Project Report of Automated Program Repair

Sepehr Nourmohammadi Muhammed Ikbal Dogan

Abstract

Modern natural language processing models, such as BART, T5, and GPT-2, are integrated into an automatic program repair system described in this project report to find and fix errors in code snippets. Significant gains in debugging efficiency are made by the system by using a strong framework that integrates tokenization, machine learning, and diff parsing using the Transformers library. PyTorch combined with a CUDA-enabled environment guarantees effective model training and computing. System efficacy is crucially shown by thorough statistical analysis of datasets and performance indicators. All things considered, this work significantly advances the area of automated program repair by implementing NLP techniques to software development.

Code link: https://github.com/sepehrnour1996/Automated-Program-Repair-Project-CS-539-CS-439-.git

I. Introduction

In this paper, a complex code debugging and repair system is presented that combines state-of-the-art natural language processing models, such as BART, T5, and GPT-2, to identify and correct flaws in code sample. Through the processing of changes between broken and fixed versions of code, the approach offers an efficient way to record errors and their related repairs. A unique pipeline driven by the Transformers library combines tokenization, machine learning training, and diff parsing inside a robust architecture.

The main idea is to provide more precise code fixes by using BART for conditional generation and T5 and GPT-2 to improve context comprehension and response generating. By employing a unique accuracy metric, the system assesses its performance and guarantees great dependability in the automatic issue fixing. Because pandas and datasets libraries efficiently manage and preprocess input data, data handling is made easier.

The system also has features for thorough testing and validation, such as a thorough statistical display of dataset properties and outcome metrics, which guarantees thorough input on system performance.This work used two datasets.6176 training samples and 686 validation samples make up the first dataset. We used 106 validation samples and 956 training samples for the second dataset. Overall, minimum, maximum, and mean execution durations are used to assess models. Furthermore, models are compared in terms of steps per second, samples per second, evaluation runtime, and evaluation loss.Effective computing and model training are guaranteed by PyTorch and a CUDA-enabled environment. This study shows notable progress in using natural language processing techniques to software development and not only increases the efficiency of debugging procedures but also makes a major contribution to the field of automated code repair.

The ManySStuBs4J Corpus

The ManySStuBs4J corpus is a meticulously curated collection designed specifically for the evaluation of program repair techniques. This corpus encompasses all bug-fixing changes identified using the SZZ heuristic. Subsequently, these changes are filtered to compile a dataset comprised exclusively of small bug fix changes.

These fixes are limited to single-statement alterations and, wherever applicable, are categorized into one of 16 syntactic templates referred to as SStuBs. The dataset extracts simple statement bugs from a variety of open-source Java projects hosted on GitHub. Notably, the corpus is available in two distinct variants:

- **100 Java Maven Projects**: This variant mines data from 100 Java Maven projects, selected based on their popularity, which is assessed by the sum of z-scores for forks and watchers.
- **Top 1000 Java Projects**: This variant, larger in scope, is derived from the top 1000 Java projects on GitHub, again ranked according to the same popularity metrics.

Specifically eliminating any stylistic modifications like spaces, emptiness, or variations in comments, only commits involving single sentence changes are kept in both datasets. Furthermore, refactorings that could appear to be problem solutions such as renaming variables, functions, and classes—are sought for and eliminated.

argument of functions. To precisely classify the modifications as bug fixes, the commit messages' use of phrases like *bug*, *fix*, or *fault* is verified.

II. Results

Due to GPU limitation (A100 GPU Google Colab) we lower the epochs to 3 only for 5% of the data.

This table I presents the statistics for the first dataset which contains a total of 6,862 samples, divided into 6,176 training samples and 686 validation samples. It also provides examples of code transformations that the automated system attempts to repair. The example given shows a buggy code annotation $@isPositiveNumber$, which is fixed by adding an attribute (includeZero = $true$) to handle zero as a positive number, improving the code's functionality.

TABLE I: Dataset Statistics and Code Examples for first data

Metric	Value
Total Samples	6862
Training Samples	6176
Validation Samples	686
Example of Buggy Code	@isPositiveNumber
Example of Fixed Code	@isPositiveNumber(includeZero = true)

TABLE II: Dataset Statistics and Code Examples for second data

Metric	Value
Total Samples	1062
Training Samples	956
Validation Samples	106
Example of Buggy Code	—final SemaphoreProxy semaphoreProxy = (SemaphoreProxy) factory.getSemaphore(packet.name);—
Example of Fixed Code	—final SemaphoreProxy semaphoreProxy = (SemaphoreProxy) factory.getOrCreateProxyByName(packet.name);—

The statistics for another dataset of 1,062 samples—of which 956 are set aside for training and 106 for validation are presented in table II. The Java example of defective code is the establishment of a semaphore proxy in which the original code obtains a semaphore that might not exist, possibly resulting in a null pointer only one. The updated code makes sure that, should one not already exist, a proxy is produced, hence enhancing code robustness. Both tables demonstrate the useful applications of NLP models in efficiently detecting and fixing problems in code snippets by highlighting the kinds of errors usually found in the datasets and how the automated repair system handles these by producing remedies.

The report's third table III shows the overall, minimum, maximum, and mean execution durations for the BART, T5, and GPT 2 models over two datasets, therefore demonstrating their performance in processing Java programs.

TABLE III: Time execution results (Seconds)

	Data1 (100 Java Maven Projects)				Data2 (Top 1000 Java Projects)			
Tools	Overall	Min	Max	Mean	overall	Min	Max	Mean
BART (Hugging Face)	2639	5.35	5.47	5.40	1810	0.57	0.71	0.61
T5	5626	0.78	10.49	7.23	1762	0.34	11.62	8.33
GPT2	3215	0.19	8.89	0.63	1777	0.20	7.00	0.67

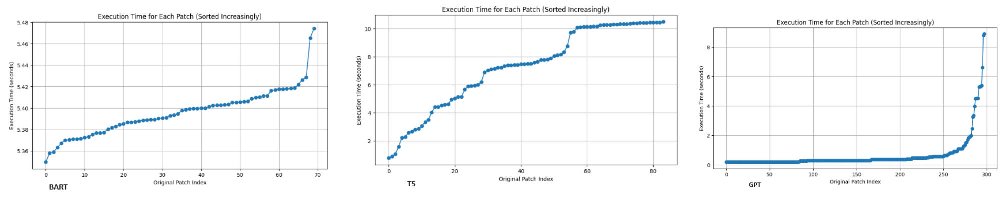


Fig. 1: Execution time on first data set

Evaluation loss, runtime, and throughput for the BART model are highlighted in the fourth tableIV over two datasets. The performance of T5 is described in the fifth tableV, which also displays evaluation loss shown as NaN (not a number) to suggest possible problems like non-convergence or computational mistakes during the evaluation stage. TableVI finally lists the evaluation metrics of GPT-2 coupled with its enhanced handling of evaluation loss and runtime efficiencies between the two datasets. The distribution of patch positions for different tools is displayed in Figures 1 and 2 of the paper on the first and second data sets, respectively.These charts probably show where the tools modified the code, including bug patches, pointing up trends or frequent places where changes take place. This can reveal information on the usual mistakes in the datasets or how well each tool handles various kinds of code problems. The paper's figures 3 and 4 exhibit instances of the BART model-generated patches for the second dataset.

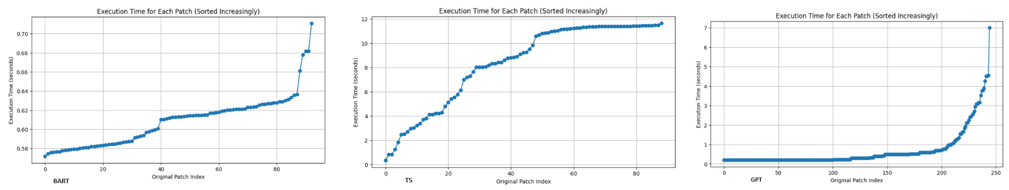


Fig. 2: Execution time on second data set

TABLE IV: Evaluation Metrics for BART

Metric	Value first data	Value second data
Evaluation Loss	1.7128	3.1984
Evaluation Runtime (seconds)	6.4086	1.0016
Samples per Second	107.043	105.827
Steps per Second	13.419	13.977

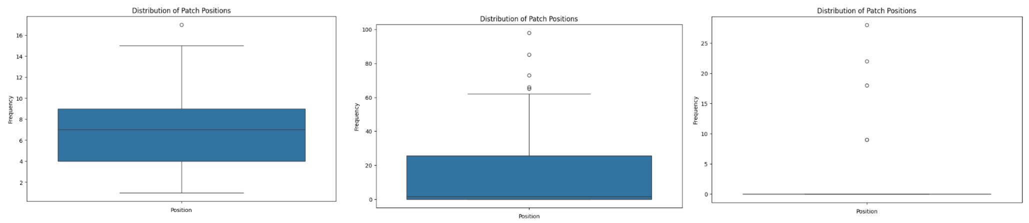


Fig. 3: Distributions of patch positions for different tool on first data

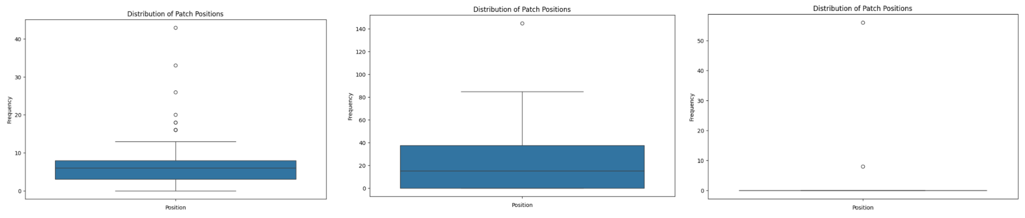


Fig. 4: Distributions of patch positions for different tool on second data

Figures 3 and 4 in the document visualize the distribution of patch positions for different tools on the first and second data sets, respectively. These figures likely illustrate where the tools made changes in the code—such as bug fixes—highlighting patterns or common areas where modifications occur. This can give light on the usual mistakes in the datasets or how well each tool handles various kinds of coding problems. The paper's figures 5 and 6 provide instances of the BART model-generated patches for the second dataset. These figures likely demonstrate the actual

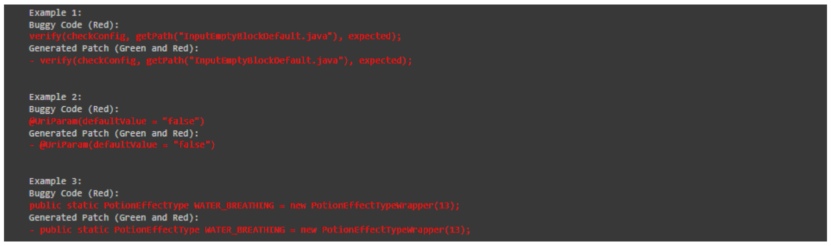
TABLE V: Extended Evaluation Metrics for T5

Metric	Value first data	Value second data
Evaluation Loss	nan	nan
Evaluation Runtime (seconds)	14.0989	2.2031
Samples per Second	48.656	48.114
Steps per Second	6.1	6.355

TABLE VI: Evaluation Metrics for GPT2

Metric	Value first data	Value second data
Evaluation Loss	nan	0.4803
Evaluation Runtime (seconds)	4.1967	0.6895
Samples per Second	163.461	153.73
Steps per Second	20.492	20.304

code changes proposed by BART to rectify identified bugs, providing visual confirmation of the model's effectiveness and the types of corrections it suggests. This can give an idea of the practical applicability of the model in real-world programming tasks and its ability to generate viable solutions for software debugging.



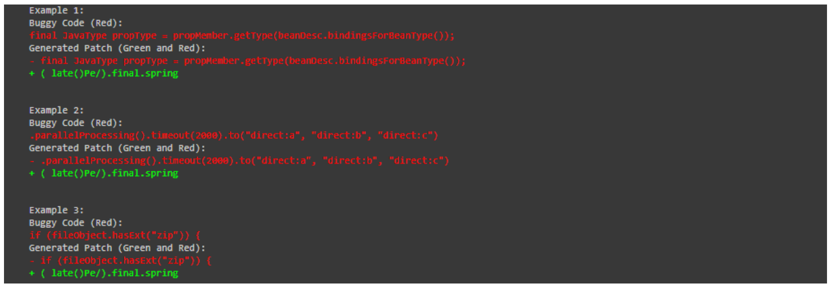
```
Example 1:
Buggy Code (Red):
verify(checkConfig, getPath("InputEmptyBlockDefault.java"), expected);
Generated Patch (Green and Red):
- verify(checkConfig, getPath("InputEmptyBlockDefault.java"), expected);


Example 2:
Buggy Code (Red):
@UriParam(defaultValue = "false")
Generated Patch (Green and Red):
- @UriParam(defaultValue = "false")


Example 3:
Buggy Code (Red):
public static PotionEffectType WATER_BREATHING = new PotionEffectTypeWrapper(13);
Generated Patch (Green and Red):
- public static PotionEffectType WATER_BREATHING = new PotionEffectTypeWrapper(13);
```

Fig. 5: 3 examples of generated patches for the second data according to BART

As can be seen in Fig 5, the results are insufficient and ineffective because we could not fix any bugs due to the limitations we imposed on the models—specifically, training for only 3 epochs with a learning rate of 0.01 and using just 5% of the data. The reason behind these constraints is our limited GPU capacity and memory.



```
Example 1:
Buggy Code (Red):
final JavaType propType = propMember.getType(beanDesc.bindingsForBeanType());
Generated Patch (Green and Red):
- final JavaType propType = propMember.getType(beanDesc.bindingsForBeanType());
+ ( late()Pe/).final.spring


Example 2:
Buggy Code (Red):
.parallelProcessing().timeout(2000).to("direct:a", "direct:b", "direct:c")
Generated Patch (Green and Red):
- .parallelProcessing().timeout(2000).to("direct:a", "direct:b", "direct:c")
+ ( late()Pe/).final.spring


Example 3:
Buggy Code (Red):
if (fileObject.hasExt("zip")) {
Generated Patch (Green and Red):
- if (fileObject.hasExt("zip")) {
+ ( late()Pe/).final.spring
```

Fig. 6: 3 examples of generated patches for second data according to BART

As can be seen in Fig. 6, the results are somewhat sufficient and effective for the second dataset.