DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2022

Java Syntax Error Repair Using RoBERTa

KTH Thesis Report

Ziyi Xiang

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Ziyi Xiang <zxiang@kth.se>
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Examiner

The Professor: Martin Monperrus
KTH Royal Institute of Technology

Supervisor

The Supervisor: He Ye

KTH Royal Institute of Technology

Abstract

Deep learning has achieved promising results for automatic program repair (APR). In this paper, we revisit this topic and propose an end-to-end approach Classfix to correct java syntax errors. Classfix uses the RoBERTa classification model to localize the error, and uses the RoBERTa encoder-decoder model to repair the located buggy line. Our work introduces a new localization method that enables us to fix a program with an arbitrary length. Our approach categorises errors into symbol errors and word errors. We conduct a large scale experiment to evaluate Classfix and the result shows Classfix is able to repair 75.5% symbol errors and 64.3% word errors. In addition, Classfix achieves 97% and 84.7% accuracy in locating symbol errors and word errors, respectively.

Keywords

Java program repair, RoBERTa, Neural machine translation architecture

Abstract

Deep learning har uppnått lovande resultat för automatisk programreparation (APR). I den här uppsatsen återkommer vi till det här ämnet och använder Classfix för att korrigera java-syntaxfel. Classfix använder en RoBERTa-classification model för att lokalisera felet och en RoBERTa-encoder-decoder model för att reparera buggar.

Vårt arbete introducerar en ny lokaliseringsmetod som gör att vi kan fixa program av godtycklig längd. Studien kategoriserar fel i symbolfel och ordfel. Vi genomför storskaliga experiment för att utvärdera Classfix. Resultatet visar att Classfix kan fixa 75.5% av symbolfel och 64.3% av ordfel.

Dessutom uppnår Classfix 97% och 84,7% noggrannhet när det gäller att lokalisera symbolfel respektive ordfel.

Nyckelord

Reparation av Java-program, RoBERTa, Neural maskinöversättningsarkitektur

Acknowledgements

I would like to thank my supervisor Ye and examiner Martin. Their suggestions and expertise guided me through a very challenging time. I am very grateful for the effort that they put into helping me to complete the study.

Acronyms

NLP Natural language processing

RNN Recurrent neural network

NMT Neural machine translation

LSTM Long short-term memory

GRU Gated recurrent unit

BERT Bidirectional Encoder Representations from Transformers

RoBERTa Robustly Optimized BERT Pretraining Approach

GNN Graph neural network

Contents

Chapter 1

Introduction

1.1 Background

Debugging programming errors is a key instrument in software development. It is a time-consuming task and requires programmers to spend hours manually fixing errors [58]. As the program becomes increasingly complex involving multiple lines and long-range dependencies, fixing these errors becomes difficult [63].

One of the most common sources of bugs is syntax errors [9, 42, 51]. Syntax errors include common mistakes such as missing delimiters, brackets and misspelling variables, which can be captured by the compiler.

These errors are often caused by the programmer's inattention but are often hard to repair due to the difficulties of fault localization [58]. Error messages sent back by the compiler may lack detail information or point to the wrong place. Errors that involve closing curly brackets often cause the compiler to point to adjacent lines instead of the incorrect line. Experienced programmers may have learned how to deal with these less precise messages, while inexperienced programmers may still suffer from the uncertainty [58].

As a result, people spend hours locating errors and fixing them manually, causing a lot of resources to be spent [43]. An earlier study [42] finds that programmers spend large portion of the development time on debugging rather than coding. Optimizing the debugging process improves overall productivity and may save billions of dollars [5, 66].

Driven by these factors, extensive research has been carried out on this subject. From the rule-based approaches, such as GenProg [27], Anglix [40], Nopol [61], ect; and more recent papers which mainly based on deep learning. Given the recent potential of deep learning, a branch of neural based program repair approaches have been proposed, including SequenceR [12], DrRepair [63] and DeepFix [18].

Many deep learning based studies rely on Recurrent neural network (RNN) enhanced by attention mechanisms [6, 18, 42]. These approaches achieve high repair accuracy by adapting optimization mechanisms, such as Graph neural network (GNN) and pointer generator network [2]. Although they have achieved some great successes, the RNN-based approaches still have some shortages and limitations [55]. Despite being slow to train, the encoder needs to compress arbitrary amounts of information into a fixed-length vector which may cause a potential information loss. In addition, the long-range dependencies can also get lost when passing long sequences sequentially.

Given the limitations of RNN, studies have considered alternatives [7, 14, 20]. One of the research areas is based on the transformer model, which is still relatively new in the field of automatic repair. There are a lot of studies focus on the C program, while tools that designed for Java are comparatively few.

In this thesis, an end-to-end approach is proposed to correct java syntax errors. Our approach uses a classification model to localize the error, and an encoder-decoder model to repair the line.

To localize the error, we are introducing a new localization method that addresses the input size limit of the transformer model. The transformer architecture only allows a fixed-length input, while the input buggy code can be arbitrary long. To solve this problem, our localization method implements a sliding window that enables us to localize error one paragraph at a time. In addition, the study proposes an optimization technique that can be used to reduce noise when localizing the error.

To repair the error, we have a repair process that is similar to neural machine translation, where the model translates a buggy program to a fixed-line. More detail is presented in the later chapters.

1.2 Problem Statement

In this thesis we address the problem of repairing syntax errors in the language Java produced by the compiler, assuming there is one error line per buggy file. In addition, we address the problem of input size limit of transformer, allowing our models to fix arbitrary long files.

1.3 Research Questions

We propose a new architecture called $Classfix$ to fix Java errors. We want to measure the effectiveness of our model on repairing real world Java bugs by comparing with the state-of-the-art approach.

In this thesis, we use Synfix [2] as the baseline for the evaluation. Synfix is a cutting edge approach to repairing Java programs that has substantially dominated prior work in the area of syntax error repair. The study is one of few that uses the transformer architecture instead of RNN and has achieved significant performance gains over previous state-of-the-art including Santos [48], Deepfix [18], SequenceR [12], RewardRepair [64],and BF+FF [1]. At the time of doing this project, Synfix is one of the best tools to fix Java syntax errors, making it an excellent candidate for our baseline.

The first research question is: How does Classfix's effectiveness compare to Synfix?

For fault localization, rather than relying on the compiler error message, our study uses the RoBERTa classification model to localize the error. The classification model takes the buggy file as input and predicts the index of the buggy line.

The second research question is: How accurate is the $Classfix_{classification}$ at localizing the error line?

The third research question is connected to the second research question. In this study, we have purposed a novel abstraction technique to improve the error localization caused by symbols. The intuition behind the method is to abstract unnecessary words to mitigate the variance of programmers coding style. Ignoring these variables decreases the vocabulary size and reduces the noises, which can potentially improve

the localization rate of symbol related errors.

The third research question is: To what extent can our abstraction optimization improve the localization of symbol errors?

The localization method can sometimes fail to localize the error. The best way to evaluate the model of $Classfix_{generation}$ is to assume perfect localization.

The fourth research question is: How good is $Classfix_{generation}$ at repairing errors when assuming perfect localization?

To repair the error, our $Classfix_{generation}$ component takes a small paragraph surrounding the buggy line as input and generates a fixed line to replace the buggy line. There are bugs that require more surrounding context to be able to fix. Concatenated more context lines allow the model to correct long-range dependency errors, but at the same time increase the task complexity. Therefore, the last objective of this thesis is to explore the relation between the length of the context lines and the correctness of the repair.

Although extensive research has been carried out on automatic repair, no single study exists that studies the impact of the contextual lines. In our experiment, we assess three different sizes of contextual lines. We aim to learn the impact and find the appropriate size for the paragraph that provides the highest repair rate.

The last research question is: How do different sizes of context lines affect the repair rate?

To sum up, our research questions are:

- RQ1(End-to-End Effectiveness): How does Classfix's effectiveness compare with Synfix?
- RQ2(Localization only): How accurate is $Classfix_{classification}$ at localizing the error line?
- RQ3(Localization Optimization): To what extent can our abstraction optimization improve the localization of symbol errors?

- RQ4(Repair only): How good is $Classfix_{generation}$ at repairing errors when assuming perfect localization?
- RQ5 (Impact of context lines): How do different sizes of context lines affect the repair rate?

1.4 Contributions

We design and explore a novel neural repair model called Classfix. The source code is available at https://github.com/lingqi1219/Classfix. Classfix consists of classification models to localize errors and encoder-decoder models to generate the patch. We use the dataset from Blackbox[10] to pre-train and fine-tune for the localization and repair tasks.

To sum up, our contributions are

- We design and explore a new neural program repair model Classfix. We conduct a large experiment to evaluate the effectiveness of Classfix, our experiment results show that our model is able to obtain state-of-the-art level repair accuracy.
- We introduce a new localization method that makes it possible to fix file with arbitrary length, while previous approaches often have a fixed-length input.
- We propose the abstraction optimization method for locating symbol errors. This optimization method enables us to localize the symbol error with a precision of 97% regardless of the document length.
- We are first to examine the effect of contextual lines to the best of our knowledge. The results of our experience show that increasing the input size does not necessarily mean that the model can fix the bugs with long-range dependency. The results suggest training multiple models with different sizes of context and combining the result.

1.5 Outline

In the background section, general background and previous studies on program repair are presented.

The method section outlines our approach, the choice of method and the research process. The method section also contains the detail of pre-training, optimization strategy, data collection and evaluation process.

Finally, the results section answers the problem statement. The discussion section explains the shortcomings and limits of this study and discusses the research question.

Chapter 2

Theory

In this chapter, we are presenting the related theoretical background. The theory section aims to serve as an brief introduction for readers who are not familiar with the technique.

The basic concepts of deep learning, RNN [52], transformer [55], Bidirectional Encoder Representations from Transformers (BERT) model [15] and Robustly Optimized BERT Pretraining Approach (RoBERTa) [31] are explained in this section.

2.1 Recurrent Neural Network

Recurrent neural networks RNN [52] are one of the most powerful and well-known architects designed to recognize sequential data, such as audio and text. It has a notable impact across a wide variety of applications, including speech recognition, image recognition and language translation.

RNN processes tokens sequentially with an internal memory that memorizes the information it received previously. While the traditional feed-forward neural network has a fixed-length input and does not have internal memory keeping track of the data from the previous time step. The feed-forward network can only consider the input from the current time step and forget information from the past, making it difficult to model sequential data.

When processing the data, RNN takes both the current input and the state it memorizes from the past and uses a cycle structure to read input with an arbitrary length.

2.1.1 Seq2seq Architecture

One way to utilize RNN is the seq2seq model. The seq2seq [52] model is widely used in sequence-to-sequence modelling such as Neural machine translation (NMT) and summarization, which the model takes an input sequence and generates an output sequence. The seq2seq model has an encoder to encode the input sequentially and compresses information to fixed-length vectors. The decoder decodes the vectors and converts them to sequence.

An example usage e in language translation is to use the encoder to encode a Swedish word and use the decoder to generate an English translation.

2.1.2 Long Short Term Memory

In RNN, the information from the previous states will be used as an input for the next time step. Thus, computing the partial derivatives requires back propagation through layers and time steps. At the same time, a sequence can be arbitrarily long and the output from the later time step can be difficult to affect the earlier time-step due to the decaying error back-flow.

Since the gradient tends to be zero and vanishes, the model will forget the information it had seen earlier, making the training slow and become difficult to capture long-range dependencies[13].

To mediate this vanishing gradient problem, ”Long Short-Term Memory” and ”Gate Recurrent Unit” [13] are introduced. These two methods use gate mechanisms to regulate the flow of information, such that, it can learn to only keep relevant information and forget non-relevant data. The improved RNN with Long short-term memory (LSTM) or Gated recurrent unit (GRU) enables it to train on long sequences. In the past years, many state-of-the-art studies of RNN used either LSTM or GRU to mitigate the vanishing gradient problem.

2.1.3 Attention

Attention is used to differentiate the importance of tokens [54]. By saving the previous states in a fixed-length vector, the model can access all information it had seen earlier. However, when the input is long, representing all previous states with one single vector will cause the loss of information.

Addressing this problem, the attention mechanism is introduced. The idea is that there are some parts of the inputs that are more relevant than the other and need to pay more attention to. In a traditional RNN architecture, the prediction will only look at the final vector which has the condensed information. While with the attention mechanism, we want to attend to all previous hidden states and pay more attention to more informative states. This is achieved by introducing an additional neural network layer that learns how much attention the model needs to attend to each hidden state.

The attention mechanism has greatly improved results for many Natural language processing (NLP) tasks, such as translation, summarization, and program repair. It allows the RNN model to focus on the most relevant part as needed, making it easier for the model to deal with long sequences.

2.1.4 Out-of-vocabulary Problem

The text is in discrete space and thus we need to use a look-up table to tokenize words and convert them to word embedding. The word embedding is a numerical input to represent the word. Many program repair studies use a fixed-size vocabulary to define the mapping from variable tokens to the integer representation. They often define a large pool that is enough to cover a wide range of samples. However, the real-world samples often contain additional unseen words and variables. When dealing with these unseen variables, these models can not recognize these words, as the result, fails to fix the error.

The out-of-vocabulary problem is a must-solve problem for many NLP tasks, especially for program repair that involves many custom variables. A way to address this problem is to use a pointer-generator network. This network can decide whether or not to copy a word from the input to the output. This copy mechanism allows the RNN model to copy unseen words that are not introduced in the vocabulary. The mechanism is first introduced in the pointer generator network paper [49] and has lately been used in many program repair studies [58, 63].

Another way to address the issue is to use sub-word tokenization algorithm. It allows to separate word into character-level units. Such that, a token can be part of the word as well as the whole word. The sub-word tokenization allows the model to recognize known character-level parts instead of unknown words.

Some known subword tokenization algorithms include Byte-Pair Encoding(BPE), WordPiece [50], SentencePiece [46] and unigram language modelling.

2.2 Graph Neural Network

The GNN has a convincing performance and high interpretability on many graph-related tasks and works particularly well with structures such as social networks and system recommendations

In the field of automatic repair [44], many studies suggest using the GNN to model the source code. The mostly used source code graph is the Abstract syntax tree which is a tree representation of code's syntactic structure. However, the abstract syntax tree can only be extracted from a syntax-error-free program. This means that it cannot be used on the syntax error repair. For the syntax error repair, how to construct an expressive representative graph can be a problem.

2.3 Transformer Architecture

The RNN model has some drawbacks, such as long-range dependencies and sequential computation. Although the introduction of LSTM reduces the vanishing gradient problem by a large margin, and the attention mechanism improves the performance; the issue remains while the sequence becomes long.

On the other hand, passing data sequentially does not utilize modern GPU effectively. The next coming token has to wait for the computed result from the previous states which makes the computation not be parallelizable.

Moreover, many studies use the optimization technique called bidirectional RNN which calculates an additional context vector from right to left, making it even more computational heavy.

In 2017, A new encoder-decoder architecture called transformer was proposed. This architecture has shortly become state-of-the-art in the field of NLP, replacing RNN in a wide range of applications. Compared with RNN, the transformer can pass all tokens simultaneously which supports parallel computation.

The transformer is mainly based on the multi-headed self-attention mechanism.

The attention mechanism in RNN weighs the relationship between all input tokens and output tokens. While in self-attention, the transformer model weighs the relationship only between the input tokens. This self-attention can be calculated multiple times to generate multiple independent relationship maps to represent different relationships.

Our study uses the transformer architecture instead of RNN for the following two reasons:

First, transformers do not process data in order, allowing the model to train in parallel, utilizing the modern GPU more efficient. This feature reduces the training time, allowing the model to train more epochs given the same training amount of resources.

Second, RNN tries to compress all information into a fixed-length vector. It may cause some potential information loss, especially when handling long sequence input like a program file.

2.3.1 BERT

BERT stands for Bidirectional Encoder Representations from Transformers. It is a transformer-based natural language processing model proposed by researchers at Google in 2018[15]. It achieves state-of-the-art results on multiple NLP tasks and has shown a better performance over legacy models like RNN.

The original architecture when the transformer [56] is proposed is an encoder-decoder model. It has an encoder that transforms sentences into embedding and has a decoder that generates output sequentially. BERT only uses the encoder part of the transformer and stacks 12 transformer encoder layers on top of each other to learn contextual relations between words.

After pre-training, BERT converts text input into a numeric representation and fine-tunes the embedding to learn a specific task such as classification and machine translation. The pre-training refers to the model being trained on a large generic corpus to learn the pattern, and the fine-tuning refers to the pre-trained model being adapted to a particular task.

There are two tasks that the BERT model is pre-trained on are: Mask language modelling(MLM) and Next sentence prediction(NSP).

In mask language modelling, the input sentence will be filled with a mask token to replace a random token in a sentence. Like filling in blanks, the BERT model is trained to fill in these blanks to learn the bidirectional contexts.

For the next sentence prediction task, BERT learns to predict whether or not a sentence follows another sentence. It aims to learn the context order across the sentences.

2.3.2 RoBERTa

RoBERTa [31] is an extension of the BERT model. It has the same architecture as the BERT but uses an improved training mechanism to pre-train the model.

There are two main differences in the pre-training step.

Firstly, RoBERTa [31] removes the Next Sentence prediction. The experiment from RoBERTa has showed that the NSP pre-training may hurt performance, and removing NSP can slightly improve the performance of the model. Their paper has also showen that using Masked language modelling alone is enough to catch inter-sentences understanding.

Secondly, the Masked language modelling has been improved in RoBERTa. In BERT, the masking is done once during data preprocessing step. The model will train multiple epochs on the same masking dataset repeatedly and overfits the masking pattern on the training data. As an improved version, RoBERTa uses dynamic masking, which generates a new masking pattern whenever a sequence is fed into the model. By performing dynamic masking, the RoBERTa model generates more training cases than the BERT model, which are more efficient on the training resource.

Chapter 3

Background

3.1 Program Repair

There exist varieties of bugs that developers may encounter, such as functional defects, syntax errors and security bugs [44].

The functional defects can be defined as a deviation of the expected result and expected result.The program behaves in an unintended way and fails to pass the test suite. The syntax error is an error in coding that is usually caused by mistakes. This type of error can be captured by the compiler and causes an error message to be generated. The security bug may introduce vulnerabilities in the software, which can be exploited by the hacker to gain privileges or unauthorized access.

To fix the bug, the programmer has to put extensive effort to analyze the execution failure and identifying the cause of failure. They also need to validate the logic to ensure if everything works correctly.

To reduce this workload, a large number of algorithms and techniques have been integrated and generate a rich body of work [16, 44]. These approaches can be classified as generate and validate (G&V) approach and synthesis-based approach. The notable G&V systems included JAID [11],ARJA [65],Astor [37] and PraPR [17].

The notable synthesis-based approaches include ACS [59], NOPOL [60], Angelix [41] and S3 [26] which convert the search problem to a satisfiability problem. All these approaches often work by extracting a repair constraint typically via symbolic execution incorporated with human knowledge for patch generation.

3.1.1 Generate&validate Repair

The generate-and-validate based approaches are mainly designed to repair general bugs such as functional defects. It can also be used to serve bugs like buffer overflow. This type of approaches has been extensively investigated in the past few years, and various of algorithms have been experimented and studied by the researcher.

The generate-and-validate based approach generates fixes by exploring potential solutions. The repair program iteratively creates candidate patches and validates the correctness of the patched programs. The candidate fix that successfully passes all test cases are considered as the plausible patches.

The patches are often generated using a set of change patterns, e.g., operators [30]. Two examples of change operators are atomic change operators and pre-defined template operators.

Atomic change operators modify one single place by deleting, modifying, and inserting the target statement. The generation of patch is often guided by algorithms to create a calculated or randomized combination of atomic changes. GenProg [28], HDRepair [25], and SCRepair [21] are examples that used atomic change operators to generate patches.

Pre-defined template operators is another to generate the patch. In some complex cases, a randomized combination of atomic changes is hard to obtain the deserved result. Therefore, a pre-defined operation is required to perform more complex modification. In particular, it can modify the program at multiple location coherently. Examples that use pre-defined template operators are ARC, AutoFix-E [41], SPR [33] and Prophet [32].

Most of these approaches validate the correctness by running the test cases. The fixed location is usually localized by a localization methods. For example, spectrumBased Fault Localization (SBFL) [53] is one of the localization methods that has been used in many different papers.

The candidate fix that is successfully validated will be returned as a potential fix. An example of validation is the test suite, which is an input-output based specification. The input can example be a set of interrelated objects passing to a method. The output can be the expected result, execution state or execution behaviour.

Two main strategies to handle the modification operation and the search for the candidate solution are search-based and brute-force. The search-based strategies are often guided by different heuristic search algorithms, and the brute-force strategies search for every possible modification obtained by the algorithm.

3.1.2 Synthesis-based Repair

The synthesis-based repair driven approach is an analytical procedure whose solution is guaranteed to solve the repair problem.

The synthesis-based repair consists of three main activities. The behavioural analysis activity analyzes the buggy program to extract the semantics information. The problem generation activity investigates the extracted information and identifies the repair target. The fix generation activity attempts to solve the task identified from the previous activity.

The synthesis-based repair is often designed to address a specific fault, for example concurrency faults, and safety policy violation and data type misuses. The semantic driven approach requires understanding the problem comprehensively. The key of the semantics-driven approach is program synthesis, in which the process takes a set of input and output pairs and produces a function that satisfies all conditions. For patch generation, basic components such as operators, constant value and variables are combined. The patch generation task is often translated to a first-order logic constraint problem and use Satisfiability Modulo Theory (SMT) solvers to find the solution. DirectFix [39], SimFix [22], NOPOL [60] are examples of semantic driven approaches.

Although this type of approach fixes the problem, it may introduce additional errors during the process. Thus the developer still needs to put manual effort to ensure the fixed program works correctly.

3.1.3 Functional Neural Program Repair

In the early period, researches attempted to defined rule-based algorithms to fix different types of errors. There were also some studies that suggested to use language models RNN for repairing the error. In the recent years, RNN-based solutions seem to have drawn the most attention with a large and still growing body of literature. Along

with RNN, Transformer and convolutional network have also gained some interest. Some notable neural program repair approaches include SequenceR [12], CURE [23] and CoCoNuT [35]. Transformer based studies include [2, 38].

SequenceR uses a sequence-to-sequence RNN model enhanced with the attention mechanism, design to fix Java code. According to the original paper, this model captures a wide range of repair operators without any domain-specific top-down design. The SequenceR model is inspired by the RNN neural machine translation architecture. It has an LSTM encoder to process the input and an LSTM decoder to generate the output. The idea is to translate buggy lines into the fixed-line. A key of this approach is to use copy-mechanism to address the out-of-vocabulary problem.

CURE also has neural machine translation architecture similar to SequenceR. One of the key ideas of this approach is to pre-trained the model on a large code corpus to learn the rigorous syntax of programming languages and how programmers write code. CURE designs a search strategy to find the most suitable patches by exploring partial code segments in the translated code snippets. To address the out-of-vocabulary problem, CURE uses a subword tokenization technique called byte pair encoding (BPE) instead of copy-mechanism. For the model architecture, CURE uses a generative Pre-trained Transformer (GPT) instead of RNN. CoCoNuT uses CNN instead of transformers and recurrent neural networks. The idea is to use stacked CNN layers to extract hierarchical features at different granularity levels. CoCoNuT uses a tokenization method analogous to word-level tokenization, which is essentially a space-separated tokenizer. To reduce the size of the vocabulary, CoCoNut considers underscores, camel letters and numbers as separators.

3.1.4 Syntax Neural Program Repair

In 2016, Bhatia and Singh [8] presented a technique that uses the recurrent neural network for repairing syntax errors. The idea is inspired by the earlier probabilistic models of source code that learn the language model to capture the regularities. Their RNN model are modelled on syntactically correct code and aims to predict token sequences to fix errors. Their model has been evaluated on 14,203 student submissions and successfully repaired 31.69% of the buggy submissions.

Bhatia and Singh's study has proven that the RNN model is capable of modelling the syntax repair problem.Their work has been discussed by a number of researchers.

Inspired by their study, researchers have started experimenting with RNN in program repair, adapting some optimizations such as GNN, attention mechanism and pointer-generator model.

For C program repair, Deepfix [18] is one of the most popular benchmark that are used by many later studies. The original Deepfix paper uses an RNN-based seq2seq model that treats the repair task same as a neural translation. Deepfix enhances the RNN model with an attention mechanism allowing the model to track long-range dependencies. Their model evaluates 6971 erroneous C programs written by students and achieves a repair accuracy of 27%.

Despite the low repair accuracy, the architecture is supported by numerous researchers later on, include TRACER [3], Sample fix [19], GGF [58], DrRepair [63], and BIFI [62].

TRACER [3] uses a finely tuned encoder-decoder RNN with attention and has an optimization to replace all literals and identifiers with abstract tokens representation.

Samplefix [19] builds on the general idea of the seq2seq model with attention. In contrast to other frameworks, Samplefix can generate and evaluate multiple fixes by learning the distribution of correctness.

DrRepair [63] uses a combination of RNN, Graph attention network and Pointer Generator network. On top of the LSTM encoder of the RNN model, DrRepair inserts a graph attention layer for capturing dependencies between identifiers. For the decoding part, it uses a pointer generator network to address the Out-of-vocabulary problem.

Graph-based Grammar fix(GGF) [58] fixes errors by extracting an incomplete Abstract syntax tree. Their model architecture is similar to DrRepair, but uses GGNN [29] instead of a Graph Attention network [57].

DrRepair and GGF both use the mechanism of graph neural networks. The strength of graph neural networks in program repair is to model dependencies between nodes and track variables across the source code. Structural information such as variable can be represented as a node and their model aggregates information between connected nodes to capture relations and dependencies.

Alongside with syntax error, there are other approaches that deal with syntactical styles include STYLER [34], CodeBuff [45] and NATURALIZE [4]. STYLER designs to correct violations of code formatting conventions and trains LSTM networks for localization and repairs. STYLER fixes 38% of the errors on its benchmark and carries out the state of the art on the repair of Checkstyle errors outperforms CodeBUFF and NATURALIZE.

In the past few years, researchers have also started to investigate the transformer architecture. As the pre-trained model [15] become widely accepted in NLP tasks, transformers have recently gained increasing popularity in many applications. Some more recent papers in program repair BERT surpasses RNN based approaches in terms of repair accuracy [2, 7, 20]. Transformers do not get too much attention in the field of syntax repair in terms of the number of literature compares with RNN.

Chapter 4

Methodology

4.1 Architecture of Classfix

In this study, we purpose Classfix which can fix syntax errors in Java. Classfix has two main components: the localization component $Classfix_{classification}$ and the repair generation component $Classfix_{generation}$. The localization component localizes the error line, and the generation component generates a one-line patch to replace the buggy line. For the error localization task, Classfix uses the RoBERTa classification model [31]. For the patch generation, Classfix uses the $Classfix_{generation}$ model to translate the target buggy line that is similar to the previous work[25].

The figure4.1.1 illustrates the repair process. The input to Classfix is the buggy source code. The source code will be passed to the localization component to predict suspicious error lines. Based on the result from the localization, the buggy lines are then concatenated with context lines and pass to the generation component to generate the patch.

Figure 4.1.1: An overview of Classfix

4.1.1 Input and Output of Classfix

The input of Classfix is the entire buggy java file. Classfix takes the file as input and localizes the buggy line using its localization component. Classfix follows [24, 36] to concatenate the localized buggy line with its surrounding context lines. The concatenated line is then passed to the patch generation component to generate a candidate patch. Classfix uses the Javac compiler to validate the suggested fixes. The compilable patches will be suggested to the user to represent the output.

4.1.2 Code Representation

The neural network model cannot accept human-readable text rather than the encoded numerical vector. The input file must be converted into tokens before passing to the model.

In programming, a lot of words and variables are infrequent, making the program difficult to deal with. Programmers may compose multiple words without the separation, or use the uppercase letter to represent the separation instead. Moreover, variables can be arbitrary long and complex which makes the created word dictionary

extremely large and contains many infrequent words.

To address these issues, our study uses a BPE tokenizer [31] to create the vocabulary. BPE tokenizes words using a sub-word tokenization algorithm based on the frequency of each word in the training corpus. In this study, the BPE tokenizer is trained on the pre-training corpus and generates a sub-word vocabulary consisting of 50 000 tokens.

Classfix represents the source code as a sequence of sub-tokens using BPE tokenizer. Each token will be assigned to a unique index number obtained from the dictionary.

4.1.3 Inference

In the inference phase, a user inputs the buggy code into Classfix. Classfix converts the text input into suitable format, tokenizes them and feeds them to the trained models.

Once the Classfix model is trained, the main cost in the inference is the classification and generation.

4.2 Localization Component $Classfix_{classification}$

In this section, we introduce the localization component $Classfix_{classification}$ in detail.

4.2.1 Classification Model Input

Previous works [18, 20, 63] attempt to pass the entire program as the input. However, it does not perform well for the following two reasons. First, the transformer model like RoBERTa has a maximum input constraint of 512 tokens, whereas the input program is often much longer than 512 tokens. Second, it is difficult to obtain a uniformly distributed sample for each label class, particularly for high index values.

To address these limitations, Classfix breaks long input files into several sections, where each section consists of a 20-line paragraph. The model localizes the error in a small paragraph and determines whether or not an error is located in this section, and at which sub-index the error is located.

As shown in the Figure 4.1.1, the input of the classification model is n lines of source code from buggy program. In this work, we configure $n = 20$.

4.2.2 Classification Model Output

The output of $Classfix_{generation}$ is one of the $n + 1$ labels as shown in table4.2.1: including "no error found", and number in a range of $1 - n$, which indicates the index of the most suspicious buggy code.

Classification output	
Output	Description
0	No error had been found in the selected section.
1-20	Found the error on the target line.

Table 4.2.1: Classification output labels

As shown in the example 4.2.1, Java class Hello has a duplicate semicolon and a missing brace on line 5 which cause an error. In this example, the file "Hello.java" has already been separated into a 5 line section. The expected output of the classification model will be the error line number, which is 5 in this example.

Figure 4.2.1: The input and output of $Classfix_{classification}$

4.2.3 Segmentation

Our localization model is acting like a sliding window that makes an iterative loop in the document.

As shown in the example in Figure 4.2.2, the sliding window moves downwards, 5 rows at each time step. The first part goes from line 1 to line 20. The next section shifts

5 rows, beginning at line 6 and ending at line 25. There are overlaps between two sections, which are 6 to 20 in this example.

Have some overlapping lines is essential to follow the error which required contextual information such as parentheses. Like in this example, there is an extra parentheses on line 20. Assuming the model correctly predicts the result, it should return 'no error' in the first section, and 'error on line 20 or 21 in the second section.

Figure 4.2.2: Moving Section 5 lines per iteration

The user can decide how many rows to move by iteration. Minor steps have a greater chance of locating the error but at the cost of computing resources. Using larger steps is less onerous in the calculation, but may be less precise. In this experiment, $Classfix_{class}$ configures three steps and five steps.

After looping through the document, the classification models will get a list of suspicious lines. The generation component will attempt to repair each of them.

4.2.4 Abstraction

When passing the input file to the model, Classfix has an option to abstract away some irrelevant ”word”.

The abstraction method is an optimization designed to optimize the localization accuracy of symbol related bugs. For this type of errors, the program may not need to understand the linguistic meaning of words and variables. Therefore, some noisy and irrelevant variable names can be covered with an abstraction to reduce the noise.

To abstract away all irrelevant tokens and at the same time reserve the important keywords to construct the code body, Classfix uses a counter. The counter counts the 300 words that appear most frequently among a large java program corpus. These 300 most frequently used words are used to represent the skeleton of the code. We observe that words above 300 words do not contain much useful information rather than some nonsense names and numbers.

A simply illustration has shown in figureFigure 4.2.3. Infrequent words such as variable name, printed message and numbers have been abstracted away. The rest of the words are preserved.

Figure 4.2.3: Example of an abstracted program

4.2.5 Model Architecture of $Classfix_{classification}$

For the localization task, Classfix uses RoBERTa classification model due to its great performance in a wide range of the text classification task.

The input sequence will be appended with a control token before feeding into the RoBERTa network. As shown in the picture4.2.4, the input passes through 12 transformer encoder layers and compute the embedding of a CLS control token and each input token.

When training a classification model, the control token is used as the aggregate sequence representation and the other embedding is often ignored. The CLS embedding is sent to a pooling layer and linear layer to compute the probability of each class.

Figure 4.2.4: The implementation of $Classfix_{classification}$ with RoBERTa

4.3 Generation Component $Classfix_{generation}$

4.3.1 Input and Output

The classification model localizes the error lines and passes them to the generation component.

The $Classfix_{generation}$ extracts the paragraph surrounding the lines and use the encoder decoder model to generate the patches.

The output of the $Classfix_{generation}$ model is the repaired line to replace the buggy line. The generation component validates the patch by compiling the patched program. It returns the compilable patch as the output.

As shown in the Figure 4.3.1, the generation component takes the error line with its surrounding context as the input. <E> <L> <R> symbols are appended prefixes to denote the relative position. The encoder decoder model uses these prefixes to keep track of the line order as shown in the Figure 4.3.2. After appending the prefixes, the paragraph will be tokenized and pass to the model.

Figure 4.3.1: Example input output of $Classfix_{generation}$

Figure 4.3.2: Input example of a buggy line and the surrounding context

Fixing a bug often depends on the context, which is the statements before and after the buggy lines. Concatenation of long contexts will make the input sequence long, which will add noise to the model making it hard to retrieve useful information.

More recent approach like Synfix addresses this problem by using single buggy line and ignores the surrounding contexts [2]. This can potentially simplify the computation of the model, but also skips errors that require dependency reasoning.

Classfix wants to keep a shorter input for simple errors but also want to have more context lines to correct complex errors. As the result, Classfix trains the model with 3 different context lengths. For each suspicious line, we can get 1 patch from each of these models. If one of them is compilable, we can return that patch as the candidate fix. If multiple patches are compilable, Classfix returns them to the user as the suggestions.

4.3.2 Model Architecture of $Classfix_{generation}$

Sascha Rothe [47] conducted an empirical study on a transformer-based encoder-decoder model that is compatible with pre-trained checkpoints. It has shown the effectiveness of initializing the encoder decoder model with the pre-trained checkpoint on multiple benchmarks, and yields state-of-the-art results[47] .

Our model encoder decoder model inherits the weight from a pre-trained RoBERTa model. By inheriting the weights, the model warm-starts to skip the costly pre-training.

To initialize from a RoBERTa checkpoint, the encoder and decoder have a stack of 12 transformer blocks composed of feed-forward layers and multi-headed self-attention layers. The encoder layers initialize with weights from RoBERTa and the decoder shares these weights in its respective layers. The decoder layers that do not exist in RoBERTa will be randomly initialized.

4.4 Program Source Errors

Classfix addresses errors that can be fixed by modifying one single line and trains separate models to fix either symbol errors or word errors. Classfix distinguishes syntax error into two types and uses separate models to repair errors. Earlier studies attempt to fix multiple types of errors by using one single model [9, 14]. However, this increases the complexity of the repair task, and also impedes the error specific optimization to be applied.

Classfix addresses two types of syntax errors that are mainly caused by programmer's inattention. The first error type relates to the symbol, such as missing semi-column and brackets. We can fix these errors easily by inserting or changing the symbols. The second error type relates to the "words". The word errors can be fixed by modifying the "words" such as variable names, types and declarations. Example errors can be type misused, misspelling, and wrong return type. For errors that require changing both symbols and words at multiple positions, programmers have to manually fix the code by looking at the single line patches generated from each category.

Symbol errors are errors that can be fixed by inserting, removing or changing symbols without changing words and content. As shown in the Table 4.4.1, one of the most

commonly seen errors in this class is missing or additional semicolons and brackets. In this case, the compiler may return error messages such as ” symbol expected”, ”illegal start of expression” and ”reached the end of file while parsing”. This type of errors can also resulting to other messages such as ”else without if” and ”not a statement” depending on the context. An incomplete if-else clause with more or fewer brackets leads to ”else without if”. A string with enclosed quotation marks leads to ”unclosed string literal”. Some other commonly seen errors in this class include missing commas and plus sign in a print statement, using a colon instead of a semicolon at the end, or using a comma instead of a semicolon in a for loop etc.

The word errors can be seen as a supplement to the symbol related errors and addressing errors that require changing words and values. Some typical examples in this category are shown in the Table 4.4.2. Missing or misspelt keywords such as class, interface, void and return can cause ”class interface or enum expected”. Missing type declaration or misspelt variable name lead to ”cannot find symbol”. The unspecified return type can cause ”invalid method declaration”. Using a variable that begins with number or symbol returns ”illegal character” etc.

Notably, the same error can cause different compiler messages to return depending on the context.

Symbols related errors		
ID	Compiler error message	Description
s1	; ( ) [or] expected, illegal start of expression, reached end of file while parsing	Forget or add extra symbols such as semicolons, brackets and braces.
s2	’else’ without ’if’	Forget or add extra brackets in an if clause.
s3	not a statement	incomplete statement, missing + symbol in a println statement etc.
s4	unclosed string literal	String literals that does not enclosed in quotation marks.

Table 4.4.1: Symbol related errors of Classfix

Words related errors		
ID	Compiler error message	Description
w1	class interface or enum expected	Omit or misspell the keyword class or interface.
w2	cannot find symbol	Miss variable type. Use wrong cases.
w3	invalid method declaration	Omit the keyword void or return type.
w4	missing return statement/value	Forget return statement or value from a non-void method.
w5 w6	incompatible types, unexpected return value,	Return from a void method.
w7 w8	illegal character, <identifier> expected	Contains illegal character other than letter and underscore. Or, variable name begins with number or symbols.
w9	variable is already defined	Re-define the same variable name or misspell the keyword return in a return statement.
w10	incompatible types/inconvertible types	Declare wrong variable type, Convert variable of type A to an inconvertible type B.

Table 4.4.2: Word related errors of Classfix

Chapter 5

Training Methodology

In this section, we introduce the training methodology for Classfix. Particularly, we introduce the data noising strategies for generating training data for Classfix. The implementation of the noising method can be found at https://github.com/lingqi1219/Classfix.

5.1 Training Data Generation

The major challenge of Classfix is lacking of sufficient amount of real-world samples. The transformer model like RoBERTa requires a large training set for each error category to be able to get a high repair accuracy.

Prior works [62, 63] employ data noising strategy to obtain a sufficient amount of training data. Inspired by these studies, Classfix uses compilable code from the dataset and noises them into buggy programs.

To correspond the two aforementioned error types in section 4.4, we now introduce the two following data noising strategies.

5.1.1 Noising for Symbol errors

The noising procedure performs atomic modification operations on the chosen line. Each line will perform at most 4 atomic modifications.

The method will choose one line code and randomly select a symbol to modify. There are 4 different operations that can be applied to the selected symbol Table 5.1.1. The

insertion operation appends one random symbol to the target location. The deletion operation removes the chosen symbol. The alteration operation changes the selected symbol and replaces it with other symbols. The duplication duplicates the symbol at the target position.

Operation	Description
Insertion	Insert a random character to the index position.
Deletion	Remove the selected character.
Alteration	Replace the character with another random character.
Duplication	Duplicate the selected character.

Table 5.1.1: Noising operations for symbol related errors

As the example in Table 5.1.2, the method can select and change the line as the following.

	Example
	System.out.println(”Hello World”);
Insertion	System.,out.println(”Hello World”);.
Deletion	Systemout.println(”Hello World”)
Alteration	System.out,println(”Hello World”),
Duplication	System.out..println(”Hello World”);;

Table 5.1.2: Example operation of symbol errors

5.1.2 Noising for Word errors

To generate word related errors, the method takes a line that contains at least one word to perform different operations: 1)modify the word, 2)delete the word, and 3)change type(e.g String or int).

For word errors, the noising method performed at most 4 operations per sample5.1.3.

Operation	Description
Modification	Perform atomic actions on a selected word.
Deletion	Delete the word.
Change type	Change word's type e.g Int,Integer String.

Table 5.1.3: Noising operations for 'word' errors

The word modification method has four different atomic operations: insertion, deletion, alteration or duplication. For a word of length L, the atomic operation will be performed at most L/2-1 times.

As shown in Table 5.1.4, the word modifier selects the following line and operates on word println. Four possible modifications has been shown as the following.

	Example
	System.out.println("Hello World");
Insertion	System.out.printlzn("Hello World");
Deletion	System.out.prntln("Hello World");
Alteration	System.out.PRint%n("Hello World");
Duplication	System.out.priintlln("Hello World");

Table 5.1.4: Example operations of 'word' errors

The noising method creates two sets for training the classification task and the generation task.

For the classification task, the training data for each label are uniformly distributed. All labels have the same amount of training cases and are shuffled before training. The output of the classification model is one of the 21 labels: the "no error found" label, and a number in the range of $1 - 20$.

When creating the training data for the abstraction optimized models, some words have been abstracted before passing to the tokenizer follows the abstraction method.

For the sequence generation task, the input is a paragraph consisting of the localized buggy line and its surrounding context lines. The output is the fixed-line, which is used to replace the buggy line.

5.2 Pre-training

Classfix pre-trains a RoBERTa model on a large java code corpus consisted from the BlueJ Blackbox dataset [10]. The pre-training uses the masked language modelling objective described in the background sections; Which the model randomly masks 15% of the words from a sentence and predicts the missing words. The pre-training follows the original RoBERTa paper, and uses the cross-entropy loss on predicting the masked tokens [31].

In this study, two RoBERTa checkpoints are trained. The first checkpoint is pre-trained on the unmodified java program. The second checkpoints pre-trained on the abstracted java program, that replaces some words with 'M' tokens follows the abstraction method described in the earlier section.

5.3 Supervised training

Apart from the pre-training, our study uses 1 million java programs from Blackbox[10]. We select 500,000 samples that are compilable and noise them into buggy program to generate category specific training data. The rest of 500,000 programs from the dataset are used to pre-train the RoBERTa checkpoints.

The noising method noises each sample from the dataset 10 times and obtain a training set of 5,000,000 broken programs. These 5,000,000 programs are further divided into 4,500,000 for the training, 500,000 for the validation.

Chapter 6

Experimental methodology

6.1 Research questions

- RQ1(End-to-End Effectiveness): How does Classfix's effectiveness compare with Synfix?
- RQ2(Localization only): How accurate is $Classfix_{classification}$ at localizing the error line?
- RQ2(Localization Optimization): To what extent can our abstraction optimization improve the localization of symbol errors?
- RQ4(Repair only): How good is $Classfix_{generation}$ at repairing errors when assuming perfect localization?
- RQ5 (Impact of context lines): How do the different sizes of context lines affect the repair rate?

6.2 Methodology for RQ1

The first research question is aimed at measuring the effectiveness of our model with other state-of-the-art on repairing java errors. In this study, we use Synfix [2] as the baseline for the evaluation. Synfix is a cutting edge approach approach to repairing Java programs and has achieved a significant performance gains over Santos [48], Deepfix [18], SequencerR [12], BF+FF [1].

Thanks to the help of the author of Synfix, we have received the file containing the

source code of Synfix. We leave the source code intact and evaluates their models on tests consisting of 2k of real-world samples. These errors were further broken down into word errors and symbol errors, each containing 1k samples, separated by an algorithm. For each pair of the error line and the man-made fix, the algorithm uses Regex to extract the symbols and words from the sentences. If the word from the source and the target line are identical but having different symbols. It will be classified as a symbol error. If the symbol is identical, but the words are different. The sample will be classified as a word error. If the sample is out of scope and the symbols and words differ at the same time. This sample will not be used for the evaluation.

A proper patch meets the following two criteria: 1) the buggy line is located and 2) the bug is properly repaired.

The study compares the corrections generated by the model with those generated by humans and uses two evaluation metrics. 1)the compile rate and 2)the full repair rate.

If the resulting patch is compiled but is not identical to the patch made by humans, we consider it a compiled patch. If the model-generated patch is identical to the human-generated patch, we consider it as a fully repaired case.

The evaluation measure has the following formula.

$$\text{CompileRateLocalized} = \frac{\text{Number of localized and compilable samples}}{\text{Total number of samples}}$$

$$\text{RepairRateLocalized} = \frac{\text{Number of localized and full repaired samples}}{\text{Total number of samples}}$$

Classfix outputs 3 patches for each suspicious line localized by $Classfix_{classification}$. If the returned patches contain a patch identical to the human-generated patch, this sample is categorized as fully repaired. To localize the error, the section steps have been set to 3, which means that the section window moves 3 lines per iteration. Our classifier retrieves a list of suspicious lines, if the target line is included in the list, we count as a successfully localized sample.

6.3 Methodology for RQ2

The objective of the second question is to measure the effectiveness of our localization model to localize error across different categories. Moreover, the study wants to learn to what extent mask optimization can improve localization.

The study uses the same set of tests used in research question 1 and compares the localization accuracy with and without using the optimization.

Localization metrics can be described as follows.

$$\text{Localization accuracy} = \frac{\text{Number of successfully localized samples}}{\text{Total number of testing samples}}$$

The error is successfully localized if the target error line is in the set returned by the classifier.

6.4 Methodology for RQ3

The third research question is connected to the abstraction optimization of localizing symbol errors. The study aims to measure the effectiveness of using abstraction optimization on symbol error localization. The abstraction optimization has been introduced in the method section. This optimization is designed to mitigate the noise of programmers programming style by masking rarely seen words.

For the evaluation of this optimization technique, the study compares the localization accuracy of the abstracted classifier with the non-abstracted classifier on the same test set used in RQ1. Since we have trained two different models: The abstracted and the non-abstracted. We can can complement their results and have two attempts instead of one, which maximizes the likelihood that at least one of them is right. If either one of the models has located the line, we can count it as a successfully localized sample.

This simply illustrates an idea of how we can use the abstracted model to improve the result the non-abstracted model.

6.5 Methodology for RQ4

In this RQ, we examine the efficiency of repairing the error line beyond the effect of fault localization. The localization method may not be able to locate the error. The best way to evaluate the model of $Classfix_{generation}$ is to assume the perfect localization.

Therefore, the fourth research question examines the line repair precision of our model encoder-decoder assuming perfect localization. The study measures the compile rate and the full repair rate for each sample line using the following formula.

$$\text{CompileRatePerfect} = \frac{\text{Number of successfully compilable samples}}{\text{Total number of samples}}$$

$$\text{RepairRatePerfect} = \frac{\text{Number of successfully full repaired samples}}{\text{Total number of samples}}$$

Our approach outputs 3 patches from models using different context lines. If an output patch is identical to the human-generated patch, the sample can be considered a fully repaired one. In addition, if the models produce at least one compileable patch, the sample is counted as compileable.

6.6 Methodology for RQ5

The final research question continues study the correctness of line repair and measures the effect of using different sizes of the context line. We want to assess the effectiveness of using a variety of contextual line sizes. The study compares the full-repair rate as well as the compile rate for each of these models assuming the perfect localization.

The sequence generation process is sensitive to input noises. Keeping the data simple with fewer contextual lines minimizes the complexity of the task. However, some bugs require to have contextual information to fix, such as missing parentheses and missing variables.

As mentioned earlier in the methods section, we are investigating three models using different context line sizes. The model with 4 contextual lines has the purpose of correcting the local error. The 10- and 20-line models deal with more complex scenarios that require contextual understanding.

For the final research question, the study considers the same evaluation parameters used in the preceding questions.

Chapter 7

Result

7.1 Results for RQ1: End-to-End Effectiveness

7.1.1 Symbol errors

The 7.1.1 table demonstrates the repair accuracy of Classfix and Synfix when repairing the symbol error. The first column refers to the name of the approach. The second and third columns indicate the corresponding full repair rate and compile rate. As shown in the first row, Classfix has successfully located 97.0% of the errors, 71.2% of the samples are fully repaired and 75.5% are compiled. In comparison, Synfix corrects symbol errors with a full repair rate of 77.7% and a compile rate of 84.6%.

Classfix and Synfix have a similar level of full repair rate on repairing the symbol error. There is a small difference in our total repair rate, and Synfix has reached a higher compile rate which is 8.8% higher than our approach.

	Full repair	Compile rate
Classfix	71.2% (712/1000)	75.5% (755/1000)
Synfix	77.7% (777/1000)	84.6% (846/1000)

Table 7.1.1: Repair accuracy for symbols error

The study examines failure cases and finds that Classfix mainly fails in the generation steps. Based on our observation, there is a chance that the decoder randomly change certain variable names to a more common name. This behaviour causes many build

errors and explains the reason that we obtain a lower compilation rate. The decoder may introduce other random errors which corrupt the line.

As presented in table7.1.2, The first column gives the source buggy code, the second column gives the patch generated by the Classfix model and the third column is the patch generated by human. In the first example, Classfix has successfully repaired the symbol error. However, at the same time, the generation model changes the word 'dayOfmonth' to 'monthOfmonth' causing another build error and fails. In the second example, Classfix replaces the name 'Carole' with 'Carcar' and breaks 'David' into two parts. Although the program still compiles, it is not identical to the human generated patch.

Besides the noise generated in the decoding process, the model may not repair long sentences that involve many symbols. Especially when it comes to mathematical functions or combined lines such as examples 3 and 4. The third example combines two lines into one. The target row contains a for-loop statement and a println statement on the same row. The sentence is long and has a rarely seen way to code making our model fail. In the fourth example, people attempts to produce a long concatenation that involves multiple words and symbols. Our model fails due to the complexity.

	buggy code	Classfix generated patch	Human intended patch
1	System.out.print (”In 100 days it will be” + dayOfweek + ”,” month dayOfmonth + year);	System.out.print (”In 100 days it will be” + dayOfweek + ”,” + month +monthOfmonth + year);	System.out.print (”In 100 days it will be” + dayOfweek + ”,” + month + dayOfmonth + year);
2	String[] FirstNames = {”alice” , ”bob” , ”carole” david”};	String[] FirstNames = {”alice” , ”bob” , ”carcar” ,”d” , ”avid”};	String[] FirstNames = {”alice” , ”bob” , ”carole” , ”david”};
3	for(int b= 1; b <= 10; b++){ System.out.println (3 + ”*” + b + ”=” + 3*b }	for(int b= 1; b <= 10; b++){ System.out.println (3 + ”*” +b ”=” + 3*b);}	for(int b= 1; b <= 10; b++){ System.out.println (3 + ”*” + b + ”=” + 3*b);}
4	return ”it is ” + color + ” , has ” + GB + ” GB ” + ” , and it cost me ” + cost + ” .”;	return ”it is ” + color + ” has ” + GB + ” GB ” + ” and ” + it cost me ” + cost + ”.	return ”it is ” + color + ” , has ” + GB + ” GB ” + ” , and it cost me ” + cost + ” .”;

Table 7.1.2: Repair example of symbol errors

7.1.2 Word errors

For word errors as shown in table7.1.3, 46.1% of the samples are fully repaired and 64.3% of the samples have got a compilable patch. In comparison, Synfix fixes symbol errors with a full repair rate of 51.3% and a compilation rate of 76.8%.

	Full repair		Compile rate	
Classfix	46.1%	(461/1000)	64.3%	(643/1000)
Synfix	51.3%	(513/1000)	76.8%	(768/1000)

Table 7.1.3: Repair accuracy for word error

The word error repair rate is lower than symbol errors because many word errors require contextual information and dependency reasoning. With regard to symbol errors, word errors are more difficult to correct, any word of the sentence can be changed randomly, misspelled or missing. The sequence of the word may be wrong and the misspelled word may be misleading. These uncertainties complicate the repair task and the model must understand the context to correct these errors. As the result, both approaches do not record higher repair rate for this category.

The same random noise problem persists and becomes more common in this category, leading to a low build rate. As shown by the first example in the table7.1.4, the words 'hasNext' and 'interest' were modified, causing a build error. In the third example, variable 'arrgs' had been changed to 'args'. The model may think that it encounters a misspelled word because 'args' was more commonly used in the main function, while it is not the case.

	buggy code	Classfix generated patch	Human intended patch
1	(itr.hasNext())	while(itr.next())	while (itr.hasNext())
2	System.out.prin (" A good question " + interest);	System.out.println (" A good question " + interesting);	System.out.println (" A good question " + interest);
3	public static void main (Stringarrgs[])	public static void main (String args[])	public static void main (String arrgs[])

Table 7.1.4: Repair example of word errors

There are a few things that make our model less precise than Synfix. Firstly, the models

are formed on the noisy data which may differ from the actual test cases. This issue has also been identified in the assessment of the classification model. For word errors and symbol errors, there are large deviations between test results and validation results, indicating the fault of our noisy method. In contrast, Synfix is formed on a larger data set consisting of actual java errors. The use of the real world java program allows them to learn about human errors and are able to predict more precisely.

Second, the encoder-decoder model frequently causes additional errors in the sequence generation steps. Unlike Synfix which uses an action-based line repair technique, the sequence generation does not have any modification rules. As a result, the generation model has an unexpected behaviour, such as changing symbols and irrelevant words. Our model fails to repair primarily due to these unexpected behaviours. In contrast, Synfix employs an action-based repair technique and performed insertion, removal and modification to repair the target line. Although Synfix does not consider the contextual lines, it still achieves a decent repair rate.

Lastly, Classfix is the computational heaviest. The program performs a complete scanning to locate the error line and generates 3 patches for each suspicious line. This may be expensive when there are multiple suspicious lines and errors in the file.

Although Classfix does not surpass Synfix, It has some great features that Synfix does not have. Synfix has a token size restricted to 1000. As described in the Synfix document[2], their overall accuracy is reduced as the length of the document increases. They achieve a full repair rate of 82.3% and 90.2% compile rate repair files between 0-100 tokens. However, it decreases to 55.0% and the build rate decreases to 60.8% for files between 900-1000 tokens. As the comparison, Classfix offers consistent performance independent of document length. A larger document may take longer to repair, but has the same level of precision, as long as errors can be located in a section.

Also, because Classfix has trained several models, the program can produce several suggestions on a single line. The user can always choose the one that works best for him.

Answer to RQ1: Although our approach does not exceed Synfix, it has achieved the same level of repair rates. The result shows that the classfix model has the potential to achieve the state of the art by further optimising. The outcome also indicates that the mechanisms used in this study are applicable and may be used

in a subsequent study.

Two main issues make Classfix less accurate and need improvement: 1) Dealing with unusual words and patterns, 2) Random noises that have been introduced into the decoding process.

7.2 Results for RQ2: Localization only

The second research question is to determine if the RoBERTa classification model is able to find the error line. The classification models are trained on the noised buggy codes described in the method section. A 10% subset is created with the same pattern as the training set. This subset is used to validate. In the validation set, the models obtained an average prediction accuracy of 96.3% for symbol errors and 92.1% for word errors average per section7.2.1.

On the test bench, the localization accuracy is slightly lower. The table 7.2.1 provides the exact value. The first column refers to the type of error. The second column gives the mean prediction accuracy if the target row was in the section. The third and fourth columns show the localization accuracy when moving the window with different step sizes.

For the symbol error, the mean prediction precision over a single section is 91.7%. When moving with 5 rows, there is 94.4% accuracy to locate the error. When moving with 3 lines, there is a 95.3% accuracy to locate the error.

For word errors, the average prediction accuracy on the single section is 76.7%. Moving the section 5 lines, the final localization accuracy is 83.6%. When moving with 3 lines, the localization precision becomes 84.7%.

	Single section	5 lines	3 lines
Symbols	91.7% (917/1000)	94.4% (944/1000)	95.3% (953/1000)
Words	76.7% (767/1000)	83.6% (836/1000)	84.7% (847/1000)

Table 7.2.1: Localization accuracy of symbol errors and word errors

Classfix achieves a high level of localization accuracy on symbol errors but struggles to localize word errors. The difference between the accuracy of the validation and the accuracy of the test shows the problem of our noising method. It is easier to generate

symbol errors with a similar pattern, but the word errors can be more complex which causes the failure.

Since our model is only trained on the noising training set, not the real-world samples. It may fail in certain rare cases, which explains the discrepancy between the validation and the test.

While Classfix does not work well on word errors, it still fulfills its main purpose: To be able to locate the error when compilers fail to locate. Unlike the symbol error that can sometimes be pointed to an incorrect position due to the unsecured parenthesis. Word errors can easily be found by the parser and display the correct error line. Since the parser can easily catch the correct error line, we do not need to rely on localization to find the word error.

Answer to RQ2: Our localization model makes it possible to obtain an accuracy of 97% on symbol errors. The high localization precision demonstrates the ability of the sliding window idea. On word errors, the model archives 84.7% due to the difference between the noising training set and the real-world errors.

7.3 Results for RQ3: Localization optimization

The abstracted classification models are also trained on the noisy buggy codes with a 10% validation set. On the validation set, the abstracted model achieves a prediction precision of 96.0% with only 0.3% difference lower than the original model.

The abstracted model performs better on real-world samples as indicated in table7.3.1. The first column refers to the variant of the model. The second column gives the precision of the prediction in the single section. The third and the fourth column show the localization rate while moving the window with 3 lines and 5 lines. The abstracted model obtains on average 93.3% precision per section. The localization accuracy is increased to 95.2% when moving 3 lines, and 95.9% when moving with 5 lines.

	Single section	5 lines	3 lines
Original	91.7% (917/1000)	94.4% (944/1000)	95.3% (953/1000)
Abstracted	93.3% (933/1000)	95.2% (952/1000)	95.9% (959/1000)
Combined	94.1% (941/1000)	95.8% (958/1000)	97.0% (970/1000)

Table 7.3.1: Localization accuracy of symbol errors with abstraction

Compares with the original variant, the abstracted variant is less sensitive to the unseen variable and naming. These variables and user comments have been abstracted before passing the input to reduce the noise. However, the abstracted model can fail if the symbol between two words is missing. If a connected symbol is missing that makes the words merging together. To solve this problem, we complement abstracted model with the original models by combining their output sets. At the cost of more false positive, we maximize the likelihood that at least one of them has successfully localized the error. As a result, the combined model had achieved 97.0% accuracy to localize the error.

Answer to RQ3:

The abstracted localization has achieved 95.9% which is 0.6% more accurate than the original model. Combining the result, we get 97.0%, which is 1.7% more precise than the original version. Not only because the original model covers the weaknesses of the abstracted model. Having two different models enables the program to have an extra chance to predict. The non-abstracted model is strong enough to learn about raw programs, but we can still boost it by applying the abstraction.

7.4 Results for RQ4: Repair only

The best way to evaluate the model of $Classfix_{generation}$ is to assume the perfect localization.

Table7.4.1 shows the repair accuracy using the learned localization model (learned FL), and the repair accuracy assuming the perfect localization (perfect FL).

Assuming the perfect localization, the full repair rate has increased by 0.2% and compile rate increases by 0.3% on symbol errors. On word errors, the full repair rate has increased by 0.4% and the compile rate has increased by 0.4%. Since the

classification model and the repair model are trained on the same dataset. If the classification model does not find the error, the sample is less likely to be corrected by the generation model.

		Full repair	Compile rate
learned FL	Symbol Errors	71.2% (712/1000)	75.5% (755/1000)
perfect FL	Symbol Errors	71.4% (714/1000)	75.8% (758/1000)
learned FL	Word Errors	46.1% (461/1000)	64.3% (643/1000)
perfect FL	Word Errors	46.5% (465/1000)	64.7% (647/1000)

Table 7.4.1: Line repair accuracy assuming perfect localization

Answer to RQ4: We obtain nearly the same precision with or without the perfect localization. This result implies that our failures were primarily caused by the generation model rather than the localization, which is consistent with our earlier speculations.

7.5 Results for RQ5: Impact of context lines

The process of generating sequences is sensitive to input noise. Keeping input simple with fewer context lines reduces the complexity of the task and has the potential to improve performance. Classfix has trained 3 models separately for word and symbol errors. Each model used a different size of contextual lines in the range of 4, 10 and 20. For a simpler illustration, the encoder-decoder model for the symbol error with 4 rows of context is designated as s4. Models consisting of 10 and 20 context lines are referred to as s10 and s20 Table 7.5.1 .

	Full repair	Compile rate
s4	68.3% (683/1000)	74.9% (749/1000)
s10	64.3% (643/1000)	69.2% (692/1000)
s20	60.3% (603/1000)	64.0% (640/1000)
s combined	71.4% (714/1000)	75.8% (758/1000)

Table 7.5.1: Line repair accuracy for symbols

The symbol models have been evaluated on the validation set created by the nosing method. s4 has the best assessment result and records a full repair rate of 84.7%. While

s10 records a rate of 82.3% and s20 obtains a full repair rate of 80.8%.

The models are less precise when assessed on real cases, as shown in the table7.5.1. s4 achieves a 68.3% full-repair rate, s10 and s20 achieves a 64.3% and a 60.3% respectively.

In addition to the full repair rate, s4 obtains a compile rate of 74.9%, s10 archives a compile rate of 69.2% and s20 gets a compile rate of 64.0%. Classfix designs to use s4 aims to correct the local error and use s10 and s20 to capture long-ranged dependencies. If one of the returned patches is compile or full-repaired, we count is a repaired sample. Combining these results together, Classfix achieves a full repair rate of 71.4% and a compile rate of 75.8%.

	Full repair		Compile rate	
w4	41.6%	(416/1000)	56.9%	(569/1000)
w10	37.8%	(378/1000)	51.2%	(512/1000)
w20	36.0%	(360/1000)	51.0%	(510/1000)
w combined	46.5%	(465/1000)	64.7%	(647/1000)

Table 7.5.2: Line repair accuracy for word errors

Similar to word errors, Classfix has trained 3 models with different sizes of denoted context lines w4, w10 and w20[7.5.2]. The validation accuracy of these models is 70.4%, 67.3% and 65.7%. Word errors are more complex, involving long-range dependencies, resulting a lower validation accuracy compared with the symbol errors. In the test bench, w4, w10 and w20 archived full repair rates 41.6%, 37.8% and 36.0%. The compilation rates for those models are 56.9%, 51.2% and 51.0%. Combining results together, we archive a full repair rate of 46.5% and a compile rate of 64.7%.

Answer to RQ5: For both errors, the model with fewer contextual lines has a higher percentage of the repair rate, due to the reduced complexity.

Even with enough contextual information, models 10 and 20 still struggled to fix long-range dependency problems. If the model is not trained exclusively for the dependency problems or used an extremely large training set. It can be difficult to correct these errors, regardless of how many lines we have used.

Instead of using a single model, the result suggests using multiple models

with different size of context lines to get more attempts. If the single model is affordable, keeping the input simple is more important than having more contextual information.

Chapter 8

Conclusion and discussion

8.1 Discussion

8.1.1 Error localization method

There are two versions of classification models implemented in this study. The earlier version takes the entire document as input and outputs the index of the predicted line, as the same as many other studies. It works well on short document but fails on large document. It can localize error with a 96% validation accuracy for codes of less than 50 lines, but the accuracy decreases after line 51. The main reason is due to the input constraint of the RoBERTa model. The RoBERTa model truncates the input and only allows to pass 512 tokens. File that is out of this boundary can not be repaired. Although there are models that make it possible to have larger input sizes, such as Long-Document Transform, real-life programs may still be arbitrarily long and complex.

In addition to the input constraint, we have the problem of an unbalanced distribution for each label. Many programs may be relatively short with a size below 100 lines, which means fewer samples can be assigned to the large index. Obtaining enough data for each label can be expensive. Predicting the minority index over 100 can also be a major problem.

Learned from the first attempt, the second version divides large program into parts less than 20 rows. This allows the model to precisely locate the error in a small section. Splitting into smaller pieces allows the model to use its full potential in manipulating

small program inputs, also solves the problem of input constraint.

Another characteristic of Classfix is to use the classification model and the encoder-decoder model separately. For instance, it is possible to use Classfix to locate the error and use Synfix to fix the target line. Alternatively, use other localization methods to localize the error line and use Classfix to repair the target line.

Lastly, the method described in this study corrects all suspicious lines returned by the classifier which uses a lot of computational resource. Alternatively, the classifier can keep the probability distributions that indicates the likelihood of being the error line and calculate n most suspicious lines.

8.1.2 Abstraction

The abstraction method requires improvement. The current version treats every word and number exactly the same way, substituting them with the same token. However, we can to do better by index them with different characters. Instead of replacing them with M', we can replace them with their main character etc.

Apart from that, the abstraction can also be used in the encoder-decoder model. Especially during the repair of symbol errors that do not need to change words. A big issue of the encoder-decoder is that it randomly changes some words; by abstracting these variables, the model should be able to concentrate on the affected area and become less sensitive to the noise.

8.1.3 Encoder decoder as the generation model

The generation model is able to do the job, but is sensitive to noise and may not repair samples with uncommon patterns. The study have a several suggestions to solve this problem.

We can constrain the decoder such that it only uses words that can be found in the same file.

laso, We can clean the line and restructure the line to reduce the noises. Synfix trimmed the buggy file nicely by removing duplicate spaces and comments which is great to have in future studies.

We still have not fully utilized the potential of the generation model. Much more needs

to be done to make it better. Even without any optimization, we can still achieve a decent result.

8.1.4 Context lines

The output result is primarily determined by the pattern in the training set rather than the context information. Even with enough contextual information, the model still struggles to solve the long-range dependency problem.

Although the single model with more contextual lines is not producing good results. It is possible to improve performance through cooperation. The result from one single model may not be reliable, having multiple models with different sizes of contexts helps to mitigate the problem.

8.1.5 Threats to Validity

Our model can only fix a single line and can be only be applied to either word or symbol error. Many programs require fixing multiple lines dependently involving both symbol and word errors. Our approach may not generalize well on these cases.

All algorithms are implemented in Python using machine learning libraries such as Numpy, Pandas, PyTorch and Huggingface. These libraries have been reviewed and tested by many researchers. However, there is no guarantee that our implementations in the study are devoid of error.

Also, the dataset that is used in this study is mainly based on novice programmers' assignments. Professional programmers may use a lot of external packages, which may not exist in the training data.

8.2 Future work

In terms of the location of errors, the study suggests a continuous study of the transformer-based classification model to locate errors. The high precision in locating smaller inputs has demonstrated the potential of this mechanism. For programs with multiple errors, a multi-class classification model can be a great way to locate multiple errors.

The encoder and decoder model is not reliable to generate patches with no additional optimization. They are sensitive and need lots of training data for this to work with out further optimization. Future studies should improve it or use an alternative technique to repair the line.

Finally, future studies can keep experimenting with different section sizes, try some other error categories and use other programming languages.

8.3 Conclusion

In conclusion, the RoBERTa classification model performs well on localizing Java errors. We propose an idea of localizing error in a section and achieve 95.3% localization accuracy on symbol errors and 84.7% on word errors. Using the abstraction optimization to abstract irrelevant variables, we improve the localization rate of symbol errors to 97%.

Using the learned localization model. the full repair rate and compile rate of symbols errors are 71.2% and 75.5%. The full repair rate and compile rate of words errors are 46.1% and 64.3%.

Our study suggests to train multiple models using different sizes of context lines as input. When dealing with a local error that does not require context reasoning, the model can preserve a minimum amount of input. When dealing with a complex error, the model will have enough information for the reasoning. It also allows the model to have multiple attempts instead of one, which maximizes the chance to fix the program.

Bibliography

[1] Ahmed, Toufique, Devanbu, Premkumar, and Hellendoorn, Vincent. "Learning lenient parsing typing via indirect supervision". In: *Empirical Software Engineering* 26 (Mar. 2021). DOI: 10.1007/s10664-021-09942-y.

[2] Ahmed, Toufique, Ledesma, Noah Rose, and Devanbu, Premkumar T. "SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics". In: *CoRR* abs/2104.14671 (2021). arXiv: 2104.14671. URL: https://arxiv.org/abs/2104.14671.

[3] Ahmed, Umair Z., Kumar, Pawan, Karkare, Amey, Kar, Purushottam, and Gulwani, Sumit. "Compilation Error Repair: For the Student Programs, from the Student Programs". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 78–87. ISBN: 9781450356602. DOI: 10.1145/3183377.3183383. URL: https://doi.org/10.1145/3183377.3183383.

[4] Allamanis, Miltiadis, Barr, Earl T., Bird, Christian, and Sutton, Charles. "Learning Natural Coding Conventions". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 281–293. ISBN: 9781450330565. DOI: 10.1145/2635868.2635883. URL: https://doi.org/10.1145/2635868.2635883.

[5] Arcuri, Andrea. "On the Automation of Fixing Software Bugs". In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion '08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 1003–1006. ISBN: 9781605580791. DOI: 10.1145/1370175.1370223. URL: https://doi.org/10.1145/1370175.1370223.

[6] Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *CoRR* abs/1409.0473 (2015).

[7] Berabi, Berkay, He, Jingxuan, Raychev, Veselin, and Vechev, Martin. “TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 780–791. URL: https://proceedings.mlr.press/v139/berabi21a.html.

[8] Bhatia, Sahil and Singh, Rishabh. “Automated Correction for Programming assignments in Programming assignments using Recurrent Neural Networks”. In: *ArXiv* abs/1603.06129 (2016).

[9] Bhatia, Sahil and Singh, Rishabh. “Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks”. In: *CoRR* abs/1603.06129 (2016). arXiv: 1603.06129. URL: http://arxiv.org/abs/1603.06129.

[10] Brown, Neil Christopher Charles. “Introduction to Analysing the BlueJ Blackbox Data
(Abstract Only)”. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. Atlanta, Georgia, USA: Association for Computing Machinery, 2014, p. 748. ISBN: 9781450326056. DOI: 10.1145/2538862.2539012. URL: https://doi.org/10.1145/2538862.2539012.

[11] Chen, L., Pei, Y., and Furia, C. A. “Contract-based program repair without the contracts”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017.

[12] Chen, Z., Kommrusch, S. J., Tufano, M., Pouchet, L., Poshyvanyk, D., and Monperrus, M. “SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair”. In: *IEEE Transactions on Software Engineering* (2019).

[13] Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[14] Ciniselli, Matteo, Cooper, Nathan, Pascarella, Luca, Poshyvanyk, Denys, Penta, Massimiliano Di, and Bavota, Gabriele. “An Empirical Study on the Usage of BERT Models for Code Completion”. In: *CoRR* abs/2103.07115 (2021). arXiv: 2103.07115. URL: https://arxiv.org/abs/2103.07115.

[15] Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[16] Gazzola, Luca, Micucci, Daniela, and Mariani, Leonardo. “Automatic Software Repair: A Survey”. In: *IEEE Transactions on Software Engineering* (2017).

[17] Ghanbari, Ali, Benton, Samuel, and Zhang, Lingming. “Practical Program Repair via Bytecode Mutation”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 19–30. ISBN: 9781450362245. DOI: 10.1145/3293882.3330559. URL: https://doi.org/10.1145/3293882.3330559.

[18] Gupta, Rahul, Pal, Soham, Kanade, Aditya, and Shevade, Shirish. “DeepFix: Fixing Common C Language Errors by Deep Learning”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI’17. San Francisco, California, USA: AAAI Press, 2017, pp. 1345–1351.

[19] Hajipour, Hossein, Bhattacharyya, Apratim, Staicu, Cristian-Alexandru, and Fritz, Mario. *SampleFix: Learning to Generate Functionally Diverse Fixes*. 2021. arXiv: 1906.10502 [cs.SE].

[20] Hellendoorn, Vincent J., Sutton, Charles, Singh, Rishabh, Maniatis, Petros, and Bieber, David. “Global Relational Models of Source Code”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=B1lnbRNtwr.

[21] Ji, Tao, Chen, Liqian, Mao, Xiaoguang, and Yi, Xin. “Automated Program Repair by Using Similar Code Containing Fix Ingredients”. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2016, pp. 197–202. DOI: 10.1109/COMPSAC.2016.69.

[22] Jiang, Jiajun, Xiong, Yingfei, Zhang, Hongyu, Gao, Qing, and Chen, Xiangqun. “Shaping Program Repair Space with Existing Patches and Similar Code”. In: ISSTA. Amsterdam, Netherlands, 2018.

[23] Jiang, Nan, Lutellier, Thibaud, and Tan, Lin. “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (May 2021). DOI: 10.1109/icse43902.2021.00107. URL: http://dx.doi.org/10.1109/ICSE43902.2021.00107.

[24] Jiang, Nan, Lutellier, Thibaud, and Tan, Lin. “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair”. In: *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*. 2021.

[25] Le, Xuan Bach D, Lo, David, and Le Goues, Claire. “History driven program repair”. In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE. 2016, pp. 213–224.

[26] Le, Xuan-Bach D., Chu, Duc-Hiep, Lo, David, Le Goues, Claire, and Visser, Willem. “S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. 2017.

[27] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. “GenProg: A Generic Method for Automatic Software Repair”. In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 54–72. DOI: 10.1109/TSE.2011.104.

[28] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. “GenProg: A generic method for automatic software repair”. In: *Software Engineering, IEEE Transactions on* 38.1 (2012), pp. 54–72. DOI: 10.1109/TSE.2011.104.

[29] Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, and Zemel, Richard. *Gated Graph Sequence Neural Networks*. 2017. arXiv: 1511.05493 [cs.LG].

[30] Liu, Kui, Wang, Shangwen, Koyuncu, Anil, Kim, Kisub, Bissyandé, Tegawendé F., Kim, Dongsun, Wu, Peng, Klein, Jacques, Mao, Xiaoguang, and Traon, Yves Le. “On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software*

Engineering. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 615–627. ISBN: 9781450371216. DOI: 10.1145/3377811.3380338. URL: https://doi.org/10.1145/3377811.3380338.

[31] Liu, Yinhan, Ott, Myle, Goyal, Naman, Du, Jingfei, Joshi, Mandar, Chen, Danqi, Levy, Omer, Lewis, Mike, Zettlemoyer, Luke, and Stoyanov, Veselin. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL].

[32] Long, Fan and Rinard, Martin. "Automatic Patch Generation by Learning Correct Code". In: *SIGPLAN Not.* 51.1 (Jan. 2016), pp. 298–312. ISSN: 0362-1340. DOI: 10.1145/2914770.2837617. URL: https://doi.org/10.1145/2914770.2837617.

[33] Long, Fan and Rinard, Martin. "Staged Program Repair with Condition Synthesis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 166–178. ISBN: 9781450336758. DOI: 10.1145/2786805.2786811. URL: https://doi.org/10.1145/2786805.2786811.

[34] Loriot, Benjamin, Madeiral, Fernanda, and Monperrus, Martin. "Styler: Learning Formatting Conventions to Repair Checkstyle Errors". In: *CoRR* abs/1904.01754 (2019). arXiv: 1904.01754. URL: http://arxiv.org/abs/1904.01754.

[35] Lutellier, Thibaud, Pham, Hung Viet, Pang, Lawrence, Li, Yitong, Wei, Moshi, and Tan, Lin. "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair". In: ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 101–114. ISBN: 9781450380089. DOI: 10.1145/3395363.3397369. URL: https://doi.org/10.1145/3395363.3397369.

[36] Lutellier, Thibaud, Pham, Hung Viet, Pang, Lawrence, Li, Yitong, Wei, Moshi, and Tan, Lin. "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair". In: ISSTA 2020. 2020. ISBN: 9781450380089.

[37] Martinez, Matias and Monperrus, Martin. "ASTOR: A Program Repair Library for Java". In: *Proceedings of ISSTA*. 2016.

[38] Mashhadi, Ehsan and Hemmati, Hadi. “Applying CodeBERT for Automated Program Repair of Java Simple Bugs”. In: *CoRR* abs/2103.11626 (2021). arXiv: 2103.11626. URL: https://arxiv.org/abs/2103.11626.

[39] Mechtaev, S., Yi, J., and Roychoudhury, A. “DirectFix: Looking for Simple Program Repairs”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 448–458. DOI: 10.1109/ICSE.2015.63.

[40] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. “Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 691–701. ISBN: 9781450339001. DOI: 10.1145/2884781.2884807. URL: https://doi.org/10.1145/2884781.2884807.

[41] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. “Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016.

[42] Mesbah, Ali, Rice, Andrew, Johnston, Emily, Glorioso, Nick, and Aftandilian, Edward. “DeepDelta: Learning to Repair Compilation Errors”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering, Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 925–936. ISBN: 9781450355728. DOI: 10.1145/3338906.3340455. URL: https://doi.org/10.1145/3338906.3340455.

[43] Mittal, Varun and Aditya, Shivam. “Recent Developments in the Field of Bug Fixing”. In: *Procedia Computer Science* 48 (2015). International Conference on Computer, Communication and Convergence (ICCC 2015), pp. 288–297. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.04.184. URL: https://www.sciencedirect.com/science/article/pii/S1877050915006936.

[44] Monperrus, Martin. “Automatic Software Repair: a Bibliography”. In: *ACM Computing Surveys* 51 (2017), pp. 1–24. DOI: 10.1145/3105906. URL: https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf.

[45] Parr, Terence and Vinju, Jurgen. “Towards a Universal Code Formatter through Machine Learning”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 137–151. ISBN: 9781450344470. DOI: 10.1145/2997364.2997383. URL: https://doi.org/10.1145/2997364.2997383.

[46] Raffel, Colin, Shazeer, Noam, Roberts, Adam, Lee, Katherine, Narang, Sharan, Matena, Michael, Zhou, Yanqi, Li, Wei, and Liu, Peter J. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

[47] Rothe, Sascha, Narayan, Shashi, and Severyn, Aliaksei. *Leveraging Pre-trained Checkpoints for Sequence Generation Tasks*. 2020. arXiv: 1907.12461 [cs.CL].

[48] Santos, Eddie Antonio, Campbell, Joshua Charles, Patel, Dhvani, Hindle, Abram, and Amaral, José Nelson. “Syntax and sensibility: Using language models to detect and correct syntax errors”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 311–322. DOI: 10.1109/SANER.2018.8330219.

[49] See, Abigail, Liu, Peter J., and Manning, Christopher D. *Get To The Point: Summarization with Pointer-Generator Networks*. 2017. arXiv: 1704.04368 [cs.CL].

[50] Sennrich, Rico, Haddow, Barry, and Birch, Alexandra. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://www.aclweb.org/anthology/P16-1162.

[51] Seo, Hyunmin, Sadowski, Caitlin, Elbaum, Sebastian, Aftandilian, Edward, and Bowdidge, Robert. “Programmers’ Build Errors: A Case Study (at Google)”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 724–734. ISBN: 9781450327565. DOI: 10.1145/2568225.2568255. URL: https://doi.org/10.1145/2568225.2568255.

[52] Sherstinsky, Alex. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network”. In: *CoRR* abs/1808.03314 (2018). arXiv: 1808.03314. URL: http://arxiv.org/abs/1808.03314.

[53] Souza, Higor A. de, Chaim, Marcos L., and Kon, Fabio. *Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges*. 2017. arXiv: 1607.04347 [cs.SE].

[54] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Łukasz, and Polosukhin, Illia. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[55] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Łukasz, and Polosukhin, Illia. “Attention is All You Need”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[56] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, undefinedukasz, and Polosukhin, Illia. “Attention is All You Need”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[57] Veličković, Petar, Cucurull, Guillem, Casanova, Arantxa, Romero, Adriana, Liò, Pietro, and Bengio, Yoshua. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML].

[58] Wu, Liwei, Li, Fei, Wu, Youhua, and Zheng, Tao. “GGF: A Graph-based Method for Programming Language Syntax Error Correction”. In: July 2020, pp. 139–148. DOI: 10.1145/3387904.3389252.

[59] Xiong, Yingfei, Wang, Jie, Yan, Runfa, Zhang, Jiachen, Han, Shi, Huang, Gang, and Zhang, Lu. “Precise Condition Synthesis for Program Repair”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 416–426. ISBN: 9781538638682. DOI: 10.1109/ICSE.2017.45. URL: https://doi.org/10.1109/ICSE.2017.45.

[60] Xuan, Jifeng, Martinez, Matias, Demarco, Favio, Clément, Maxime, Lamelas, Sebastian, Durieux, Thomas, Le Berre, Daniel, and Monperrus, Martin. “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs”. In: *IEEE Transactions on Software Engineering* (2016).

[61] Xuan, Jifeng, Martinez, Matias, DeMarco, Favio, Clément, Maxime, Marcote, Sebastian Lamelas, Durieux, Thomas, Le Berre, Daniel, and Monperrus, Martin. “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs”. In: *IEEE Transactions on Software Engineering* 43.1 (2017), pp. 34–55. DOI: 10.1109/TSE.2016.2560811.

[62] Yasunaga, Michihiro and Liang, Percy. “Break-It-Fix-It: Unsupervised Learning for Program Repair”. In: *International Conference on Machine Learning (ICML)*. 2021.

[63] Yasunaga, Michihiro and Liang, Percy. “Graph-based, Self-Supervised Program Repair from Diagnostic Feedback”. In: *International Conference on Machine Learning (ICML)*. 2020.

[64] Ye, He, Martinez, Matias, and Monperrus, Martin. “Neural Program Repair with Execution-based Backpropagation”. In: *Proceedings of the International Conference on Software Engineering*. 2022. URL: http://arxiv.org/pdf/2105.04123.

[65] Yuan, Yuan and Banzhaf, Wolfgang. “ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming”. In: *IEEE Transactions on Software Engineering*. 2018.

[66] Zhivich, Michael and Cunningham, Robert K. “The Real Cost of Software Errors”. In: *IEEE Security Privacy* 7.2 (2009), pp. 87–90. DOI: 10.1109/MSP.2009.56.