Source Code Generation in Extreme Programming

Jonathan Mahoney
jpmahoney@vt.edu
Virginia Tech
Falls Church, USA

Harish Ravi
harishr@vt.edu
Virginia Tech
Falls Church, USA

ABSTRACT

Software development has many life cycles and development processes that are used to build reliable solutions. One of the common paradigms used today is Extreme programming. This methodology makes use of pair programming, iterative development, and test-driven development. The focus of this study is to reduce the amount of time, resources, and effort put forth practicing TDD using deep learning applications to generate code based on developed unit tests. We fine-tune pre-trained Natural Language processors built for code translation using a corpus of functional Java code, and JUnit unit tests generated with the EvoSuite. We expect that model will generate a well-structured template of code, that may supplement a pair programmer in Test Driven Development(TDD) Environment. Also, this lays the foundation for the model to be used similar to linter plugins in an IDE. In this particular project, we will be building two transformer-based model for generating the source code from the unit tests. We look to evaluate the generated source code on their interestingness and are looking into quantifying and qualitative evaluation of the generated code sequences.

CCS CONCEPTS

• **Software and its engineering → Pair programming**; • **Computing methodologies → Machine translation**.

KEYWORDS

Test Driven Development, Extreme programming, seq2seq, CodetoCode, Transformers

ACM Reference Format:
Jonathan Mahoney and Harish Ravi. 2021. Source Code Generation in Extreme Programming. In *Proceedings of Project Proposal (Intro to DL).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, the quality of enterprise software is one of the primary concerns in software development. Writing understandable, maintainable, and well-tested code is one of the key requirements in modern enterprise software development. The need for writing maintainable code has led to the evolution of various software delivery paradigms where Code Quality Assurance is at the heart of the development process. One such paradigm that has been gaining popularity over the days is Extreme programming, which has seen rapid adoption in various corporate software development processes [9]. With the rising popularity of this paradigm, we would like to explore a potential application of Code-Code software generation models in a Ping-Pong pairing TDD environment, which is one of the important aspects of software delivery using Extreme Programming. TDD involves writing the unit test before the source code is written. This not only ensures that no untested code is ever written, but can make the code more refactor-able in turn making it easy to maintain. [6]

Extreme Programming prescribes that Test Driven Development works well when pair programming is done. In a ping-pong pairing of TDD, one developer writes the test case, and the other developer helps makes the test case pass by writing the source code. [7]. Unlike, traditional pair programming which is done for many reasons, pairing is done in Extreme programming to ensure code quality, and hence will two programmers of similar competency. This resource allocation scheme may not be feasible in enterprise software development, as resource utilization is theoretically doubled for ensuring Code quality. So instead TDD is done with one developer writing the test cases followed by the code, which may be suboptimal, as the developer might not get any tangible third party feedback which is the purpose of pair-programming and eliminates the feedback loops inherent in pair programming which is not an optimal execution of TDD practices.

In this project, we propose using a deep-learning-based system to play the role of the Source Code programmer, who writes the code that makes the unit tests pass. We believe Code quality, is ensured by maximum expert feedback on code semantics, and hence an expert human developer playing this role, Code quality guarantees similar to Extreme programming can be ensured with an AI pair programmer. With the recent advancements in transformer-based models, it's possible to develop systems that can generate high-quality code for a fixed set of unit tests. This iteration of the project aims to build a baseline model that generates code from a set of unit test cases that would sufficiently spike our interest. The future scope of this project can extend well into intelligent Code Refactoring and iterative feature development. These are areas of direct interest for every technology-driven company as Codebase health can directly impact vital business milestones.

2 RELATED WORK

In this section, we explore the related work and other literature surveys that inspire and support our proposal.

We consider our proposal to be quite novel in that Code Generation tasks explored currently, do not take into consideration Extreme Programming concepts like pair-programming in a Test Driven Development Environment. Most of the research in Code

Intro to DL, April 01, 2022, Washington, D.C.

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Generation is not centered around code refactor-ability, but we believe this is an important aspect that must be at the center of enterprise code generation tasks. However, we also speculate that this is a potential area that would likely benefit from code generation models, and this is a sentiment shared by people in the industry [17]. However, it should be noted that Code Generation itself is a well-researched area encompassing models that perform exceedingly well in the relevant problem space. Hence, we feel the need to explore the work done in this domain at a technical level, without drawing too many parallels to TDD itself.

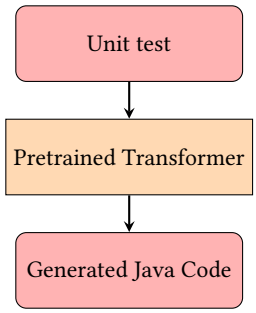
Roziere et al. [21] have explored tasks in the domain in the purview of source code translation. Their work primarily revolves around sanitizing the output of a code generation model. Machine learning models can make mistakes in their predictions, and in NLP tasks we tend to settle for good enough predictions. But, this doesn't work in software translation because these erroneous predictions would mean that we would have errors in our program when they run or compile. To counteract this problem, unit tests are traditionally used, as they provide a level of quality assurance on the code that has been generated. In their paper, they explored methodologies with which source code translation performance can be improved, given that we can generate unit test cases in the target language from the source language. The presence of unit tests allows for the deterministic evaluation of source code. Using a beam search decoder, and the evaluation metrics guided by unit tests, the authors of this paper managed to greatly improve the performance of Code Translators. We believe TDD is the next iteration for these products, as in TDD the evaluation of the generated code can be done with a high degree of determinism, hence setting the stage for transformer-based models to generate production-ready code out of the box. To generate translations from Java to C++/Python, they used a pre-trained Transcoder [15], as the baseline model which was later trained and fine-tuned for this task.

Transformers are a recent standard for Natural Language Processing that prioritizes the attention pieces in a sequence. They are used for seq2seq tasks and have used attention to accomplish the same. Traditionally RNNs and LSTM were the standards for these tasks, but recently they have started to vastly outperform even RNNs that use attention and sequence pointing to the importance of attention. In addition to that, they have been found to effectively learn the universal knowledge in a domain, and hence benefit from transfer learning. [23] Transformer-based models have found widespread adoption even in software engineering tasks. CodeBERT [2] and CodeGPT [16] has been adopted from their NLP counterparts BERT and GPT, to have the programmatic understanding in various programming languages like Java, Go, etc., by pre-training on the corpus of repositories by performing Masked Language Modelling(MLM). The pretraining with Denoising AutoEncoder or MLM can lead to significant learning in the programming domain, given the great performance of these models when fine-tuned by supervised learning techniques on a plethora of Software Engineering tasks.[16]

There were also other transformer models of interest which we explored, but for our project we would theoretically only need these models as a starting point for our tasks, and since we are only working in a Monolingual setting with Java the most popular language, we don't concern ourselves too much with the pretraining specifics. The notable mentions however are CodeT5 [25] and DOBF[5]. CodeT5 is trained in a way that the pretrained encoder and decoder can be utilized unlike other models discussed earlier, meaning a decoder need not be trained from scratch during our supervised fine tuning. DOBF however acheived state of the art results in Code to Code tasks, by utilizing a Deobfuscation Pre-training Objective instead of MLM.

Another model that has been shown to perform really well is CodeBART proposed by Wasi et al.[2], which uses a BART like model for PL tasks. The standard BART uses BERT as an encoder and GPT as the decoder, and has been shown to perform really well for Code-Code tasks.

```
@Test
public void testConcatenate() {
    String result = Concatenate("one", "two");
    assertEquals("onetwo", result);
}
```




```
public String Concatenate(String one, String two){
    return one + two;
}
```

Figure 1: Training Pipeline

3 DATASET

In our task, the machine learning model should generate the Source Code from a set of unit test cases. To train such a model, we need a corpus of programs with the corresponding unit test cases with good test coverage. While it can be a daunting task to either gather or prepare such a dataset manually, we can generate such a dataset with reasonable utility.

For this task, we found the data generation pipelines described by Roziere et al. [21]. In their work, they generated the test cases for a corpus of Java Programs using EvoSuite [10]. EvoSuite is an open-source tool that uses a search-based technique for generating unit test cases given a Java program. It provides high-quality test cases because it generates these test cases using a mutation-based approach with some novel optimizations.

Fraser et al. [11] have created a corpus of auto-generated unit test cases in java from many popular open-source repositories found on SoundForge, named SF110. While their task intends to benchmark automatic unit test case generators in Java with expected baseline performance to improve on, we intend to use their current benchmark dataset generated from EvoSuite as a means to fine-tune

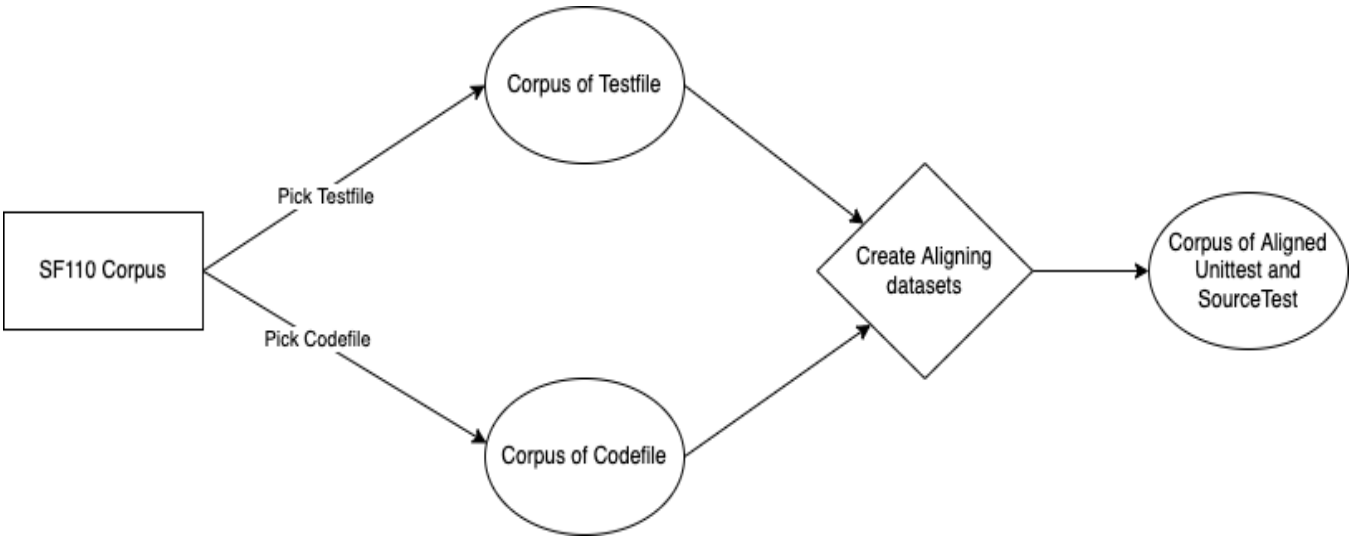


Figure 2: Data Generation pipeline

our PL transformers for a Code-Code generation task, which is to generate the source code from a written unit test.

The data found in the SF110 corpus has the following characteristics which may or may not hold in the real world, or a ping-pong pair programming. A single unit test is present for each code file, and all the possible unit tests are assumed to have been written beforehand so the only task that is left is Source Code Generation. But, in real-world Software development, the generation of Unit test cases occurs iteratively by targetting one function or functionality at a time, and the unit tests written later on will be conditioned on the Code written by the Pair programmer responsible for Coding. Thus, it's not hard to notice a game-theoretic model at play here. Also, the SF110 dataset, doesn't fit into the practical narrative of Test Driven Development(TDD) because in a TDD environment unit test cases must be fixed before the source code is even written, but unfortunately, the Evosuite doesn't operate in the TDD paradigm, and hence the expected utility of SF110 is in the traditional Software Development domain, and not the TDD domain.

While the flaw discussed in the earlier section might be off-putting we can counter that by carefully defining the expected scope of our AI Programmer. Despite, the game-theoretic properties of modeling the Ping-Pong pairing, the duty of an AI programmer is the same as that of a human programmer, which still boils down to generating source code from a unit test case potentially one function at a time. While, the SF110 dataset may not be from the domain of software development we would like, it still has high utility for building pair programming tools in Extreme Programming, as the agile practices are prescriptions at the end of the day, and if we utilize the SF110 dataset, we might potentially get poor performance, even from a skilled human programmer as generating the source code from many unit test cases, can be an extremely daunting task since realistically we will not be working with more than one unit test at a time. However, since the goal is to build an AI pair programmer we will utilize the SF110 dataset while loosening up the evaluation criteria in the training phase, but significantly tone down the prediction task difficulty for the evaluation phase, while simultaneously ramping up the dataset quality, even if it's significantly smaller.

It is also to be acknowledged that despite Evosuite being the de-facto standard for unit-test generation in Java, on closer inspection, it can be discerned that it's nowhere near-human performance in the quality of unit-test cases generated. This can be a distressing aspect of using SF110 corpus because bad quality unit-test cases coupled with a highly complex codebase can significantly impact the learning horizon of the training model. Test-Driven Development relies on extremely high-quality test cases that fix the requirements in stone, and a corresponding generation of small code blocks which might do a complex functionality together with one block, but by breaking down the target problem one at a time. Since the codebases used from SoundForge are probably open-source tools, they are bound to do a ton of complex functionalities. Evosuite despite its huge strengths, cannot reliably produce unit-test cases of the quality demanded by Extreme Programming.

This is a serious flaw and we considered collecting data from Bigquery [13] and generating aligned test-source code from the real-world open-source repositories hosted in Github, which didn't seem realistic in the expected time frame for this project. However we are planning to utilize transformer-based models which can be regarded as an expert system in the domain of interest, thus can transfer their learning when fine-tuned explaining their fast convergence for many tasks when evaluated with the context of expected convergence time. Considering the same, we pose the question can transformers perform Source Code Generation effectively by not relying on the quality of the training dataset, but by transfer learning? Considering this, we were searching for surprises more than an expectation of performance. We would utilize the SF110 corpus with low-quality unit test cases and a source code of high complexity to train the transformers while utilizing a real-world corpus to test our Code-Code model, which has unit test cases with precise specification and a target source code of low complexity.

The data generation pipeline is described in Fig. 8, which involves aligning the Source Code with the Test Code, by identifying each Source Code and the corresponding Test Code File, generated by Evosuite in the respective directory. At the end of this, we make our data palatable for training by pre-processing the same with Python, and basic python scripting. This provided us with the dataset using which we use to fine-tune our transformers. For evaluating the model performance more realistically, we generated the aforementioned realistic Unit-test data by sampling 10 Source Code and Unit test pairs from various TDD centric open source repositories for Java.[22] [12] Using these datasets, we hope to evaluate the utility of the source code generated by our model.

Pretrained Transformer models as discussed earlier can lead to faster convergence in NLP tasks. They leverage the domain knowledge gained from pre-training in a vast corpus of generalized data and can transfer this knowledge into a specialized domain when they are fine-tuned. Many Natural Language models have been developed in the domain of software engineering, and hence we look forward to using them in our project. These include CodeBERT, CodeGPT, PLBART[1], CodeT5, DOBF, Transcoder, CuBert [14] to name a few. They can be either an encoder-only model, where the pre-trained Code Encoder can be used and the Decoder is to be learned from scratch, or they might have a pre-trained Encoder and Decoder like CodeT5. The intricate workings of some of these models have been covered in our literature survey. Almost all of the models listed or explored have been exposed to rounds of self-supervised pretraining with either Denoising Auto Encoding or Masked Language Manipulation or DOBF in Java, and hence we don't need to do any additional pre-training to incorporate language knowledge. Fig. 1 encapsulates the typical training process that would occur for our task. The tokenized input is fed into the model which in turn provides the tokenized output which is used to reconstruct the output. We utilize these pre-trained transformers to perform the seq2seq task of Source Code Generation from Input Test Cases. The model is fine-tuned in a supervised manner until the loss converges.

We would like to benchmark the performance of the Code Generator starting from all of the transformers described in our literature survey. However, considering the time constraints we restrict the scope of our training explorations to CodeT5 [25] and PLBART[1]. Both of these models don't require any additional Denoising Auto Encoding, as this is a MonoLingual Code to Code transformation task, and both of these models have learned representation for java. CodeT5 and PLBART is available on Huggingface, initially we considered using DOBF as well, but considering the complexity of setting up a fine-tuning pipeline without using Huggingface, we decided against the same. These models were fine-tuned to perform optimally for the task of generating source code from test cases. The model architectures and the expected utility will be described briefly in the section below.

3.1 Architectures

3.1.1 PLBART. PLBart was proposed by Wasi et al. [2] This model is very similar to BART. It provides the ability to generate code summarizing, code generation, and code translation. PLBart has the same architecture as $BART_{base}$. BART has 6 encoder layers and 6 decoder layers. The one exception is the addition of normalization layers to both the encoder and decoder [2] is added to the architecture with PLBart.

We will be using its code generation capabilities by implementing the 'plbart-base' pre-trained model. We are making use of pre-trained PLBart due to time limitations and the lack of a very large corpus of data. The model is also an easy import through the Huggingface transformers package.

3.1.2 CodeT5. CodeT5 is a sequence to sequence model that has the same architecture as T5. T5 is an encoder-decoder model made primarily for NL-NL such as English translated to German. CodeT5 takes that same concept but is used for a programming language to programming language transfer as shown in their image Fig. 3. To make this possible the authors trained CodeT5 on CodeSearchNet. We believe this will be useful when transferring tests to code.

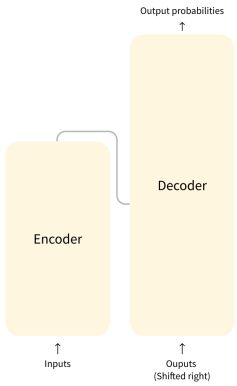


Figure 4: Transformer Architecture

4 CONDITIONAL GENERATION

While most transformers encode the given data into a contextual embedding, we can construct a target sequence by using Conditional Generation techniques. The conditioned language generation is the objective of our task as well, post-generating the contextual embeddings. We utilize Huggingface generators to decode the generated embeddings, which decodes the encoding obtained from conditional generation. The various prominent strategies from the Huggingface library[19] that we considered are listed below.

4.0.1 Greedy. One of the decoders we have considered is the Greedy Search. This method when decoding the sentence performs a search of the weighted objects. Depending on which object has the highest weight associated will be chosen. This seems like an obvious solution for determining the predicted sentence. However, depending on what item is chosen early in the sequence can have a drastic effect on the rest of the formulated output. By being greedy and taking the closest highest value the model can miss the rest of the output. This can result in a repeated output that becomes stuck on certain objects. To be able to reach the other layers other routes have to be tested to see which one has the highest total weight. This is done by adding beams.

This was our preferred method of generating Code2Code output, as according Platen [19], it can generate interesting outputs. Fig. 5 captures the essence of this dilemma perfectly.

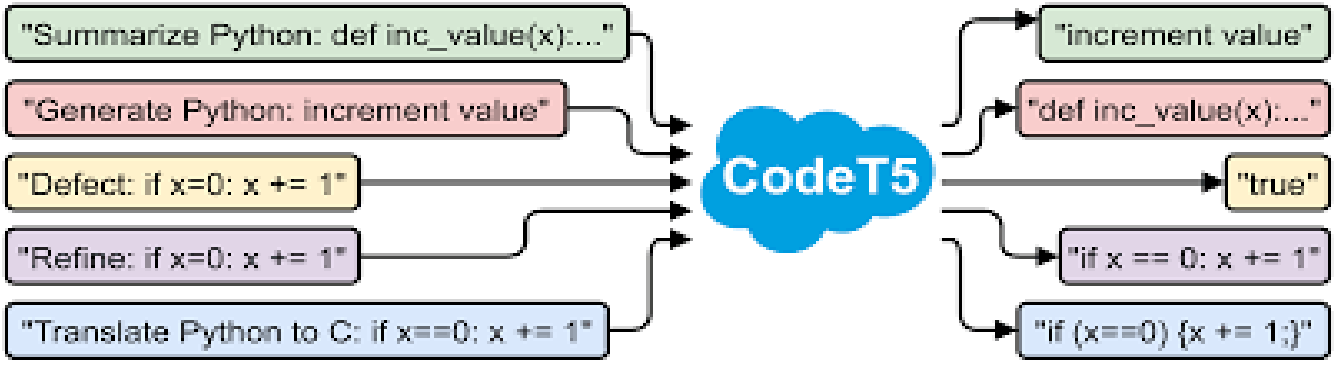


Figure 3: CodeT5 Code Generation [25]

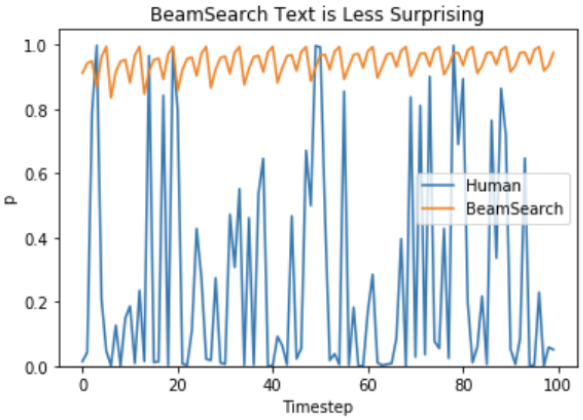


Figure 5: Lack of randomness and surprising outputs [19]

4.0.2 Beams. Beams is an extension of the Greedy Search decoder. The Greedy Search clearly misses opportunities by following what is immediately in front of it. Beams allows for multiple paths to be taken showing which one could have the highest total probability of being correct. The individual beam is calculated by multiplying all the weight probabilities associated with an object on the chosen path. Then this final value is compared to the other chosen beam which will commonly take the next highest value early on using its greedy search.

This does not solve the repeated output issue always. One way to resolve repeated output with beams is by adding an n-gram factor that prevents a certain sequence to be displayed often. However, this would not be viable in software development because we use the same sequences in a multitude of places.

It is to be noted that, we did initially consider Beam search in our proposal as a potential avenue to explore in this project, and we wanted to give it a try despite the fallacy mentioned earlier. However, we decided not to rely on this technique because according to Platen [19], beam search suffers from not being able to generate interesting outputs and since our hypothesis tries to capture interesting output we decided against trying out Beam search given the data

4.0.3 Sampling. Sampling is very different from the previous methods. Sampling selects a random value from a group of conditional probability distributions. This seems to however result in sets of n-grams that make sense but lead to incoherent output as a whole, which kind of dooms its usability in PL tasks, as ideally, we would like to see code outputs that are syntactically, semantically, and structurally sound in Code generation tasks.

4.0.4 TopK. In Natural language processing, Top-K sampling the K most likely next words are filtered and the probability mass distribution is applied to only those K words. This can result in great human-sounding sentences. However, it has a hard time adapting to a dynamic length of context. This could be proven to be difficult with translating to code. Code is very dynamic in length depending on the problem, language, and the varying ways one can solve the same problem.

In practice, our CodeT5 model performed poorly with the greedy sampling approach for output sequence generation, hence we needed a viable alternative that did not strain the code generation pipeline significantly, and thus Top-K sampling was our choice of decoder, which performed slightly better than Greedy. The implications of these results will be discussed in the System section.

4.0.5 TopP. Top-P instead of finding K items takes a probability and sees the smallest number of values from the probability mass distribution can pass it. This continues to redistribute the PMD until a sentence is formed. While this seems very promising, considering our resource limitations, we decided to stick with the aforementioned techniques.

5 EVALUATION TECHNIQUES

The generated source code should be useful for the human programmer. In seq2seq tasks, evaluation of the output would ideally require expert human feedback, but such feedback is impossible to acquire and tenuous if generated manually. However, there are standard techniques with which the quality of generated sequences is ascertained. The BLEU score [18] is one such metric that computes the modified n-gram precision on blocks of text. Also, we would

evaluate the generated output using CodeBLEU [20], which evaluates the underlying Abstract Syntax Tree in addition to using the n-gram similarity metrics inferred by BLEU. These metrics would give a good picture of the sanity of the Source code generated by the models.

However, we should acknowledge that the scope of this project is just limited to generating Source code from Test Cases. In Test-Driven Development, however, good source code will be syntactically correct and will pass the test case provided. In ping-pong pair programming [7], no more source code than required to pass the given set of unit tests will be written. These are important concepts in evaluating the output of these modules because code produced from TDD is highly refactorable. Hence, evaluating the output for its refactorability is vital as well. But, evaluating the refactorability of Source Code is a highly subjective topic with no deterministic expert opinion. Hence, we theorize that if a code development adheres to the base guidelines of Extreme Programming it should be refactorable. Hence, we would like outputs that pass all the unit test cases, and we would expect the unit test cases to have high coverage on the generated code, based on the conventions in a Ping-Pong style pair-programming for TDD. This would be a good specification to look into for future implementation.

While implementing the evaluation pipeline, we realized that CodeBLEU has a very scrupulous requirement that made it significantly hard to use in practice. The CodeBlue implementation provided in CodeXGLUE requires the evaluator and the evaluated code to be of equal length, which makes sense in theory, but in reality, we expected it to skew the results considerably towards Naivecopy since our models generated longer output sequences. Hence, making it necessary to truncate the same to use it.

Thus we settled on utilizing BLEU scores and smoothed BLEU scores. Smoothed BLEU scores are a variant of BLEU scores [8] that can be used in scenarios where there are no n-gram matches. We performed method-2 smoothing, where the n-gram match count starts at 1 instead of at 0. While the BLEU score would help us quantify the performance of our model reliably.

We will also generate predictions for high-quality datasets, and look into the generated codes by each model to get a sense of the nature of representations learned by the models. This will help us to qualitatively analyze the results obtained and considering that we are looking for surprises as a result of our experiments, the qualitative analysis would form the bulk of analysis and reflections.

6 SYSTEM

6.1 Data Preprocessing

The data preprocessing pipeline is fairly simple for the project. We had the SF110 Corpus which had the source code from various repositories with each project exhibiting a unique scaffolding structure. For the first part of the data preprocessing step, we had to crawl the directories and extract the locations of the source files and unit test files of interest, post which we proceed to match the corresponding unit test with its source which we then align suitably into a JSON file. This data format is readily usable for model-finetuning and evaluation.

6.2 Contextual Encodings

Transformers are capable of generating fixed-length contextual embeddings for each token of a given input sequence. Each transformer has a distinct vocabulary, which dictates how the input text would be tokenized. Also, the transformer's tokenizer has special tokens allocated for padding, stop sequence, and other purposes. In the domain of Programming Language Processing, the transformers are made familiar with a target programming language vocabulary with techniques such as Masked Language Modelling and Variational Auto Encoding. However, since both CodeT5 and PLBart were familiar with Java Vocabulary we reused the encoding pipelines of these pre-trained models to produce contextual encoding.

However, given that we have finite memory to train and evaluate our models, we would have to make sure that the number of tokens for a given model cannot be overwhelmingly high. This involves fixing the number of tokens that can be given as input in each input data point. Also, since the number of tokens itself is dependent on the model's vocabulary, the selection of a fixed max token length is a subject and model-specific task. To achieve this we tokenized every training datapoint's input and output. By observing the statistics of each model on this front, we took a number a bit higher 75th percentile of the values as our input or output encoding's max length respectively. Using these values, we configure a tokenizer that correspondingly produced an encoding by truncating or padding the input tokens to produce a fixed number of tokens, which is then encoded to provide a stream of contextual encodings, which successfully encodes the input to feed the model for fine-tuning or the expected output to compute the loss and backpropagate it to fine-tune the model weights.

In addition to this, we split our dataset of 22947 SourceCode-Unittest pairs into train-validation-test sets with a ratio of about 80-18-2. While we did initially want to use 10% of the dataset for testing we resorted to a lower percentage given the resource limitations in generating text for a large number of data points and also during evaluation with BLEU metrics. Considering the resource constraints, we settled for the aforementioned data split, which still left us with a considerable chunk of data points to evaluate the model performance.

6.3 Fine-tuning Transformers

We chose to fine-tune two transformers for the task of Source Code Generation as described earlier. The process of fine-tuning transformers is pretty much the same on a broad level. This involves inorder - tokenization and deciding model specifics, encoding the input sequence, optimizing a pre-trained model, modeling the output embeddings with a decoding strategy of choice with conditional generation strategies followed by an evaluation of the generated conditional sequences using metrics that quantify performance in a human-friendly manner.

The process of fine-tuning involves loading a pre-trained model from the HuggingFace repository. We loaded the models our models from the respective repository for CodeT5 [24] and PLBart [4]. The PL-PL models are built similar to Text-Text models, and hence they inherit various functionalities from the base NLP models such as T5 and Bart respectively. For our implementation, however, we were

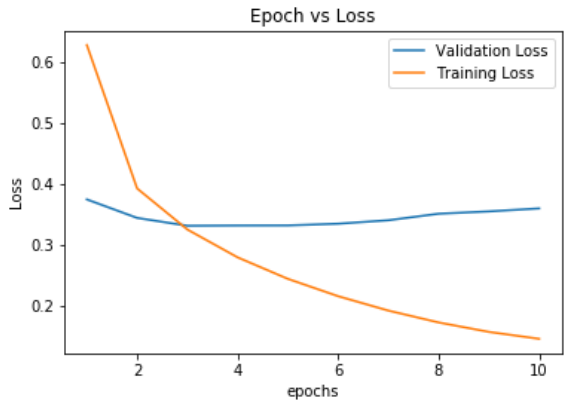


Figure 6: Loss during Fine-tuning Plbart

interested in a ConditionalGeneration module [3], which adds a language modeling head to the respective model, making it capable of performing Seq-Seq tasks.

Both these models had the same pipeline for fine-tuning model parameters. We used PyTorch to load the model weights and optimized the models using an Adam optimizer with a learning rate of 0.0001 and a cross-entropy loss. We backpropagated the loss between generated embeddings and the expected embeddings to fine-tune our model parameters. We trained the model for about 10 epochs in an NVIDIA A100 (80GB VRAM) GPU rented from datacrunch.io. The models took about 5 hours to train each with a batch size of 8 on the aforementioned instance.

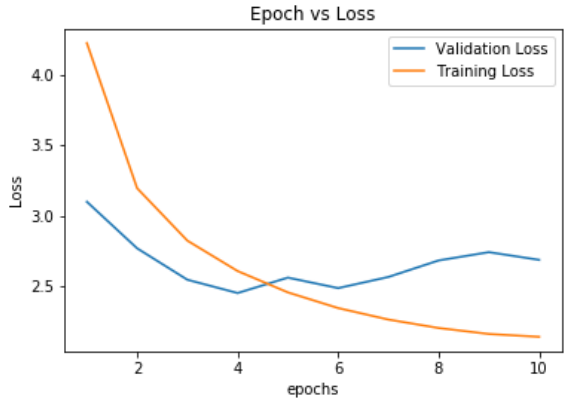


Figure 7: Loss during Fine-tuning CodeT5

6.4 Decoding and Source-Code Generation

The Source Code was to be generated from the stream of encoded tokens. The decoding strategy for obtaining the target sequence from a stream of potential token encodings was discussed in the earlier section. But paying mind to the resource constraints, we were forced to utilize Greedy as our top choice for decoding the output encodings into Code. However, we also used Top-K in CodeT5 when we found that the output encodings were not good. In trying this we wanted to assert if the decoding strategy is the main factor or if the representations learned by the model were perhaps not sophisticated enough to learn good knowledge representations for the task at hand. The Appendix contains the output representations of various models, and it's clear that PLBart is akin to a novice programmer whereas CodeT5 has not learned any useful representations.

7 EVALUATION

The Table[1] contains our average bleu and smoothed bleu scores utilizing the CodeT5 and PlBart, and also a NaiveCopy sequence generator. A naive copy sequence generator predicts the output to be the input. Based on the results quantified, both of our models supposedly perform worse than the Naive Copy algorithm. This is a troubling prospect, but that doesn't mean both of our models have objectively accomplished nothing. By dissecting the results, we see that the Naive Copy performs significantly worse when the data quality rises, indicating that the baseline for interesting models is way below what is set by the Naive Copy. We truly believe that Plbart is an interesting model for the Source Generation problem, by looking at its generation in Appendix A. However, in Test Driven Development a good portion of the test code will be reused in the source code, thus making Naive Copy have a lot higher BLEU score than we would expect. This phenomenon is especially true when we use Automatic test generation tools like Evosuite, where the generated unit test case is generated procedurally from the source code, hence resulting in a high degree of similarity.

The BLEU score for CodeT5 is too low to be considered interesting, in both the Evosuite dataset and the handcrafted dataset. We also tried two decoding strategies to see if there is any relevant learning that takes place and judging by the results we don't see anything interesting by changing the decoding strategy, leading us to believe that the learned representations are not good enough to be helpful in this problem.

8 OUTCOMES AND REFLECTION

Despite the low Bleu scores exhibited by the models, PlBart shows promise as it learned representations akin to how a novice programmer would potentially approach TDD. While there is a lot left desired on the front of syntax correctness, linting, and formatting. The model tried its best to utilize the source code to generate something resembling a source code with pieces of the unit test. This is an interesting implication, which denotes the potential to learn more useful patterns.

As stated in the earlier sections, one of the main goals of this experiment was that given a semi-ideal dataset can we distill the knowledge inherent in an expert system like transformers to accomplish the objective. By looking at the evaluation results in the high-quality dataset, we can infer that PlBart has distilled that expertise, while CodeT5 failed to achieve the objective. The model became fixated on predicting code comments instead of predicting code blocks. We tried different decoding techniques to see if code generation is present in the output space, but our efforts were in vain as indicated by the outputs, since the output space seems to be entirely predicting the license comments of the source code.

Table 1: Describing the main Features used in Classification

Model	DecodingStrategy	Source	BLEU	Smoothed BLEU
Code-T5-Small	Greedy	Evosuite	0.3343	0.025
PLBart-Base	Greedy	Evosuite	0.4286	0.1295
Code-T5-Small	Top-K	Evosuite	0.3554	0.0775
Naive-Copy	Greedy	Evosuite	0.4582	0.186
Code-T5-Small	Greedy	Manual	0.2711	0.0433
PLBart-Base	Greedy	Manual	0.4122	0.1037
Code-T5-Small	Top-K	Manual	0.3450	0.0983
Naive-Copy	Greedy	Manual	0.4597	0.1318

This could be an effect of truncation of source code tokens in the ground truth, as code blocks starting with humongous comment blocks, effectively poisoned the model leading it to a bad parameter horizon. Our inability to decode useful information using various likelihood estimation techniques leads us to believe that the model representations are likely not good enough as suggested by Wellect et al. [26]

The way forward is to fine-tune PlBart either using more training with larger datasets, by utilizing only quality datasets, or by novel changes in the architecture. Looking back, however, we would not like to explore a model that cannot effectively distill domain knowledge. While CodeT5 has been shown to do well, on PL-PL tasks as described in earlier sections, we also could have used the larger variants of CodeT5 like 'CodeT5-base' and 'CodeT5-large', as they support larger d_{max} values. This could be the difference-maker, as we believe the reason for CodeT5's worse performance was that it wasn't able to look farther than the license comments for many of the input data, and hence learned to predict comments since it probably never saw a lot of code blocks, to begin with. Since we were already operating in huge memory loads increasing max encoding length was not a practical option, but dropping abnormally large training data points was, which we would have hoped to try if we had more compute time and resources.

We had access to a searchable corpus of code from java, and generating a better quality human written aligned source code to unit test was possible, and this could potentially ensure our model training space stays in the PL domain and doesn't morph into a code comment generation space. We also considered increasing the max input and output embeddings themselves, but decided against it, as ideally in the ping-pong programming, the source code is written iteratively, so only a small amount of source code is generated each iteration, and hence the max encoding size should not be unrealistically high. However, we could have forced more attention on the parts of the training data that matter by removing comments and perhaps even removing linting altogether, since we are only concerned about code generation and optimal code can be linted with little to no effort with the aid of modern programming tools. This phenomenon could also be said of the import files, but they can be a vital piece of the source code conveying strong domain context. But in a broad sense, attention could be forced on parts that matter by removing code blocks irrelevant to the problem domain.

8.1 Lessons Learnt

FFrom the course perspective, we were able to see the significance of attention and how it dictates the output sequences. CodeT5 failed to learn a representation that focused its attention on the relevant section of the sequences, as it predominantly predicted only Code comments instead of source code. The PlBart model fared extremely well on this front, as it seemed to generate representations that had attention on vital parts of the problem to tackle, and even when comments were predicted, they seemingly contained sections potentially used by the scaffolding tool to generate code during compilation, like the prediction of the '@author' keyword from the appendix which has useful connotations when working with frameworks. Another takeaway message was that attention is supposedly a self-supervised learning objective, but while we explored techniques like Masked Language Modeling and Denoised Auto Encoders they are usually generalized techniques to improve representations, and when they fail we are left empty-handed. This project helped us realize that this is not always the case, as we could have done many things like the ones discussed in the earlier section to regularize the inputs and force the model to have optimal attention.

In a practical sense, we learned about Seq2Seq modeling using Transformers, as we successfully managed to train and evaluate multiple transformers. While the results can always be better, we are extremely elated by the performance of the Plbart model in this task, and we truly believe by looking at various Source codes that were generated, the model performs akin to a novice pair programmer, and with iterative improvement in the data, training strategies we are confident that this can evolve into a viable AI pair programmer.

The things we hope to learn are better ways to quantify our results. Despite our attempts to qualify our results, without a good quantifying metric to boot, it's hard to improve model performance iteratively. Since the naive copy is the most basic baseline one could come up with and considering this problem horizon it can be potentially hard to beat even for transformers with good learning representations. However, this is in the purview of our knowledge domain on how to evaluate the models but this is a horizon we should expand upon significantly.

8.2 Broader Impact

The impact of successfully generating test cases still stands which is a potential use case in ping-pong pairing. Thus the direct outcome of this project is an IDE plugin that any developer can potentially

install, write unit tests, and get a boilerplate code. This tool can be expanded upon, by making it generate source code by focusing on a small set of test cases at a point in time and generating the corresponding source code block. With the enhanced quality of the underlying models, the confidence of the generated source code improves significantly making the programmer necessary only to drive the development with test cases. This can make training programmers in new codebases significantly easier, as pair programming is one of the most vital strategies used by modern companies to train junior developers, and instead of relying on a senior programmer, we can procedurally onboard [7]. According to Böckeler et al. [7], there are power dynamics in play that can impact developer confidence, and this can be overcome by supplementing the human with an AI.

While TDD has a lot of implications in Quality Assurance, we truly believe the applicability of this project is not just limited to that domain. In fact, the most ambitious iteration of this project would involve directly converting a given Functional Specification Document into a codebase, complete with fully functional and high-quality unit tests. This doesn't just significantly ramp up coding delivery pipelines, but also provides unparalleled quality assurance. With game-theoretic modeling, we can potentially model an entity that generates unit test cases iteratively from a Functional Specification Document, and the pairing AI agent generates the source code. This is done iteratively until both the models converge with satisfying accuracy. But, in essence, these models can define the development pipelines of tomorrow, and potentially today!

9 CONCLUSION

In conclusion, we have built two models utilizing the base transformers PlBarT and CodeT5 for the task of source code generation from the test case. On fine-tuning these models, we observed that PlBart worked akin to a novice programmer while the CodeT5-based generator worked extremely poorly potentially due to the bad model representations. The quality of our dataset is not palatable for this task, and our primary emphasis was to investigate interesting phenomena rather than fine-tune performance, since we posit there will be a lack of the same. Both of our models performed worse than Naive copy as we hypothesized, but on inspecting the sequences generated by PlBart, it was evident that there was some learning, and its performance is akin to a novice programmer. The Naive-copy baseline doesn't do justice to measure interestingness because of the data horizon resulting from utilizing test-case generators, because they generate unit test files that are highly similar to the source code. The CodeT5 model failed to pay attention to the vital parts of the input code as it predicted the initial Code comments about licensing more than the actual code. This gave us key insights into how Transformer fine-tuning would look like, and also helped us understand and ideate how to take this solution to the next iteration.

10 DISCUSSIONS

The reviews we received for this project during the proposal talked about utilizing better evaluation metrics, and after trying out multiple avenues even we feel the need for the same, and quantifiable metrics failed to capture interestingness well. However, we did qualitatively evaluate the results which we hope addressed this concern.

11 CONTRIBUTIONS

Harish Ravi worked on Data Generation. Both the authors contributed equally to the Model fine-tuning and Model evaluation. Jonathan worked on setting up conditional generation pipelines. Also, both the authors contributed equally to the report and presentation.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://www.aclweb.org/anthology/2021.naacl-main.211

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. *CoRR* abs/2103.06333 (2021). arXiv:2103.06333 https://arxiv.org/abs/2103.06333

[3] Wasi Ahmed. 2022. Plbart. https://huggingface.co/docs/transformers/model_doc/plbart#transformers.PLBartForConditionalGeneration

[4] Wasi Ahmed. 2022. UCLANLP/plbart-base · hugging face. https://huggingface.co/uclanlp/plbart-base

[5] Marie anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (Eds.). https://openreview.net/forum?id=3ez9BSHTNT

[6] Kent Beck. 2015. *Test-driven development by example*. Addison-Wesley.

[7] Birgitta Böckeler and Nina Siessegger. 2020. On pair programming. https://martinfowler.com/articles/on-pair-programming.html

[8] Boxing Chen and Colin Cherry. 2014. A Systematic Comparison of Smoothing Techniques for Sentence-Level BLEU. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Baltimore, Maryland, USA, 362–367. https://doi.org/10.3115/v1/W14-3346

[9] Oluwaseun Alexander Dada and Ismaila Temitayo Sanusi. 2021. The adoption of Software Engineering practices in a Scrum environment. *African Journal of Science, Technology, Innovation and Development* 0, 0 (2021), 1–18. https://doi.org/10.1080/20421338.2021.1955431 arXiv:https://doi.org/10.1080/20421338.2021.1955431

[10] Gordon Fraser and Andrea Arcuri. 2013. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* 20 (2013), 783–812.

[11] Gordon Fraser and Andrea Arcuri. 2014. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[12] et al Hendisantika. 2022. Hendisantika/spring-boot-unit-test: Spring Boot Unit Test Example. https://github.com/hendisantika/spring-boot-unit-test

[13] Felipe Hoffa. 2017. All the open source code in github now shared within BigQuery: Analyze all the code! https://hoffa.medium.com/github-on-bigquery-analyze-all-the-code-b3576fd2b150

[14] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 5110–5121. https://proceedings.mlr.press/v119/kanade20a.html

[15] Marie-Anne Lachaux, Baptiste Rozière, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. *CoRR* abs/2006.03511 (2020). arXiv:2006.03511 https://arxiv.org/abs/2006.03511

[16] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. https://doi.org/10.48550/ARXIV.2102.04664

[17] PhD Michael Larionov. 2019. Machine learning and test-driven development. https://towardsdatascience.com/machine-learning-and-test-driven-development-6b5d793b1783

[18] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA,

311–318. https://doi.org/10.3115/1073083.1073135

[19] Patrick Von Platen. 2020. How to generate text: Using different decoding methods for language generation with Transformers. https://huggingface.co/blog/how-to-generate

[20] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020). arXiv:2009.10297 https://arxiv.org/abs/2009.10297

[21] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. https://doi.org/10.48550/ARXIV.2110.06773

[22] et al Tomekkaczanowski. 2013. Tomekkaczanowski/Junit-Put-Examples: Code examples for "practical unit testing with junit and Mockito" book. https://github.com/tomekkaczanowski/junit-put-examples

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[24] Yue Wang. 2022. Salesforce/Codet5-small · hugging face. https://huggingface.co/Salesforce/codet5-small

[25] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR* abs/2109.00859 (2021). arXiv:2109.00859 https://arxiv.org/abs/2109.00859

[26] Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. 2019. Neural Text Generation with Unlikelihood Training. *CoRR* abs/1908.04319 (2019). arXiv:1908.04319 http://arxiv.org/abs/1908.04319

Appendices

A CODE EVALUATION

Input

```
package com.example.util;

import com.example.model.ToDo;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class PayloadValidatorTest {

@Test
public void validatePayLoad() {
ToDo toDo = new ToDo(1, "Sample ToDo 1", true);
assertEquals(false,
PayloadValidator.validateCreatePayload(toDo));
}

@Test
public void validateInvalidPayLoad() {
ToDo toDo = new ToDo(0, "Sample ToDo 1", true);
assertEquals(true,
PayloadValidator.validateCreatePayload(toDo));
}


}
```

Predicted

```
import.text.Message;
import.util.ArrayList;
import.util.List;
import.util.Validator;
import com.mythoughtworks.xstream.annotations.XStreamException;
import com.mythoughtworks.xstream.annotations.ValidationException;
/** * * @author MyEclipse Hibernate framework */
public class ToDoValidator {
public static void validateInvalidPayLoad(boolean validate, String message)
{
if (validate) {
validateInvalidPayLoad(message);
}
}
/** * * @param validate */
public static void validateInvalidPayLoad(boolean validate, String message)
{
if (validate) {
validateInvalidPayLoad(message);
}
}
/** * * @param validate */
public static void validateInvalidPayEquals(boolean validate, String
message) {
if (validate) {
validateInvalidPayEquals(message);
}
}
}
```

```
package com.example.util;

import com.example.model.ToDo;

public class PayloadValidator {

public static boolean validateCreatePayload(ToDo toDo) {
return toDo.getId() <= 0;
}

}
```

Expected

Figure 8: PLBart Evaluation