

2024

Scheduled Contrastive Loss In Continued Transfer Learning For Software Engineering Tasks

Aaron Michael Harris

College of William and Mary - Arts & Sciences, amharris04@wm.edu

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Harris, Aaron Michael, "Scheduled Contrastive Loss In Continued Transfer Learning For Software Engineering Tasks" (2024). *Dissertations, Theses, and Masters Projects*. William & Mary. Paper 1727787997.

<https://dx.doi.org/10.21220/s2-r118-1j93>

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Scheduled Contrastive Loss in Continued Transfer Learning for Software
Engineering Tasks

Aaron Michael Harris

Williamsburg, VA

Bachelor of Arts, Virginia Commonwealth University, 2006

A Thesis presented to the Graduate Faculty
of The College of William and Mary in Virginia in Candidacy for the Degree of
Master of Science

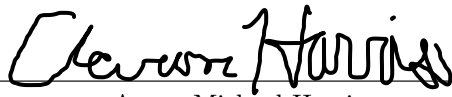
Department of Computer Science

The College of William and Mary in Virginia
August 2024

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science



Aaron Michael Harris

Approved by the Committee, May 2024



Committee Chair

Denys Poshyvanyk, Thesis Advisor, Professor, Computer Science
College of William & Mary



Adwait Nadkarni, Associate Professor, Computer Science
College of William & Mary



Oscar Chaparro, Assistant Professor, Computer Science
College of William & Mary

ABSTRACT

Considerable research has been performed with regard to using text-to-text machine learning methods to perform various software engineering tasks. At the same time, contrastive learning has shown promise in other modalities, such as computer vision-related problems, and has been explored to some extent in terms of limited software engineering tasks. We demonstrate that contrastive loss, on its own, is insufficient to surpass current baselines for these tasks; however, we note that there is a high degree of orthogonality in the results from existing and contrastive models.

We show that when our contrastive method is used as an additional transfer learning step in the training process, the results contain a large portion of the overlap between the distinct models, as well as producing new positive results, effectively capturing the majority of the results from the distinct models and increasing overall model accuracy. By employing this method, we are able to exceed the baseline accuracy of four software engineering tasks by varying margins, ranging from marginal ($<1\%$) to 262% during single beam tests, with minor improvements at selected other beam sizes, in both single- and multi-task training strategies.

TABLE OF CONTENTS

Acknowledgments	iii
Dedication	iv
List of Tables	v
List of Figures	vi
1 Introduction	2
2 Background	5
3 Related Works	9
3.1 Transformers and Contrastive Loss	9
3.2 Transfer Learning	10
3.3 Bug Fixes, Mutant Generation, and Assert Generation	11
4 Methodology	14
4.1 Data Augmentation & Processing	16
5 Results	19
5.1 Research Question 1	19
5.2 Research Question 2	20
5.3 Research Question 3	21
6 Future Work	24

7	Threats to Validity	25
8	Conclusion	27
A	Prediction Examples	29
B	Data Augmentation Examples	35
	Bibliography	39

ACKNOWLEDGMENTS

I would like to extend my deep thanks to Dr. Denys Poshyvanyk for his continuous application of encouragement, expertise, and patience during this research effort. I first learned of this research area through Yanfu Yan, who also guided me through many of the early concepts that led to this stage of research. Additionally, my colleagues Sarah Revillar, Cara Dutil, Jason Cooper, and Dr. Martin White, who have often served as sounding boards while I talked through problems. I would also like to extend my gratitude to the committee members for offering their instruction, time, and support that have enabled this research, and for their consideration of its merits.

To my daughters, Madison and Victoria.

LIST OF TABLES

2.1	Selected samples of task-related pairs.	7
4.1	Summary of dataset size increases using the semantic-preserving augmentation method.	18
4.2	A sample from the BFS dataset, processed through the augmentation and abstraction processes.	18
5.1	Experimental results for single-task models.	22
5.2	Experimental results for the multi-task model.	23
5.3	Overview of overlap analysis (beam size 1)	23

LIST OF FIGURES

4.1	Process diagram for including contrastive loss as an additional learning pass.	14
-----	--	----

Scheduled Contrastive Loss in Continued Transfer Learning for
Software Engineering Tasks

Chapter 1

Introduction

Machine learning methodologies related to software engineering have been a focus of considerable research [30]. In particular, work related to the repair of software bugs, injection of code mutants, assert generation, and code summarization via neural machine translation (NMT) have seen progress via the usage of transformers and transfer learning [17]. While robust testing, both human and automated, can lead to a decrease in the resource requirements for bug repair [1], a method for the automatic repair of bugs is desirable for cost and frustration reduction. These developments are of great importance to the software engineering community; the time and resources dedicated to code repair, for example, are estimated to make up approximately 16% of software development costs [15], and taking a considerable portion of development time usage [18]. In addition, other repetitive software engineering tasks, such as the generation of assert statements, represent an area where an automatic generation method would reduce workloads for developers.

It is important to note that repairing a bug, for a software developer, is not simply replacing the faulty text with the appropriate variation; it is localizing and identifying the bug, determining the appropriate fix, and considering the impact of that fix on the code as a whole. In this research, when we reference code repair or fixing a bug, we are concerned with the corrective actions necessary to transform the bug into fixed code, and not with localization or other concerns. Code mutant injection contributes to security-

related software engineering tasks like test suite creation. This presents its own unique set of challenges, such as determining which mutants are plausible with a given input of code, though it keeps in common the time-related detriments of bug repair. Assert generation can lead to more robust unit testing.

The capability to automatically repair software bugs and generate code mutants, along with myriad other text-to-text software engineering tasks, lends itself well to transformer-based approaches; we frame the problem in the same way one might consider the translation between two human languages [3]. Code, like human language, is structured and carries meaning. As with human language, there is often more than one way to convey a given idea. Consider translation between English and Spanish. It would be insufficient to simply translate each word, one at a time, between the vocabularies of the two. Instead, an effective translator would consider not only the strict word meanings between languages, but their order, grammatical nuances, and other connotations that are not provided by dictionaries alone. When we discuss fixing a bug (or other software engineering tasks) via NMT methods, it is not only appropriate but fundamental to the implementation that we regard this process as *translating* the code from some buggy origin to its fixed alternative, or whatever source and destination configuration is appropriate given the task.

Of particular note and importance to our efforts in this area of research is the overlap between text-based and vision-based problems. Methodologies such as masked auto-encoders, which started in the text domain and made vast improvements in the vision domain, are one such example [12]. In that vein, we seek to employ contrastive learning to improve the accuracy of bug repair and mutant generation tasks. Contrastive loss is highly effective in vision-based tasks [14], and has shown usefulness crossing over into code-related NMT tasks [8].

In this effort, we propose to extend the research of Mastropaolo et al. [17] by changing elements of the loss function and dataset construction. We find that contrastive learning is capable of finding orthogonal results to the baseline, though at a lower rate. Therefore, we use transfer learning to incorporate the benefits of the baseline method and our novel con-

trastive learning with scheduled temperature implementation to create a combined result of greater magnitude than either method independently, capturing a variable proportion of the correct results of the baseline while adding new correct results from the novel method.

Chapter 2

Background

Machine learning provides a method for predicting the solutions to problems. In its most general form, the machine learning model seeks to learn some unknown function, F , by learning the pattern in data provided. The data is provided as a series of input features, X , and learning is the process of training the model to predict a given label or output, Y . Thus the process of machine learning can be given as:

$$F : X \rightarrow Y$$

The process of learning, in a machine learning context, is represented through the adjustment of weights. These weights, or parameters, are stored as a numeric value representing the vector perpendicular to a line or hyperplane, dividing various representations in a space. As new data is introduced while training a model, these vectors are adjusted based on the response of a loss function that determines the difference between the correct answer and the predicted answer via a process called back propagation, causing points in that space to be appropriately grouped together. The particulars of this process are well-known within the research community and are not themselves the topic of our work; it is, however, necessary to state that the selection of a particular loss function is instrumental in determining how a given machine learning model learns from data.

Consider a software bug and a given fix - solutions can be wide-ranging, but in general

the fix is some variation on the bug. For example, an incorrect operator or the improper arrangement of two functions. A bug can have many causes, such as an incorrect implementation of logic, system or compatibility-related issues, or syntax errors. For our discussion of this problem, we refer to these two code segments as m_{bug} and m_{fix} , and seek to find an m_{fix} such that the new code segment no longer produces the erroneous behavior present in the original code segment. We refer to the tuple (m_{bug}, m_{fix}) as a Bug-Fix Pair (BFP), noting that the given m_{fix} is not necessarily the only solution, simply the solution present in the dataset. Conversely, a code mutant can be expressed as the inverse of such a tuple, (m_{fix}, m_{bug}) , given that a correct code segment will be translated into a buggy code segment. For assert generation, the provided pair is represented as (m_{test}, m_{assert}) , as the input and output are not an edit of the input, but instead an appropriate assert derived from the context of the input segment. In each case, the issue of considering a valid fix, mutant, assert as only that which is present in the ground truth of our data is present.

This collection of tasks contains two different patterns of output; bug repair and mutant generation tasks are concerned with editing the input into an output, while assert generation seeks to replace a predetermined tag with the appropriate line of code, eschewing the remainder of the input with regard to the generated text. Table 2.1 provides examples of these tasks.

Within the realm of software engineering tasks, we concern ourselves with both features and targets that both consist of text. For example, in the framework of using machine learning to learn bug repair, we can reframe our machine learning paradigm to state:

$$F : m_{bug} \rightarrow m_{fix}$$

Given these parameters, the goal is to determine a method that effectively translates between the two code segments. Unlike the base example of machine learning, we are seeking to predict a string of outputs that compose the final code segment. In the domain of human language translation, transformers have demonstrated a capability on par with

Task	Sample Input	Sample Output
Bug Repair	<pre>public java.lang.String METHOD_1() { return new TYPE_1 (STRING_1) .format(VAR_1[((VAR_1.length)-1]) .getTime(); }</pre>	<pre>public java.lang.String METHOD_1() { return new TYPE_1 (STRING_1) .format(VAR_1[((type)-1]) .getTime(); }</pre>
Mutant Generation	<pre>public TYPE_1 node() { return this.VAR_1; }</pre>	<pre>public TYPE_1 node () { return this.node; }</pre>
Assert Generation	<pre>setBrokerShareVisible () { m.setBrokerShareVisible(true); "<AssertPlaceholder>"; } isBrokerShareVisible () { return brokerShareVisible; }</pre>	<pre>org.junit.Assert.assertTrue(m.isBrokerShareVisible())</pre>

Table 2.1: Selected samples of task-related pairs.

that of human interpreter output [22]; given the similarity of the problem, it is reasonable to consider the transformer architecture as an appropriate vehicle for our task. In their 2017 work [29], Vaswani et al. introduced the transformer architecture, which has been used to great effectiveness in research related to natural language processing (NLP) and NMT. Specifically, it excels at text translation and adjacent fields. Of particular importance in the architecture is the concept of an attention layer, which focuses on each element of the input sequence to determine its importance to other input sequence elements, providing the model a variety of context when considering those elements. This development gave the architecture an advantage over previous methods, such as Long Short-Term Memory models [13].

The transformer architecture is delineated into an encoder and decoder, which are responsible for learning the formation of latent space representations from the input sequence and generating the output sequence, respectively. The loss function for the encoder and decoder is typically cross-entropy loss, due to its ability to determine the (dis)similarity between the training target and the predicted probability distribution.

Of course, the approach does come with its limitations, in that such an architecture learns from the provided data, and any output produced at test time is restricted to a rearrangement of the tokens which it has learned [28]. Variables and other named elements

in code present a risk of creating a large vocabulary problem. To subvert this issue, we employ code abstraction to reduce the number of idioms present in both training and test. We also note a key difference between the use case of human language translation and that of code repair and mutant generation: in a human language translation, the intent is to convey the same meaning, while in the software engineering tasks presented, it is to correct or break the meaning, or to generate the appropriate assert from its context.

Chapter 3

Related Works

3.1 Transformers and Contrastive Loss

In 2020, Raffel et al. created the Text-to-Text Transfer Transformer (T5) architecture [23]. T5 uses a byte-pair encoding method of tokenization, enabling the model to perform tasks with inputs that may not be found in the training vocabulary; as we are constructing code outputs that can vary greatly, this aspect of the T5 transformer is particularly useful. There are several size variants of the T5 architecture, ranging from small (60 million parameters) to larger models with billions of parameters; for our methods, we use the small variant. The default T5 transformer continues to employ cross entropy loss, and the baseline we measure against follows this method, and is covered in more detail in Chapter 4: Methodology.

Converse to the cross entropy loss default, contrastive loss seeks to create a distribution space where similar embeddings have a lower distance between them while ensuring dissimilar embeddings have a greater distance. To enable this dual functionality, datasets used with contrastive learning require augmentation to provide additional inputs that are similar in nature. While image dataset augmentation has numerous methodologies for augmentation, it is a more difficult problem with text datasets, specifically code-centric ones [8]; we cannot simply change the color or horizontally flip our code inputs. Instead, we

perform this augmentation using a rules-based Java-language code augmentation tool [34], which generates semantically identical variations of the code input using a rules-based approach.

While contrastive learning has made an impact in research related to image-based tasks [14], there is area for growth in the domain of NMT and software engineering-related machine learning. Research related to the use of contrastive learning in software engineering tasks has been performed by Ding et al., with emphasis on clone and bug detection; the work primarily focuses on the problem of out-of-distribution elements and how to correct the issue [8]. We also note that the crossover between text and image modalities has shown improvements in the past, such as in the work of He et al. which demonstrated that masked auto-encoders (MAEs) were capable of producing accurate results in image-related tasks after being initially used in text-related ones [12].

An important aspect of a contrastive loss function is the idea of temperature - a lower temperature parameter tends to produce more determinant results, while a higher temperature tends toward the exploratory. Additional research has been performed concerning the use of scheduling changes in temperature throughout training [16], which has shown increased representational learning. We employ scheduled temperature changes using several growth models (linear, Fibonacci, decreasing) in our efforts.

3.2 Transfer Learning

Transfer learning is not a new concept; the first related paper with regard to neural networks was published in 1976 [4]. The essential element is that a network trained on one topic can utilize the learning already performed to enhance its capabilities in learning a separate, but related, topic. These processes are referred to as "pretraining" and "fine-tuning" the model. Multi-task learning (MTL), where a single pretrained model is simultaneously fine-tuned on several downstream tasks, has been shown to be effective [35].

In our work, we use a T5 model that has been pretrained on a large dataset of code

and code-related text, then fine-tuned on task-specific datasets for bug fixing (small and medium variants), mutant generation, and assert generation; this model has been trained using cross-entropy loss. We use transfer learning from the models provided in the baseline [17], with the key differences being (a) the datasets have been augmented to include modified, though similar, elements and (b) contrastive learning is used for the encoder portion of the transformer. As the decoder is concerned with creating the output and not learning the latent representation, it is not altered.

3.3 Bug Fixes, Mutant Generation, and Assert Generation

Early research in this domain by White et al proposed deep learning methodologies for software engineering task applications, and uses recurrent neural networks (RNNs) for training experiments ; the authors describe their work as "one step—the first step—toward deep learning software repositories" [33]. As an anchor point for this effort, our research seeks to extend the work by Mastropaolo et al. [17], which continued in the tradition of White et al. along with that of Tufano et al. [28], with the addition of the transformer architecture. This study demonstrates an attention-based method for repairing code by learning the patterns associated with bug-fix pairs in a large corpus of software changes. The authors were able to attain between 9% and 50% accuracy in selecting the correct fix for a given bug. Additional work by many of the same authors [27] further clarifies the design decisions of their research. Further research by Chen et al. [5] similarly tackled the task of bug fix selection, though specifically on one-line changes, unlike [28] and [27] which performed at the function level. Additional work by White et al. confirmed the use of deep learning techniques for software engineering tasks, in this case clone detection, achieving 93% success rates across multiple types of clones at the file and function levels [32]. While we do not consider clone detection in our task list, similar data preparation methods in the form of abstracted syntax, are employed. Further, the details of learning from fragments of code in a deep learning environment is essential to our research.

Earlier work by Hajipour et al. [11] used conditional variational auto-encoders (CVAE) to learn distributions of potential fixes, with a focus on learning diverse solutions to the given bugs. Focusing on fixing small and medium C-based bugs, it focused on a smaller dataset than employed in our research, but achieved a high degree of accuracy. This, in turn, was based on the research of Gupta et al. [10], which similarly provided a method for fixing bugs in C code; Hajipour et al. highlight that one problem with similar automated software repair methods resides in the discrepancy between the generation of a fix by the model and the intent of a developer.

Work by Mousavi et al. [20] logically delineates automatic software repair into two categories: runtime (preventing/rescuing from fault at execution) and source code (repairing prior to runtime); our research is focused on the latter. Their research focused on identifying the various hurdles that automated attempts at software repair face, among them the difficulty in finding a suitable fix when the complexity of the bug becomes too high, referencing the work of Motwani et al. [19], who state that under specific syntactic scenarios such as adding additional loops, these types of methods tend to fail. Motwani et al. further point out that bugs which require changes to method signatures present an additional level of complexity that is troublesome for automated methods.

Mutant generation has been approached in baseline research [17]. The use of code mutants was proposed by DeMillo et al. [6] as a method of testing test cases by inserting small defects into code, which at the time of writing was based on FORTRAN code segments. Later work by Aşık et al. [2] produced a method of generating python-based mutants, using transformer architecture and abstracted code. The paper considers those results that are perfect predictions and similar to perfect predictions, and relays a 6% to 35% accuracy rate, with higher accuracy when counting "other buggy codes" produced, with a high degree of lexical accuracy in produced code.

Research by Shamshiri et al. [24] suggests the use of a search-based approach to produce adequate software tests that would identify changes in the code base. Watson et al. introduced AuTomatic Learning of Assert Statements (ATLAS), a method for generating

assert statements for a given code input using an NMT approach. This work proceeded from previous efforts [9] [21] that did not employ deep learning based approaches. The authors are able to achieve 31% generation of the same assert statement as one created by a live developer in single beam generation, with results of 50% accuracy when expanding to a top-5 evaluation [31]. This was improved in the work of Mastropaolo et al. [17] to 56% for abstracted code and 68% with raw code inputs.

Chapter 4

Methodology

The baseline we adopt from Mastropaolo et al. [17] introduces a method for tackling multiple software engineering tasks using a transformer-based approach. The T5 model is pretrained on a large corpus of code and code-related data (such as summaries), which is in turn fine-tuned on task specific datasets. We adopt their datasets (derived from [28], [27], [31]), with augmentation as outlined in Section 4.1, and begin with the code provided by their research. While Mastropaolo et al. experiment with both pretrained and unpretrained versions of T5, we instead continuing fine-tuning their existing models with our additional contrastive loss approach.

Figure 4.1 provides a visual representation of our process, and is detailed below.

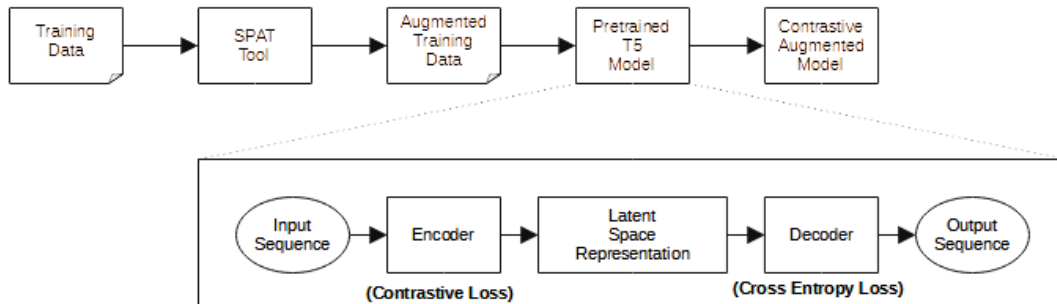


Figure 4.1: Process diagram for including contrastive loss as an additional learning pass.

Testing of contrastive loss in this area begins by utilizing the model code from the

baseline [17], with our implementation of contrastive loss for Mesh Tensorflow [25] to facilitate Tensor Processing Unit (TPU) training. Further, we replace the baseline datasets with our augmented datasets (see Section 4.1). We conduct a series of trials to determine potential starting points for temperature as a hyperparameter.

Our method for scheduled training continues the transfer learning approach on the baseline by performing a second round of fine tuning on the existing model that uses a scheduled contrastive loss objective function to attempt to increase the performance of the T5 model in the selected tasks. We focus on four tasks from the baseline [28], namely small bug fixes (BFS), medium bug fixes (BFM), mutant generation (MG), and assert generation (AG). BFS and BFM are treated as distinct tasks. We do not consider the remaining task present in the baseline, Code Summarization, as it is not a code-to-code task, producing a plaintext description from the supplied code snippet; we are primarily concerned with code-to-code generation.

We implement a custom training loop to perform continued transfer learning using temperature-based contrastive learning with the stated schedulers (linear, Fibonacci, and decreasing). The selection of an appropriate temperature for this work is regarded as an additional hyperparameter; we will vary the temperature ranges and schedules to tune it.

The schedulers, at each iteration, increment (or decrement) the temperature value of the contrastive loss function by a rate that is determined by dividing the range of temperature change steps over the course of the planned training period. For linear schedules, this is simply the difference in the high- and low-temperature values. For the Fibonacci-based schedulers, it is the cardinality of the indices of Fibonacci values to be used.

Our primary concern is increasing the accuracy at various beam sizes, aligned with the baseline. Further, we will analyze the differences in the accurately predicted results with those of the baseline to determine orthogonality. We will also observe the differences in training duration; we begin this effort with the expectation that contrastive scheduled learning will take longer to perform, due to the increased size of the dataset after applying the data augmentation.

We intend to answer the following research questions:

- RQ1: What effect does contrastive loss in continued transfer learning have on code repair, mutant generation, and assert generation prediction accuracy?

Our figure of merit will be the accuracy at selected beam sizes (1, 5, 10, 25, 50) for the Bug Fix (Small), Bug Fix (Medium), and Assert Generation tasks, and beam size 1 for the Injection of Code Mutants task, in keeping with the baseline paper.

- RQ2: What are the limitations of temperature-based contrastive loss in these tasks?

Our implementation of scheduled contrastive loss is, to the best of our knowledge, novel for use within a MeshTensorflow-based approach to T5-based software engineering task machine learning research. As such, we seek any issues introduced by our specific implementation, as well as general limitations noted during our efforts.

- RQ3: What overlap do the results have with the baseline method?

Our stated goal is to increase accuracy while maintaining the results captured in the baseline. We consider the proportions of baseline results that are still present in the results from our method compared against the unique results produced by the contrastive learning model.

4.1 Data Augmentation & Processing

Our datasets are derived from those presented in the baseline [28]. We employ separate datasets for BFS, BFM, MG, and AG. In order to augment the dataset with functionally equivalent alternatives, we generate variants using [34] (specifically, a configurable implementation available at [7]) based on the following rules:

- Boolean Exchange
- Conditional Expression to Single If
- Infix Expression Dividing

- Loop Exchange
- Loop If Continue to Else
- Reverse If Else
- Single If to Conditional Expression

While there are additional rules available in the augmentation tool, we limited the list based on two criteria. First, after an initial test run, which rules reliably provided multiple results for our input set. Second, we removed rules that provided only boilerplate insertions, such as inserting a single dummy log statement; the reasoning behind this decision is that we seek changes with functional equivalence and substance in the code difference. As every input sequence would, keeping with the example, be capable of receiving an extra meaningless line, it would not generate a semantically similar and substantially, textually different variation.

Only the training data is augmented; the test sets are left in their original state to make comparison with the baseline clear. Once the data has been augmented, it is passed through the src2abs tool [26] to create the necessary abstractions. The exception is the Assert Generation dataset, which is left in raw form; we note that in the baseline [17], the raw formatted AG task exceeds the accuracy of the abstracted version. Finally, we create a "stacked" dataset to ensure that the variations are distributed well throughout, with the goal of not having too many variations of the same original input present in any given batch. For each task, ten variations of the input dataset are generated and then concatenated. Appendix B provides a selection of samples of augmented code elements and additional details.

Table 4.1 provides a summary of the increases in dataset size produced by this augmentation. We note that BFM received the largest increase in size, which follows from the nature of the datasets. BFS and MG both tend to have smaller inputs and targets, while BFM has more room for elements that the tool can consider for augmentation.

Dataset	Initial Count	Augmented Count	% Increase
BFS	46680	53154	13.87%
BFM	52364	119234	127.70%
MG	92477	110316	19.29%
AG	126,477	184791	46.05%

Table 4.1: Summary of dataset size increases using the semantic-preserving augmentation method.

Original	Variant
<pre>public void METHOD_1 (android.view.View view) { if (!(context) instanceof TYPE_1) { } else { ((TYPE_1) (context)).METHOD_2(string); } }</pre>	<pre>public void METHOD_1 (android.view.View view) { if ((context) instanceof TYPE_1) { ((TYPE_1) (context)).METHOD_2(string); } else { } }</pre>

Table 4.2: A sample from the BFS dataset, processed through the augmentation and abstraction processes.

Table 4.2 provides a sample of an augmentation, post-abstraction - in this case, the swapping of branches in an if...else statement via the Reverse If Else rule. The condition at the beginning of the statement is reversed, and the branches are exchanged. This provides the semantically similar, textually different desired variation.

Chapter 5

Results

Of the three contrastive temperature schedulers, the Fibonacci-based scheduler outperformed the linear and linear-decreasing schedulers in most cases, though relatively close. For example, the linear and linear-decreasing methods for BFS provided accuracy of 20.5%, with the Fibonacci scheduler achieving 21.2%. We attempted Fibonacci-decreasing method as well, resulting in 20.8%. As such, we continued with the Fibonacci scheduler for our further attempts, increasing the temperature from the first through tenth Fibonacci numbers over the course of training.

Accuracy generally declined after 20 epochs. As each dataset is composed of ten variations, including some duplicates in cases where ten variations were not generated by [7], is approximately the length of ten of the baseline datasets, this can be considered equivalent to 200 epochs. This is done to increase the instances in which differentiation between similar and dissimilar samples are encountered, giving the contrastive loss function opportunities to increase the distance in the representation.

5.1 Research Question 1

What effect does contrastive loss in continued transfer learning have on code repair, mutant generation, and assert generation prediction accuracy?

The results of these experiments can be seen in Tables 5.1 & 5.2. For the single-task

models (Table 5.1), our methods provide a marked increase in the accuracy of BFS and BFM tasks (40.7% and 8.9%, respectively), while providing a very minor increase in MG accuracy (0.9%) on Beam 1. The AG task fails to exceed the baseline figures at any tested beam sizes. In later beam sizes, only the BFM task improved on the baseline at Beam 5, showing a 1.1% increase. All other trials for single task models fail to pass the baseline, with the divide growing as the beam size increases.

For the multi-task model, markedly higher improvements in accuracy are seen on Beam 1. BFS and BFM increase by 96.5% and 262.5%, respectively, with MG continuing to have a lesser increase at 2.7%. For Beam 5, BFS and BFM show 0.3% and 5.2% increases. As with the single-task models, later beam sizes do not see an improvement over the baseline for BFS and BFM, and MG is restricted to the first beam.

The results for the AG testing in the multitask model exceed the baseline at all tested beam sizes except Beam 50. The largest gains occur in Beam 1, as with most other trials, with a 32.1% increase. As the beam size increases, the difference between the experimental and baseline accuracies begins to converge, with the Beam 50 accuracy falling to 0.2% below the baseline.

With these results, we can state that contrastive loss in continued transfer learning leads to higher accuracy for these tasks at early beam sizes, with diminishing returns for additional beam sizes until a breakpoint at which cross entropy loss alone becomes more accurate.

5.2 Research Question 2

What are the limitations of temperature-based contrastive loss in these tasks?

Due to the nature of the temperature scheduler, which changes over the course of the planned training length and not on a per-epoch basis, we conduct initial model runs to determine the appropriate number of epochs to perform. This is done empirically, with 20 epochs of the stacked augment datasets providing the highest accuracy in each case. This

adds some time considerations to training, as training at e.g. 10, 20 and 30 epochs requires three separate training runs; early stopping is not an option with this implementation.

As noted in Section 5.1, the accuracy of our method decreases versus the baseline as the beam sizes increase, until the improvement vanishing entirely. Thus we consider the beam size as a limitation of this approach; for tasks which rely on higher beam sizes, there comes a point when cross entropy loss alone dominates our method.

We also find examples where the generated code output is feasibly an acceptable fix, mutant, or assert for the input, but does not match the ground truth of the dataset. Appendix A provides samples of failed predictions for the experimental results, to include those suitable but imperfect predictions, and an analysis of the same.

5.3 Research Question 3

What overlap do the results have with the baseline method?

For this analysis, we consider whether results in one result set are present in the other; for example, in the BFS task, what portion of the baseline results does the experimental contrastive model capture? This is done by comparing the inclusion of indices from the test set in each result set. We consider these counts as a proportion of the baseline, which we call the *unique multiple*. A unique multiple value less than one indicates that the given task did not exceed the baseline, such as with the Single Task AG results. These results can be found in Table 5.3. As the mutant generation task is restricted to beam size 1, it is excluded from this section of analysis. A higher unique multiple can indicate the experimental model produced more results than the baseline, though it does not indicate that baseline results were captured. Thus we also calculate what percentage of the total results in the experiment are contained in the joint set.

For the single task BFS trial, we see that the largest proportion of results are contained in the joint set (55.493%), which combined with the unique multiple (2.855) indicates an ideal scenario for our method - the accuracy increased without losing too many of the

baseline results. For the BFM trial, we can see the opposite. The joint percent of shared results is very low (12.899%) with a unique multiple of 1.103 indicates a situation with higher overall accuracy, but many of the results from the baseline are not captured in the new model. Finally, the AG trial shows a scenario where the overall accuracy decreased, albeit slightly, but with an exceptionally high joint percent value. This indicates a similar overall output, with more results lost than gained.

The multitask model, while demonstrating generally more favorable results, provides an interesting contrast to the single task. The BFS trial experiences a lower joint percent and a higher unique multiple (32.231% and 3.622, respectively), showing a high accuracy increase as discussed in Section 5.1. The AG trial, which failed in the single task version, rises slightly over the baseline. Similar to the single task version, it maintains a high joint percent and low overall unique multiple. The BFM trial for this beam size, representing the highest accuracy increase seen in this research, also has the lowest joint percent (3.926%) and the highest unique multiple (4.059). A near-negligible amount of the results from the baseline are kept in the new method.

Task		Beam				
		1	5	10	25	50
Bug Fix (Small)	Baseline	15.08	32.08	37.01	42.51	45.94
	Experiment	21.22	31.68	35.40	40.23	43.80
Bug Fix (Medium)	Baseline	11.85	19.41	23.28	28.60	32.43
	Experiment	12.91	19.64	22.34	25.57	28.42
Mutant Generation	Baseline	28.72				
	Experiment	28.98				
Assert Generation	Baseline	68.93	75.95	77.70	79.24	80.22
	Experiment	65.37	71.49	73.96	75.90	77.02

Table 5.1: Experimental results for single-task models.

Task		Beam				
		1	5	10	25	50
Bug Fix (Small)	Baseline	11.61	35.64	43.87	52.88	57.70
	Experiment	22.81	35.75	41.17	46.86	50.35
Bug Fix (Medium)	Baseline	3.65	19.17	24.66	30.52	35.56
	Experiment	13.29	20.23	23.52	26.80	30.73
Mutant Generation	Baseline	28.92				
	Experiment	29.72				
Assert Generation	Baseline	58.60	66.90	70.31	73.19	74.58
	Experiment	59.32	68.25	70.92	73.22	74.45

Table 5.2: Experimental results for the multi-task model.

Single Task			
Task	BFS	BFM	AG
Baseline Perfect Prediction	880	776	12970
Experiment Perfect Prediction	1238	845	12300
Baseline Unique Results	193	667	1111
Experiment Unique Results	551	736	441
Joint Results	687	109	11859
Unique Multiple	2.855	1.103	0.397
Joint Percent	55.493	12.899	96.415
Multitask			
Task	BFS	BFM	AG
Baseline Perfect Prediction	678	239	11026
Experiment Perfect Prediction	1331	866	11161
Baseline Unique Results	249	205	1012
Experiment Unique Results	902	832	1147
Joint Results	429	34	10014
Unique Multiple	3.622	4.059	1.133
Joint Percent	32.231	3.926	89.723

Table 5.3: Overview of overlap analysis (beam size 1)

Chapter 6

Future Work

There is still room for future research in this area. First, there is the issue of hyperparameter tuning. At present, we do not have an *a priori* method for determining the appropriate temperature ranges or type of scheduler to be used; all results are currently obtained empirically. A method for determining temperature ranges before training would be desirable. Alternatively, an implementation that learned the appropriate temperature during training may be of value.

Second, our implementation’s temperature scheduler changes the temperature over the entirety of the training period. This makes it difficult to incorporate early stopping while still traversing the entirety of the given temperature range. It would be beneficial to explore methods that perform the scheduled temperature changes on a per-epoch basis.

Finally, our intent with the research was to capture the baseline results while adding to overall accuracy with new results. In this, we are only partially successful: The accuracy increased, but not without loss to the captured baseline perfect predictions. Further research is required to determine what changes to our method, if any, will more successfully maintain the joint percent capture.

Chapter 7

Threats to Validity

We strive to provide a methodology for software engineering machine learning tasks that is generalized and rigorous; however, there are areas of this research that could introduce issue to the results we provide.

- We note that an internal threat to validity exists from the decision to use only Java-based inputs. While this was done to keep with the baseline selected, it limits the nature of our findings to similar inputs.
- With regard to the construction of our method, the nature of using unshuffled, static datasets to ensure the non-overlapping variations from data augmentation introduces the question of whether the ordering of these datasets could produce differences in our outcomes.
- As with other research using similar architectures and methods, we maintain the limitations derived from the problem of outputs limited to a rearrangement of tokens learned during training.
- Due to the hyperparameter tuning issues noted above, we introduce a conclusion validity issue until a method is developed to determine exact temperature ranges and optimal schedules, and the implementation of an effective epoch-based approach.

- Our analysis of the accuracy of this method is based on the production of a perfect prediction; that is, we only count those results where the output is exactly matched to the ground truth. For a given input, it is possible that there are multiple possible real-world bug fixes, multiple versions of a mutant, and various asserts not provided by the live developer. Validating *any* appropriate m_{fix} for a given m_{bug} and vice versa for mutant generation would require considerable manual effort.
- As demonstrated in Appendix A, many of the erroneous predictions of our model are related to a lack of context at the function level; for example, AG predictions that provide a potentially suitable assert statement that replaces "api" with "user" in a context where either could be accurate. It may be of value to produce a dataset that includes not only the function, but some context of the code surrounding a given call to that function.

Chapter 8

Conclusion

Many software engineering tasks, such as bug repair, mutant generation, and the creation of appropriate assert statements, can be time and resource intensive. Due to the patterns in bugs, mutants, and asserts, it is possible to generate fixes and breaks for a given input sequence; the question resolves to finding the best method of doing so given the current state of technology.

In this work, we implement our contrastive loss scheduler for text-to-text software engineering tasks in a transformer architecture. Our results indicate that fine-tuning the existing baseline models with additional iterations using these schedulers produces considerable accuracy improvements on one-shot inferences, with lower returns at later beam sizes. These improvements range from very minor, such as in the case of mutant generation, to considerable, as with fixing medium-sized bugs. While the accuracy generally increases at lower beams, the amount of baseline results captured in the new method varies greatly from task to task.

It would be premature to declare these tasks as a solved matter; even with the increased accuracy found in the presented methodology, the potential complexity of software bugs alone dwarf the capabilities of current models. This is not even to mention software bugs that arise from business logic issues or system-level problems. Mutant and assert generation may carry with them additional complexities that are beyond our modeling efforts. That

being said, any step toward higher accuracy when solving the categories of tasks presented herein may allow for the time of live developers to be dedicated to the more complex tasks.

Appendix A

Prediction Examples

As our results count perfect predictions only in terms of accuracy, it is unwarranted to show examples those elements which were predicted accurately; the input matches the target output. Instead, our interest is in those results that were exceptionally close to the target output, but were off in some small manner that prevented them from being counted. We conduct this analysis by calculating the Levenshtein distance between the input and the prediction. Levenshtein distance calculates the number of edit operations between two given strings of text. We are restricting our analysis in this section to beam size 1.

We note that some predictions in our BFS experiment produced no change in the output; the input and the prediction are identical. For example, in the multi-task BFS trial, 458 of the 5835 test samples (7.84%) predicted an output perfectly matching the input. Additionally, 56 of the failed predictions (0.95%) have a Levenshtein distance of 1 from the target. In several cases, this is caused by the model opting for one abstracted function name over another. For example:

```
Input:
public void METHOD_1() throws TYPE_1
{
    TYPE_2 VAR_1 = new TYPE_2(this.VAR_2, STRING_1);
    VAR_1.METHOD_2();
    VAR_1.METHOD_3();
    TYPE_3.assertNotNull(VAR_1.METHOD_4());
}
```

Target:


```

public void METHOD_1() throws TYPE_1
{
    TYPE_2 VAR_1 = new TYPE_2(this.VAR_2, STRING_1);
    VAR_1.METHOD_2();
    TYPE_3.assertNotNull(VAR_1.METHOD_4());
}

```

Prediction:

```

public void METHOD_1() throws TYPE_1
{
    TYPE_2 VAR_1 = new TYPE_2(this.VAR_2, STRING_1);
    VAR_1.METHOD_3();
    TYPE_3.assertNotNull(VAR_1.METHOD_4());
}

```

The only difference found is the invocation of METHOD_2() vice METHOD_3(). Given the input, we note that the key difference between the input and the prediction is the removal of one of the two function calls; our model correctly chose to remove one of those two calls, but selected the wrong one. We then seek out an example where the Levenshtein distance is still 1, but the distance between the input and the prediction is also 1 and find the following example:

Input:

```

public java.lang.String METHOD_1(TYPE_1 VAR_1, java.lang.String VAR_2, java.lang.String VAR_3, TYPE_2 response)
{
    if(METHOD_2(VAR_1, VAR_2, VAR_3, response))
        return STRING_1;
    return STRING_2;
}

```

Target:

```

public java.lang.String METHOD_1(TYPE_1 VAR_1, java.lang.String VAR_2, java.lang.String VAR_3, TYPE_2 response)
{
    if(METHOD_2(VAR_1, VAR_2, VAR_3, response))
        return STRING_2;
    return STRING_1;
}

```

Prediction:

```

public java.lang.String METHOD_1(TYPE_1 VAR_1, java.lang.String VAR_2, java.lang.String VAR_3, TYPE_2 response)
{
    if(METHOD_2(VAR_1, VAR_2, VAR_3, response))
        return STRING_1;
    return STRING_1;
}

```

The correct fix to the bug, per the dataset, is swapping STRING_1 and STRING_2

in the return statements. In the prediction, the second return statement was correctly altered to be `STRING_1`, but the model failed to predict the change of the first return.

Having examined samples at the short edit distance scale, we continue to those with a higher degree of difference. The following example has an edit distance of 33 between the prediction and the target, and 32 from the input to the prediction:

```
Input:
public java.lang.Integer METHOD_1()
{
    return VAR_1.get(0);
}

Target:
public java.lang.Integer METHOD_1()
{
    if ((VAR_1.size())<1)
        return null;
    else
        return VAR_1.get(0);
}

Prediction:
public java.lang.Integer METHOD_1()
{
    if (!(VAR_1.isEmpty()))
    {
        return VAR_1.get(0);
    }
    return 0;
}
```

This example provides an interesting result. The solution given in the target is that instead of simply returning the `get(0)` value of an object, we instead want to make sure an element 0 exists - if so, return it, otherwise, return a null value. Our prediction suggests returning the value if the object is not empty, otherwise returning 0. While it is dependent on the nature of the object in question, *this could be regarded as a semantically equivalent response*; it closely resembles a **reverse_if_else** rule application within our augmentation tool's methods (see Appendix B for more details). This lends credibility to the idea that the model may be producing adequate fixes for a given bug, but such a fix is not reflected in the accuracy of the model due to not matching the prediction proscribed in the dataset.

When examining the BFM multi-task results, we find a similar example:

```

Input:
private java.lang.String METHOD_1(java.lang.String VAR_1)
{
    if(VAR_1==null)
    {
        throw METHOD_2(STRING_1);
    }
    java.lang.String VAR_2=STRING_2;
    for(inti=0;i<(VAR_1.length());i++)
    {
        VAR_2+=(VAR_1.METHOD_3(i))+1;
    }
    return VAR_2;
}

```

```

Target:
private java.lang.String METHOD_1(java.lang.String VAR_1)
{
    if(VAR_1==null)
    {
        return STRING_2;
    }
    java.lang.String VAR_2=STRING_2;
    for(inti=0;i<(VAR_1.length());i++)
    {
        VAR_2+=(VAR_1.METHOD_3(i))+1;
    }
    return VAR_2;
}

```

```

Prediction:
private java.lang.String METHOD_1(java.lang.String VAR_1)
{
    if(VAR_1==null)
    {
        throw new java.lang.NullPointerException();
    }
    java.lang.String VAR_2=STRING_2;
    for(inti=0;i<(VAR_1.length());i++)
    {
        VAR_2+=(VAR_1.METHOD_3(i))+1;
    }
    return VAR_2;
}

```

We see that the intended fix for the input is to return a preset string (note that `STRING_2` is not a parameter, nor defined in the function). The prediction of our model instead throws a null pointer exception, which has a high possibility of being a valid response for `VAR_1` being null when passed to the function. This sort of bug is highly

dependent on the context of the function's usage; nothing about the function's internal content provides relevant details to further inform the model's decision making. This is in keeping with the model attempting to replicate the actions of a live developer; without context, the developer would experience similar issues in providing a fix, but with the added advantage of being able to observe how the function is used to determine what would correct the error. Thus the model is producing a potentially valid fix, but not one that is provided in the dataset.

Finally, we observe the AG trial predictions. First, we consider a case where the ground truth assert is highly specific:

```
Input:
testSize()
{
    "<AssertPlaceholder>";
}
size()
{
    return elementData.size();
}

Target:
org.junit.Assert.assertEquals(7, tree.size())

Prediction:
org.junit.Assert.assertThat(list.size(), org.hamcrest.Matchers.is(0))
```

There is no argument that the prediction in this case is incorrect. It does not provide the same functionality that the target output does; however, it is unlikely that a model would arrive at the target. There is nothing about the input that would give the idea "the appropriate assert in this case would be to ensure the tree size is seven". In a similar manner, we see the following result:

```
Input:
test()
{
    "<AssertPlaceholder>";
}

Target:
org.junit.Assert.assertTrue(((name.contains(""))==(result)))
```

```
Prediction:
org.junit.Assert.assertTrue(true)
```

The test in this case is completely bereft context, a generic test that simply asks to be given an assert without any awareness. Undoubtedly, or at least well within our base assumptions, the assert statement given in the target is valid when given the full context. However, the prediction of our model is an assert statement that will always pass.

We proceed to the initial scope of the BFS analysis and look for predictions with low Levenshtein distance between the target and prediction. Receding to this view provides the following example:

```
Input:
testLogin()
{
    "<AssertPlaceholder>";
}
isLoggedIn()
{
    returnloggedIn;
}

Target:
org.junit.Assert.assertTrue(api.isLoggedIn())

Prediction:
org.junit.Assert.assertTrue(user.isLoggedIn())
```

This example is not much longer than the previous, but that length is important - the `testLogin()` function has a slightly more detailed name, and a second function in the input provides additional context. The definitive answer is to check the **api** object to determine if the account/user/service is logged in. The model's prediction appears to make an assumption that it is a user that must be logged in, and asserts accordingly. This is a potentially valid answer that context surrounding the usage of the function would potentially direct in a more correct manner.

Appendix B

Data Augmentation Examples

In this section we examine samples of various data augmentation rules that are applied to our dataset. The data augmentation tool [7] can apply a rule multiple times to the same input code segment, assuming there are multiple instances of the input trigger that can be affected by the rule. For example, consider the following sample from the AG task training set (note that unrelated portions of the input have been replaced by a comment line):

```
testOverflow (
{
    # Unrelated Code Portion Replaced Here
    try
    {
        currentThread . setPriority ( Thread . MAX_PRIORITY );
        for ( int i = 0 ; i < numFiles ; ++ i )
        {
            java . lang . String fileName = java . lang . String . format ( %05d , i ) ;
            files . add ( new name . pachler . nio . file . File ( parent , fileName ) ) ;
        }
        for ( name . pachler . nio . file . File f : files )
            # Unrelated Code Portion Replaced Here
        do
        {
            resultKey = ws . poll ( ) ;
            if ( resultKey != null )
            {
                name . pachler . nio . file . List < name . pachler . nio . file . WatchEvent < ? > > list = resultKey . pollEvents ( ) ;
                numEvents += list . size ( ) ;
                if ( ! overflowDetected )
                    overflowDetected = listContainsEvent ( list , name . pachler . nio . file . OVERFLOW , null ) ;
            }
        }
    }
}
```

```

        while ( ( ! overflowDetected ) && ( resultKey != null ) ) ;
            <AssertPlaceholder> ;

    }
    finally
    {
        currentThread . setPriority ( initialPriority ) ;
        for ( name . pachler . nio . file . File f : files )
            f . delete ( ) ;

    }
}
reset ( )
{
    if ( ! ( isValid ( ) ) ) return false ;
    if ( ! ( signalled ) ) return true ;
    signalled = false ;
    return service . reset ( this ) ;
}

```

This code segment provides several opportunities to create a variant based on the **reverse_if_else** rule, which will either negate a single if statement and insert an else branch, or swap the contents of an if-else structure’s branches. For example, the `reset()` function at the end of the input contains two opportunities for this rule to be applied as:

```

Variant 1:
    if (!(isValid())) ; else return false;
Variant 2:
    if (!(signalled)) ; else return true;

```

Functionally, these two variants are equivalent to the original code; no additional functionality is introduced or nothing is taken away. In this example, the code is more cumbersome and potentially less readable, but it meets our need of a semantically identical code segment to provide augmentation to our dataset. In total, the augmentation tool provided three variants of this segment, each under the **reverse_if_else** rule.

Other code segments provide opportunities for different types of rule implementations. For example, this code segment allows for instances of **reverse_if_else**, **infix_expression_dividing**, and **single_if_to_conditional_expression**:

```

testHLLMergeDisjoint ( )
{
    # Unrelated Code Portion Replaced Here

```

```

    <AssertPlaceholder> ;
}
count ( )
{
    if ( ( invalidateCount ) || ( ( cachedCount ) < 0 ) )
    {
        if ( encoding . equals ( com . github . prasanthj . hll . HyperLogLog . EncodingType . SPARSE ) )
        {
            # Unrelated Code Portion Replaced Here
        } else {
            # Unrelated Code Portion Replaced Here
            if ( noBias )
            {
                # Unrelated Code Portion Replaced Here
                if ( numZeros != 0 )
                {
                    h = linearCount ( m , numZeros ) ;
                }
                if ( h < ( getThreshold ( ) ) )
                {
                    cachedCount = h ;
                }
            } else {
                double yD4Zgcep = 0.033333 * pow;
                if ( ( cachedCount ) <= ( 2.5 * ( m ) ) )
                {
                    if ( numZeros != 0 )
                    {
                        cachedCount = linearCount ( m , numZeros ) ;
                    }
                } else if ( ( ( chosenHashBits ) < 64 ) && ( ( cachedCount ) > ( yD4Zgcep ) ) )
                {
                    if ( ( cachedCount ) > ( ( 1 / 30 ) * pow ) )
                    {
                        # Unrelated Code Portion Replaced Here
                    }
                }
            }
        }
        invalidateCount = false ;
    }
    return cachedCount ;
}

```

Each of the if statements provides an opportunity for reversal, and the if statements without else branches, that contain only an assignment operation, can be converted to conditional expressions. The infix expression division provides a more detailed change, extracting portions of a calculation and creating a variable to store the extraction. In an example from the above input:


```

Original:
if ( ( cachedCount ) > ( ( 1 / 30 ) * pow ) )
{
    cachedCount = ( ( long ) ( ( - pow ) * ( java . lang . Math
        . log ( ( 1.0 - ( ( ( double ) ( cachedCount ) ) / ( ( double ) ( pow ) ) ) ) ) ) ) ) ) )
}
Variant:
long OHV7k5Kc = ( 1 / 30 ) * pow;
if ( ( cachedCount ) > ( OHV7k5Kc ) )
{
    double CmUjovHF = ((double) (cachedCount)) / ((double) (pow));
    cachedCount = ( ( long ) ( ( - pow ) * ( java . lang . Math . log ( ( 1.0 - ( CmUjovHF ) ) ) ) ) ) ) ) )
}

```

Overall, the augmentation tool provided 46 variants of this input among the three listed categories.

Finally, there are some code segment inputs that do not provide any opportunity for augmentation for the selected rules. Typically, these are the instances of simpler inputs that do not contain enough grammatical elements to find an appropriate variant. For example:

```

testGetCount ( )
{
    "<AssertPlaceholder>" ;
}
getCount ( )
{
    return asInt ( "count" ) ;
}

```

Overall, for code segments where a rule matches the input, approximately 3 variants are created on average. There are some hefty outliers; for example, one input for the AG dataset generated 75 variations. Further, the AG dataset produced variants for only 6.6% of inputs, due to the presence of simpler inputs.

Bibliography

- [1] ANDREA ARCURI. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011.
- [2] SERGEN AŞIK AND UGUR YAYAN. Generating python mutants from bug fixes using neural machine translation. *IEEE Access*, PP:1–1, 01 2023.
- [3] DZMITRY BAHDANAU, KYUNGHYUN CHO, AND YOSHUA BENGIO. Neural machine translation by jointly learning to align and translate, 2016.
- [4] STEVO BOZINOVSKI AND ANTE FULGOSI. The influence of pattern similarity and transfer learning upon the training of a base perceptron b2, 1976.
- [5] ZIMIN CHEN, STEVE KOMMRUSCH, MICHELE TUFANO, LOUIS-NOEL POUCHET, DENYS POSHYVANYK, AND MARTIN MONPERRUS. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, September 2019.
- [6] R.A. DEMILLO, R.J. LIPTON, AND F.G. SAYWARD. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
- [7] DEVY99. Data-augmenter. <https://github.com/Devy99/data-augmenter>, 2023.
- [8] YANGRUIBO DING, SAIKAT CHAKRABORTY, LUCA BURATTI, SAURABH PUJAR, ALESSANDRO MORARI, GAIL KAISER, AND BAISHAKHI RAY. Concord: Clone-aware contrastive learning for source code, 2023.
- [9] GORDON FRASER AND ANDREA ARCURI. Evosuite: automatic test suite generation for object-oriented software. ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.
- [10] RAHUL GUPTA, SOHAM PAL, ADITYA KANADE, AND SHIRISH SHEVADE. Deep-fix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017.
- [11] HOSSEIN HAJIPOUR, APRATIM BHATTACHARYYA, CRISTIAN-ALEXANDRU STAICU, AND MARIO FRITZ. Samplefix: Learning to generate functionally diverse fixes, 2021.
- [12] KAIMING HE, XINLEI CHEN, SAINING XIE, YANGHAO LI, PIOTR DOLLÁR, AND ROSS GIRSHICK. Masked autoencoders are scalable vision learners, 2021.

- [13] SEPP HOCHREITER AND JÜRGEN SCHMIDHUBER. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.
- [14] PRANNAY KHOSLA, PIOTR TETERWAK, CHEN WANG, AARON SARNA, YONGLONG TIAN, PHILLIP ISOLA, AARON MASCHINOT, CE LIU, AND DILIP KRISHNAN. Supervised contrastive learning. *CoRR*, abs/2004.11362, 2020.
- [15] HERB KRASNER. The cost of poor quality software in the us: A 2018 report. *Consortium for IT Software Quality*, 9 2018.
- [16] ANNA KUKLEVA, MORITZ BÖHLE, BERNT SCHIELE, HILDE KUEHNE, AND CHRISTIAN RUPPRECHT. Temperature schedules for self-supervised contrastive methods on long-tail data, 2023.
- [17] ANTONIO MASTROPAOLO, NATHAN COOPER, DAVID NADER PALACIO, SIMONE SCALABRINO, DENYS POSHYVANYK, ROCCO OLIVETO, AND GABRIELE BAVOTA. Using transfer learning for code-related tasks. 2022.
- [18] ROBERTO MINELLI, ANDREA MOCCI AND, AND MICHELE LANZA. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC '15*, page 25–35. IEEE Press, 2015.
- [19] MANISH MOTWANI, SANDHYA SANKARANARAYANAN, RENÉ JUST, AND YURIY BRUN. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering (EMSE)*, 23(5):2901–2947, October 2018. <https://doi.org/10.1007/s10664-017-9550-0>. DOI: 10.1007/s10664-017-9550-0.
- [20] S. AMIRHOSSEIN MOUSAVI, DONYA AZIZI BABANI, AND FRANCESCO FLAMMINI. Obstacles in fully automatic program repair: A survey, 2020.
- [21] CARLOS PACHECO AND MICHAEL D. ERNST. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA '07*, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery.
- [22] MARTIN POPEL, MARKÉTA TOMKOVÁ, JAKUB TOMEK, ŁUKASZ KAISER, JAKOB USZKOREIT, ONDŘEJ BOJAR, AND ZDENEK ZABOKRTSKY. Transforming machine translation: a deep learning system reaches news translation quality comparable to human professionals. *Nature Communications*, 11:4381, 09 2020.
- [23] COLIN RAFFEL, NOAM SHAZEER, ADAM ROBERTS, KATHERINE LEE, SHARAN NARANG, MICHAEL MATENA, YANQI ZHOU, WEI LI, AND PETER J. LIU. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.
- [24] SINA SHAMSHIRI. Automated unit test generation for evolving software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, page 1038–1041, New York, NY, USA, 2015. Association for Computing Machinery.

- [25] NOAM SHAZEER, YOULONG CHENG, NIKI PARMAR, DUSTIN TRAN, ASHISH VASWANI, PENPORN KOANANTAKOOL, PETER HAWKINS, HYOUKJOONG LEE, MINGSHENG HONG, CLIFF YOUNG, RYAN SEPASSI, AND BLAKE HECHTMAN. Mesh-tensorflow: deep learning for supercomputers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 10435–10444, Red Hook, NY, USA, 2018. Curran Associates Inc.
- [26] MICHELE TUFANO. src2abs. <https://github.com/micheletufano/src2abs>, 2019.
- [27] MICHELE TUFANO, JEVGENIJA PANTIUCHINA, CODY WATSON, GABRIELE BAVOTA, AND DENYS POSHYVANYK. On learning meaningful code changes via neural machine translation. *41st ACM/IEEE International Conference on Software Engineering*, May 2019.
- [28] MICHELE TUFANO, CODY WATSON, GABRIELE BAVOTA, MASSIMILIANO DI PENA, MARTIN WHITE, AND DENYS POSHYVANYK. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, September 2019.
- [29] ASHISH VASWANI, NOAM SHAZEER, NIKI PARMAR, JAKOB USZKOREIT, LLION JONES, AIDAN N. GOMEZ, ŁUKASZ KAISER, AND ILLIA POLOSUKHIN. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.
- [30] CODY WATSON, NATHAN COOPER, DAVID NADER PALACIO, KEVIN MORAN, AND DENYS POSHYVANYK. A systematic literature review on the use of deep learning in software engineering research, 2021.
- [31] CODY WATSON, MICHELE TUFANO, KEVIN MORAN, GABRIELE BAVOTA, AND DENYS POSHYVANYK. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20. ACM, June 2020.
- [32] MARTIN WHITE, MICHELE TUFANO, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.
- [33] MARTIN WHITE, CHRISTOPHER VENDOME, MARIO LINARES-VASQUEZ, AND DENYS POSHYVANYK. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345, 2015.
- [34] SHIWEN YU, TING WANG, AND JI WANG. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022.
- [35] YU ZHANG AND QIANG YANG. An overview of multi-task learning. *National Science Review*, 5(1):30–43, 09 2017.