Justin Scyphers

Bug Fixing Using Pre-Trained-then-Fine-Tuned T5 Model on Java Code

This project had a simple ambition, to expand upon the learnings and scope of project 2 to attempt to fix all bugs in a given function/method via a transformer model.

Assignment 2 had us use Python code and mask 15% of the data and split it into 4 categories:

- Pre-training data, data to create a pre-trained model, the base level that is capable of performing the task, but perhaps not very well due to various reasons.
- Fine-tuning data, data to refine the model and further specialize it to perform a specific task, in our case to fix bugs.
- Testing data, data set aside for the purpose of being fed to the model and have it attempt to fix the code. This data is for, as the name implies, testing if the model functions as desired.
- Evaluation data, data set aside for evaluating the model after testing to judge it's ability in a more final sense. If training data is like studying for a test, this was the test itself.

The same applies for this project, however, we actually have a set of data provided for us. The "medium dataset" from TOSEM 2019 - ACM Transactions on Software Engineering and Methodology by Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk.

This project is a much larger one than what set out to accomplish. Through the application of Neural Machine Translation (NMT) techniques to automatically generate bug-fixing patches for software bugs. By analyzing millions of confirmed bug fixes from GitHub repositories, the researchers created method-level pairs of buggy and corrected code, referred to as bug-fix pairs. These pairs were abstracted to focus on essential code structures, preserving frequently used identifiers and literals, termed idioms. An encoder-decoder model was then trained on these abstracted pairs to translate faulty code into its corrected version. The study demonstrated that this model could accurately predict developer-generated fixes in 9% to 50% of cases, depending on the number of candidate patches considered. Additionally, the model effectively captured various abstract syntax tree operations and could generate potential patches rapidly

Our structure was much simpler, utilizing the medium dataset from their data page on their website, as well as our own collection of data from various github respositories, we aim o pre-train and fine-tune a T5 transformer model after preparing the data in various ways.

Before we can train a model, however, we must first collect and process the data. For the medium dataset, this is simple. The data is already neatly organized into a training, testing, and evaluating folder with two text files containing the buggy and fixed versions of the collected code already tokenized. Including both will allow us to both train on a fixed and buggy dataset, but also test and evaluate with both versions to gain a better understanding of the models capability and reliablity. The only modification needed is to split the training data into pre-train and fine-tune sets to accommodate our strategy.

This dataset is comprised of 65,455 java functions that are between 50-100 tokens each. This raises an immediate concern on the models trained with a dataset as relatively small as this. With 65 thousand smaller functions, concerns of a models ability to handle longer and/or more complex functions arise. A model trained on this dataset theoretically wouldn't be robust, less effective at handling a real world use like a plugin for a developers IDE. What that application isn't our goal, the concerns of the dataset remain.

To address this issue, we will create a second dataset where we grab a larger number of Java functions from numerous github repositories from many different authors. This will increase the diversity and complexity of the code the model will encounter. However, this dataset requires a fair bit more work to get ready to feed into the model. To start, it isn't already collated and tokenized, so first we will need to pull all relevant functions from every file, strip all comments and special characters, and then tokenize the data. Once this is finished, we then need a buggy version of our data to mirror the medium dataset. There are various methods to do this such as tools like PITest or Major or code your own solution. For this project, we will attempt to use a few of these tools and ensure the changes made are sufficient.

To delve further into the dataset issues, the problem of the medium dataset being 50-100 characters can be offset by how many changes and the complexity of the changes are made for the model to learn from and fix. If you significantly modify a function to include numerous types of errors in inconsistent places, the complexity of the function is much higher than a large function with 1 type of issue in the same place. The same can be said the length of a function, however, more code is more real estate to inject and modify bugs. The bigger issue is the diversity of the code, the same structure across the majority of the methods is a bigger concern for how the model will learn what a function is and how to properly fix it.

Once the 4 different datasets are made and properly split into the 4 categories mentioned at the start, we can begin working on a pretrained model. To do this, we will download the hugging face T5 model and construct python code to use it. This was largely done before, so the process is roughly the same. Utilizing the hugging face forums and documentation I set out to modify the code from assignment 2 to use T5-base instead of my previous distilbert.

The T5 model was selected for this project by the professor, however, the T5 model is a solid choice

With this inclusion, I attempted to modify the code to use it making minimal changes as the code was proven to work previously, however, I ran into issues immediately in dependencies and libraries missing. Various pytorch, python version, and general structure changes in the difference between these models became cumbersome enough that I started over from scratch and built a new script to generate the pre-trained model. This took several days and a lot of trial and error, but the steps were largely the same and well documented online making the script simple enough. However there was a problem in its execution, the same problem I ran into with project 2.

In project 2, I initially set out to use the CodeBERT model because it excels at analyzing code, having been pre-trained on a large corpus of programming languages including Python. Its focus on understanding and generating code made it a natural choice for my Python-based project, as it excels in tasks like code search, defect detection, and other classification or token-level tasks. Additionally, CodeBERT's ability to capture the syntax and semantics of Python code provided a solid foundation for training on bug-related scenarios, making it an effective tool for analyzing and improving Python codebases. Another, more simple reason, is that the setup and execution of the model was well documented for a beginner like myself. I had never used a transformer model or performed the tasks involved so a detailed guide and explanation was very helpful for this project.

However, the process of training the model took many hours to several days, often erroring during the process, leading to, at times, multiple days being lost while waiting for it to fail. This forced me to restart the lengthy process several times, which became a significant challenge as the project deadline approached. To address this, I switched from codebert to distilbert, a smaller and faster variant designed to retain much of the functionality of its larger counterparts while being more computationally efficient. This transition allowed me to continue developing the Python-focused bug-fixing model within the time constraints, albeit with some trade-offs in performance and capabilities. Despite the setbacks, the experience provided valuable insights into managing resources, troubleshooting training issues, and adapting models to meet project requirements.

While working on the T5/java project, I initially began training using the T5-base model due to its balance between size and capability, believing this version to be a compromise between a higher end model and my hardwares struggles in the past. However, much like my experience with CodeBERT, the training process for T5-base proved to be incredibly time consuming and computationally demanding. The model frequently ran into resource limitations, especially during the pre-training phase on both datasets, causing interruptions and necessitating restarts. With deadlines looming and training progress stalling, I had no choice but to switch to the smaller and more lightweight T5-small model. This decision was made out of necessity to ensure the project could be completed within the time and hardware constraints available.

The switch to T5-small brought a mix of benefits and drawbacks. On the positive side, T5-small significantly reduced training times and memory usage, allowing for smoother experimentation and quicker iterations. This efficiency made it possible to complete the fine-tuning process without the tremendous time and resource dedication of T5-base. However, the trade-off was a potential decrease in the model's ability to handle complex relationships in the code, as its smaller size limited its capacity to capture nuanced patterns and dependencies. While this reduced performance was a drawback in theory, this project likely wouldn’t encounter that as, even with my larger collected dataset, these datasets and use case was still small enough the impact would be manageable. This would serve as a good proof of concept to then later be leveraged with something like T5-large or higher.

When the pre-training ended, we had two models to begin fine-tuning. Seeing as the old code for pre-training didn’t work, I created a new fine-tune script to process the fine-tune, testing, and evaluation datasets and fine-tune the model. This proved to be relatively smoother, but issues of tuning the script to cover dependencies and bugs took a few more days.

In the process of creating the fine-tuning script, I baked the testing and evaluation in upon the models completion to test the model once done. This hasn’t worked to date, however, as I have still not gotten this to work. I tried across many days to get this functionality working, however, the code involved in testing is more complicated for me, causing many iterations of this code to have been made and subsequently written over. I have no excuse for this, I should have been able to get this working by now, but with various other time sinks and the debugging taking a long time, this project unfortunately stands incomplete. Regardless of my grade on this project, however, I will continue this as I genuinely find it interesting to do.

The plan to test and evaluate was to take the two fine-tuned models for both the medium dataset and my collected dataset, four in total, then test and evaluate them using their own and each others datasets. This would look something like this:

Trained Model	Medium **Test Data**	Medium **Eval Data**	Collected **Test Data**	Collected **Eval Data**
Buggy Medium Set				
Fixed Medium Set				
Buggy Collected Set				
Fixed Collected Set				

The main goal of this was to test the medium dataset trained models on the collected tests and evaluations to see how well it performed with code potentially more complex and diverse then it was trained on. By capturing data like the inference time, prediction accuracy, and confidence scores I would have an idea of which model performs better, but then collect the data to be made into summary scores like bleu and rouge to get more comprehensive, final measurements of the two approaches.

While this project ultimately remains incomplete, the journey has provided invaluable insights into the complexities of training machine learning models for software engineering tasks. The challenges encountered—ranging from hardware limitations to the nuances of model dependency management—highlight the importance of meticulous planning and adaptability in research.

Despite not achieving the primary goal, the project demonstrated a foundational proof of concept for using T5 transformer models in bug-fixing tasks. The iterative learning process, particularly in dataset preparation and script debugging, has equipped me with skills that I plan to refine and apply in future endeavors.

Looking ahead, I intend to revisit this project with better-resourced environments and more robust datasets. By overcoming the limitations encountered during this iteration, there is potential for the model to become a practical tool for automated bug fixing in real-world scenarios. This experience underscores that failure is often a stepping stone to innovation, and I am excited to continue this work beyond the scope of this assignment.