet the requirements of the deep learning model, we propose a method based on a large volume of high-quality source code through static code analysis, code desensitization and mutation. A method for constructing an automatic single semantic defect repair model based on deep learning is proposed. We prove that there is great potential for sequence-to-sequence model to solve this problem. Our model achieves 68.39% accuracy when be tested in the mutated data and it can fix 36.25% samples exactly in the ground-truth data with semantic defects. Finally, a system for automatically repairing programs with single defect is designed and implemented as a lightweight Chrome plugin based on the deep learning model. The plugin can collect the code segments generated or directly typed by users in an online programming environment and provide defect repair suggestions. We verify the feasibility and effectiveness of the system by illustrating a typical case study. In the future, we plan to experiment our model on a larger ground-truth dataset. We will also try to solve the automatic repair problem for programs with multiple defects.

[2] https://leetcode.com/

ACKNOWLEDGMENT

REFERENCES

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," IEEE Transactions on Software Engineering, vol. 38, pp. 54, Jan 2012.

[2] W. Weimer, ZP. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering, pp. 356-366, May 2013.

[3] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 802-811, May 2013.

[4] Y. Qi, X. Mao, and Y. Lei, "Making automatic repair for large-scale programs more efficient using weak recompilation," Proceedings of the 28th IEEE International Conference on Software Maintenance, pp. 254-263, Sep 2012.

[5] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 180-189, Sep 2013.

[6] Y. Qi , X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 254-265, May 2014.

[7] J. Xuan, M. Martinez, F. Demarco, M. Clement, SL. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," IEEE Transactions on Software Engineering, vol. 43, pp. 34-55, Jan 2017.

[8] HDT. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," Proceedings of the 35th International Conference on Software Engineering, pp. 772-781, Jul 2013.

[9] Z. Qi , F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, pp. 24-36, Jul 2015.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, vol. 1, pp. 448-458, May 2015.

[11] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," Proceedings of the 38th international conference on software engineering. ACM, pp. 691-701, May 2016.

[12] F. Long, and M. Rinard, "Staged program repair with condition synthesis," Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, pp. 166-178, Aug 2015.

[13] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," Proceedings of the 39th International Conference on Software Engineering. IEEE Press, pp. 416-426, May 2017.

[14] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing Common C Language Errors by Deep Learning, " Proceedings of the 31st AAAI Conference on Artificial Intelligence, 2017, pp. 1345-1351.

[15] MJ. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," arXiv preprint arXiv:1703.02458, Mar 2017.

[16] M. Allamanis, and M. Brockschmidt, "SmartPaste: Learning to Adapt Source Code," arXiv preprint arXiv:1705.07867, May 2017.

[17] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities," arXiv preprint arXiv:1707.04742, Jul 2017.

[18] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic Code Repair using Neuro-Symbolic Transformation Networks," arXiv preprint arXiv:1710.11054, Oct 2017.

[19] I. Sutskever, O. Vinyals, and QV. Le, "Sequence to sequence learning with neural networks," Advances in neural information processing systems, pp. 3104-3112, 2014.

[20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," arXiv preprint arXiv:1409.0473, Sep 2014.

[21] MT. Luong, H. Pham, and CD. Manning, "Effective approaches to attention-based neural machine translation," arXiv preprint arXiv:1508.04025, Aug 2015.

[22] O. Vinyals, Ł. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton, "Grammar as a foreign language," Advances in Neural Information Processing Systems, pp. 2773-2781, 2015.

[23] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," Proceedings of the 2015 International conference on machine learning, pp. 2048-2057, 2015.

[24] Z. Xie, A. Avati, N. Arivazhagan, D. Jurafsky, and AY. Ng, "Neural language correction with character-based attention," arXiv preprint arXiv:1603.09727, Mar 2016