Code-to-Code translation from Java to Python using CodeBERT and CodeT5 models

Arnav Chhabra
arnavc@vt.edu
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, USA

Harsha Vardhan Reddy Bonthu
harshareddy97@vt.edu
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, USA

Abstract

There has been tremendous growth in applications of Deep learning research in software engineering domain. With new programming languages being developed every day, new functionalities and security features are being developed. A lot of time is being spent by the developer in trying to convert these old legacy systems into modern code. So automating this code translation process has become critical to helping developers. In this paper, we look at two state-of-art models: **CodeBERT** and **CodeT5** model which are capable of performing a plethora of software engineering tasks. We propose a novel technique to implement code translation from Java to Python using these pre-trained models. These models are not natively configured for such a code translation, thereby we engineer these models for the task at hand. We evaluate the results of these machine learning models to see how well they perform and discuss the reasons behind their performance and go over an in-depth analysis of the working (including pre-training) of these models and compare with the Facebook's **Transconder** model. We conclude by assessing and looking at how the future looks for code-to-code translation software engineering tasks using Deep Learning and ways we can improve our models.

Keywords: Transfer Learning, Code Translation, Java, Python, CodeBERT, CodeT5, Software Engineering

ACM Reference Format:
Arnav Chhabra and Harsha Vardhan Reddy Bonthu. 2022. Code-to-Code translation from Java to Python using CodeBERT and CodeT5 models. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (CS 5814 Project Proposal).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

CS 5814 Project Proposal, April , 2022, Blacskburg, VA

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

1 Introduction

Deep learning research in software engineering has been growing more and more over the recent years. There are many advantages of using deep learning techniques to solve the problems being faced by developers in development tasks. These tasks range from bugs and error detection, data security, code summarization and documentation translation. All these tasks are important to solve using deep learning to help so that they can help developers solve these issues. Helping automate these tasks can help developers speed up the development process and also help create more robust and quality solutions. This deep learning techniques will help avoid the developers eliminate a lot of work which is currently being done manually like adding comments to the code or even finding certain bugs which can be automated using various deep learning techniques. Automating these tasks will help developers to stop spending time on issues like comment generation, bug detection, code translation and spend more time on able to develop new and exciting features which will improve their products greatly.

With there being new programming language being developed everyday and they can be used with various new functionalities. With this advancement of technologies code to code translation has become a topic of vast importation to update the legacy software in accordance with modern standards. With new languages like Kotlin, Golang, Rust etc coming up to meet the new software needs across different various verticals in the technology sectors. The amount of effort it takes for these conversion of legacy systems is monumental in nature taking lots of time and money to complete these tasks. So to automate this task of code translation is something that will greatly help developers in reducing their efforts in the conversions of these source code from the legacy systems to the new modern age systems. Automating these tasks will greatly help developers in reducing the time and effort required to translate this source but also deep learning techniques can be used to ensure that is correct and also help testing the product to ensure that the product is implemented correctly.

In this paper we look at the source code conversion from various sources from Java to Python using the deep learning models CodeBert and CodeT5. In this paper we will look into the different source code translation techniques in the

section 2. In section 3 we look at the various sources of the dataset including the structure of the dataset. In section 4 we look at the two models into detail regarding their architecture and how they are implemented. With section 5 being regarding the evaluation and results we got with the source code translation of the models. Section 6 looks at the impact these models can have in the real world and how they can be used. Section 7 being the conclusion section which contains the final thoughts regarding the results observed with the source code translation model.

2 Literature Survey

In computing history not many source to source translators have been created given the complexity and nuances in order to accomplish the task. Cfront was the original compiler which converted code from C++ to C, developed by Bjarne Stroustrup. As Cfront was written in C++ [15], it was a challenge to bootstrap on a machine without a C++ compiler/translator. Facebook had developed HipHop[3] which translated PHP to C++ in 2010. Haxe[5] was developed in 2006 with an effort to have a high-level cross-platform programming language and compiler that can produce applications and source code, for many different computing platforms from one code-base. It could compile into multiple languages. Google also launched J2ObjC[7] which translated from Java to Objective-C.

In the earlier days there were a limit on the resources available in terms of GPUs and the amount of source code available. Therefore the parallel training of the machine learning models which was required to complete source code translation related tasks. To counter this issue at the time Facebook research created a model to implement a machine translation system that works only on a single multilingual corpa.[8] They propose a model which takes two different programming language implementations and transform them into the same space. By learning converting the languages back from this shared space the machine learning model is able to complete code translation tasks without having to require labelled data to perform the code translation tasks. This model helps them tackle the problems of the limited computation power and the lack of data.

With the advancement of the amount of source code data from sources like GitHub, LeetCode and other online repositories has enabled us to have enough data to train bigger and deeper machine learning models. With the advancement of deep learning technologies like transformers, autoencoders in recent times has helped machine learning models understand source code in better representation help improving the quality of the source code translations being generated by the machine learning models. Also the increase of the GPU technologies and has helped increase the parallel training capabilities for which the models can be trained to help the models learn the programming languages in a better way which helps in the different source code related tasks. As all these tasks can be completed with higher accuracy if the model is able to understand the syntactic and semantic meaning of the programming languages. With this vast increase in resources the parallel approach has become more practically possible to implement thus help implement more complete machine learning models that can be used across various languages.

In the Facebook transcoder[10] approach taken by the authors is to create an unsupervised multilingual source code translation approach to improve on the accuracy and readability of the source code being generated by the earlier models. They look into improving the existing rule-based and neural network based approaches to code translation with the help of the advancement of various new cutting edge deep learning techniques. They convert the source code between Java, C++ and python to ensure with a high accuracy with the help of a monolingual source code to ensure that they do not require expertise in source or the target language. With the help of this approach the code can be used across different programming languages.

In the Plbart[1] paper the authors take a different approach to code translation where they consider a sequence-2-model for the code translation task. They train their models on Java and Python functions. For their code translation task when the use the model they consider Java and C# as the languages being used to complete the source code translation tasks. The reasoning given behind not training the model on C# by the authors is that Java and C# are programming languages that have a lot of syntactic similarity. So when pre-training the model using the Java programming languages the model can learn the different syntactic characteristics of Java and C#. So when using the model to convert Java and C# code will be quite effective as the model knows the syntactic nuances of the Java language which is quite similar to C#. They observed good results for this translation task despite not training the model not explicitly being trained using the C# programming language. This proved that their pre trained model actually understand the syntactic and semantic nuances of the Java programming language which in turn helped it to understand a quite similar language in C#.

In the CodeBert [4] approach to code translation the authors decided to use a transformer based architecture model to complete the code translation tasks. The CodeBert model is pre-trained in such a way that it uses both multi modal and bimodal data. It is trained in such a way to provide plausible alternatives to generator based models. Helping the generators to perform better when actually implementing the tasks being implemented using CodeBert models. This helps the CodeBert model in source code translation task as the generator is being pre trained to understand the tokens being sent it to during the pre training phase and therefore helps it perform better when converting source code. The CodeBert model performs well on even unseen

programming languages for various tasks which shows that shows the model actually understands the programming languages that it is pre-trained on and is not a rule based system which only can perform tasks on the programming languages which it is trained on.

In the CodeT5 [16] implementation of code translation they consider a different approach of only doing an encoder pre-training or the decoder pre-training. They consider both the components to be pre-trained with different programming languages which will in turn help it perform better when performing the various tasks like code translation, bug detection and source code translation tasks etc. They also use a novel approach which helps the model help identify certain source codes and to be able to identify them when they are masked in nature. Thus helping the model to not only understand the source code but helps it generate better source code.

3 Data

In deep learning, often the accuracy of the model depends on the quality of the dataset upon which it's trained. Therefore, it was imperative to get a high quality dataset, so that our models can be trained to maximum efficiency. As we were working in Code-to-Code translation domain, we needed similar code functions in both Java and Python. Finding such a data was not an easy task, because people often write a function in only one language and writing the same one in another is usually not required at all. We then resorted to competitive coding platforms, where solutions to a set of algorithmic questions are solved by numerous users in multiple languages. We then choose the following avenues to get our data from:

- GeeksforGeeks
- Leetcode
- Codeforces
- Atcoder
- Google CodeJam
- Project Euler

We got the Google Code Jam and Atcoder based on the paper written by Nafi and his team [11]. The Leetcode [6] and Project Euler [12] data were taken from public Github repositories. We crawled the sites GeeksforGeeks and CodeForces to get their data. The statistics of the data can be seen in the following table.

Source	Problem	Java		Python		Pairs
		#Soln	Avg.L	#Soln	Avg.L	
Atcoder	716	3,294	260.9	3,505	154.3	16,305
CodeJam	126	494	375.2	598	310.8	2,423
Codeforces	2,242	7,801	191.7	9,368	120.8	33,295
GeeksforGeeks	5,111	5,111	194.5	5,111	167.0	5,111
Leetcode	118	118	145.1	118	129.6	118
Project Euler	162	162	227.4	162	169.1	162
Total	8,475	16,890	-	18,862	-	57,414

Table 1. Description of the data collected [2]

3.1 Preprocessing

We begin by tokenizing the solution code and removing docstrings and comments. For Java, we utilize the javalang tokenizer, while for Python, we use the standard library tokenizer. We filter away solutions that are longer than a specified length threshold (= 464) after tokenization. Each challenge has [1 – 20] approved solutions during the initial data gathering. We refer to the AVATAR [2] dataset to identify and model our preprocessing accordingly. To enhance diversity among solutions to the same problem, we want to keep the solutions that are the most different from each other. We chose five solutions that differ the most from others using the open source library in Java and dfflib in Python.

4 Model Description

We have chosen to employ the following two models:

- CodeT5 [16]
- CodeBERT [4]

We will compare it with baseline of the following model:

- Transcoder [10]

4.1 CodeT5

CodeT5 builds on an encoder-decoder framework with the same architecture as T5 [14]. It aims to achieve generic representations for both programming language (PL) and natural language (NL) by utilizing pre-training on unlabeled source code. As explained in the Figure 3 CodeT5 extends the noising Seq2Seq objective in T5 by proposing two identifier tagging and prediction tasks to enable the model to better leverage the token type information from PL, which are the identifiers assigned by developers code [14].

4.1.1 Pre-Training Tasks of CodeT5. CodeT5 has been pre-trained on 8.35 million functions in 8 programming languages (Python, Java, JavaScript, PHP, Ruby, Go, C, and C#). We will use the Code-T5 base checkpoint for our translation task of Java code to Python code. Let's take a look into the pre-training tasks of CodeT5 to understand it's inner working and the reason why it's so powerful:

- **Masked Span Prediction (MSP)** : Typically, the denoising aim distorts the source sequence with noising operations before requiring the decoder to restore the original messages. CodeT5 uses a span masking goal similar to T5 [14] that arbitrarily masks spans of varying lengths and then anticipates these masked spans at the decoder with the help of some sentinel tokens. [16]
- **Identifier Tagging (IT)** : It seeks to inform the model if this code token is an identifier or not, in the same

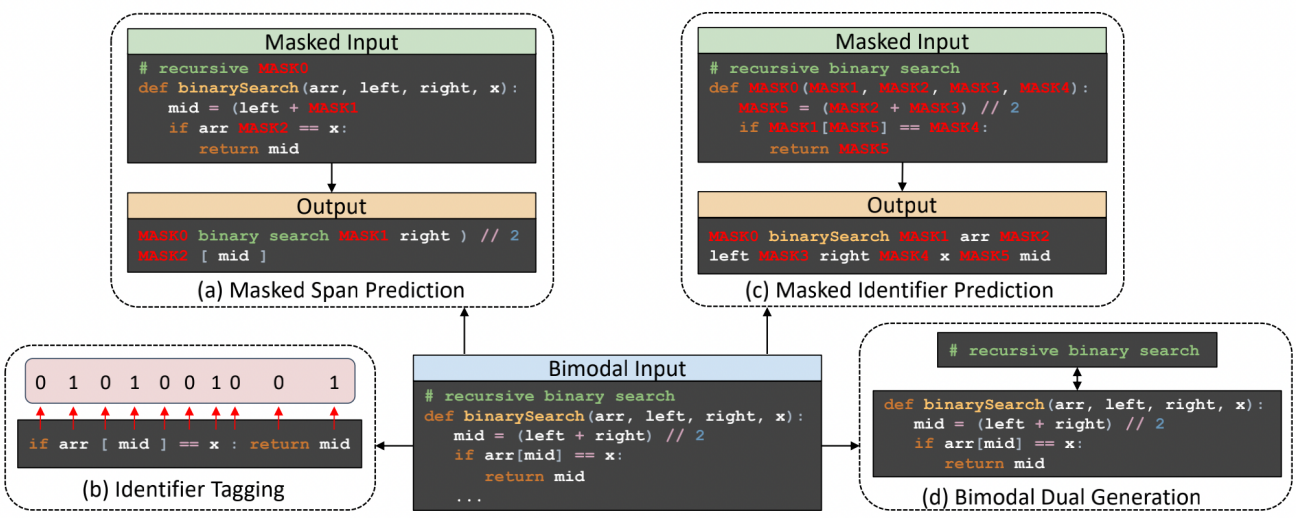

Figure 1. Pre-training tasks of CodeT5 [16]

way that syntax highlighting in some developer-assisted tools does.

- **Masked Identifier Prediction (MIP)**: In the PL section, CodeT5 masks all identifiers and uses a single sentinel token for all occurrences of a single identifier. In CodeT5's identifier-aware denoising pre-training, it alternately optimizes these three losses with an equal likelihood.
- **Bimodal Dual Generation** : CodeT5 leverages the NL-PL bimodal data to train the model for bidirectional conversion to bridge the gap between pre-training and fine-tuning. It uses two training instances with opposite orientations and add language ids for each NL and PL bimodal datapoints. This operation can also be thought of as a variant of T5's span masking, as it masks the entire NL or PL segment from the bimodal inputs. The goal of this job is to better align the NL and PL counterparts. [16]

4.1.2 Fine Tuning Tasks of CodeT5. We will run the CodeT5 model which reads 40423 records with the following parameters:

- Average Source length: 220
- Average Target length: 145
- Maximum Source length: 548
- Maximum Target length: 3400

After performing the tokenization process, we get to the following parameters:

- Average Source length: 236
- Average Target length: 180
- Maximum Source length: 3029
- Maximum Target length: 5646

4.1.3 Multi-task Learning and Bimodal Dual Generation Effects. The impacts of bimodal dual generation during pre-training and multi-task learning during fine-tuning can be investigated for our project. On both CodeT5-small and CodeT5-base, the bimodal pre-training leads to regular enhancements in task generation and code summarization. Nevertheless, for PL-PL creation and comprehension tasks, this pre-training activity may not benefit and may even harm efficiency. We believe this is because bimodal dual generation teaches an improved alignment between PL and NL, which enhances both PL and NL tasks in the first place. As a natural consequence, this goal may cause the model to be biased toward PL-NL tasks, affecting its efficacy on PL-PL functions.

4.1.4 Ethical Concern with CodeT5. While CodeT5 has the promise to be a useful tool for automating code to code translation, there are certain ethical concerns to be aware of. According to the CodeT5 team, the following risks are still being worked on:

- **Bias for Automation**: The software may provide capabilities that appear to be proper on the surface but do not correspond to the developer's intentions. We do want to point out that these concerns are cautionary and doesn't cause any major side-effects. If developers follow these wrong code ideas, the schema may be harmed, and troubleshooting will take significantly longer, with serious safety implications.
- **Security concern**: Some confidential material from the training examples may be encoded by pre-trained models. It's likely that the program won't be able to delete all of the sensitive data and will instead generate code that harms the program.

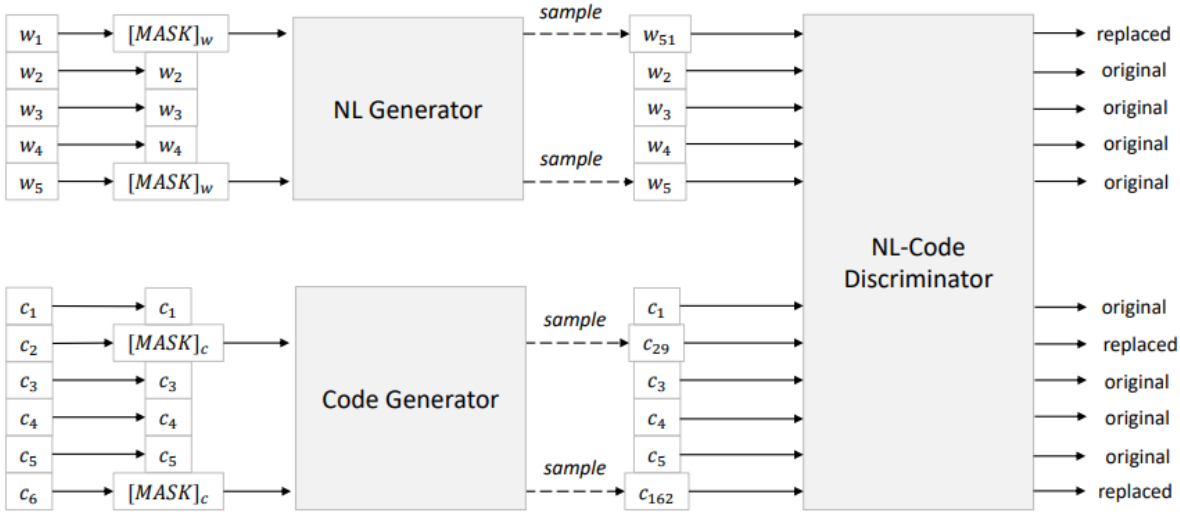

Figure 2. CodeBert Architecture [4]

4.2 CodeBERT

CodeBERT is a pre-trained model for programming language, which is a multi-programming-lingual model pre-trained on NL-PL pairs in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go). CodeBERT captures the semantic relationship between natural language and programming language and generates general-purpose representations that can be used to support NL-PL comprehension and generation tasks. CodeBERT follows BERT and RoBERTa, and uses multi-layer bidirectional Transformer as the model architecture of CodeBERT.

4.2.1 Pre-Training tasks of CodeBERT. We describe the two objectives used for training CodeBERT as follows:

- **Masked Language Modeling (MLM)**: It applies masked language modeling on bimodal data of NL-PL pairs. The MLM objective is to predict the original to- kens which are masked out, formulated as follows, where pd is the discriminator which predicts a token from a large vocabulary. The final loss function is given by :

  $\min_\theta(L_{MLM}(\theta) + L_{RTD}(\theta))$

- **Replaced Token Detection (RTD)**: Only bimodal data (i.e. data points from NL-PL pairs) is used for training in the MLM aim. The goal of replacement token detection is presented below. The RTD goal was created with the intention of learning a pre-trained model for natural language quickly. We adapt it for our circumstance, with the benefit of being able to train with both bimodal and unimodal data. The discriminator is programmed to determine whether or not a word is the original, which is a binary classification problem. The RTD aim applies to every place in the input, and it varies from the GAN (generative adversarial network) in that if a generator produces the proper token, the label of that token is "real" rather than "fake." The generators can be implemented in a variety of ways. CodeBERT creates two efficient n-gram language models with bidirectional contexts, one for NL and the other for PL, and learns them from unimodel datapoints. The method can be easily extended to learn bimodal generators or more complex generators, such as Transformer-based neural architecture, which is learned jointly. [4]

4.2.2 Fine Tuning Tasks of CodeBert. We run the CodeBert machine learning model with 40423 records with the following parameters:

1. Average Source Length: 220
2. Average Source Length: 145
3. Maximum Source Length: 548
4. Maximum Target Length: 3400

For the pre-processing of the data before sending it to the CodeBert model the functions are tokenized and then set with a max length based on the parameters provided while running the training/testing functionalities. There the record parameters will be set to the following parameters:

1. Average Source Length: 256
2. Average Source Length: 256
3. Maximum Source Length: 256
4. Maximum Target Length: 256

4.2.3 NL-PL Probing. On the NL side, CodeBERT choses NL-PL couples with one of the six keywords (max, maximum,

min, minimize, less, greater) in their NL documentations and arrange them into four possibilities by combining the first two keywords and the middle two keywords. The goal is to get pre-trained models to pick the correct one out of three alternatives. That is, the whole code and disguised NL documentation are included in this setting's input. The objective is to choose the correct answer from four options. [4]

CodeBERT chooses codes containing the phrases max and min for the PL side, and we design the job as a two-choice response selection issue. The purpose is to choose the correct solution from two choices, which includes entire NL documentation and a disguised PL code. Because code completion is a critical case, we'd like to see how well the model can anticipate the proper token based on previous PL situations. As a result, we introduce a new setting for the PL side, where the input comprises the whole NL documentation as well as any previous PL codes.

4.3 Transcoder

TransCoder employs a sequence-to-sequence (seq2seq) model with attention, which consists of a transformer-based encoder and decoder. For all programming languages, it employs a single shared model. It is trained utilizing the initialization, language modeling, and back-translation methods of unsupervised machine translation.

We can see this in the following figure.

4.3.1 Pre-training with Transcoder.

- **Initialization (Pre-training)**: The model's cross-lingual nature stems from the large number of shared tokens (anchor points) that occur between dialects. The anchor points in English-French translation are mostly numerals, as well as city and person names. Common keywords (e.g. for, while, if, try) as well as numerals, mathematical operators, and English strings that exist in the source code serve as anchor points in programming languages. [10] Transcoder analyzes an incoming packet of source code segments for the masked language modeling (MLM) objective at each cycle, arbitrarily mask out a few of the tokens, and train TransCoder to predict the tokens that were masked out depending on their circumstances.
- **Denoising auto-encoding** : The seq2seq model can build high-quality approximations of input data thanks to XLM pretraining. However, because it has not been taught to decode a sequence based on a source representation, the decoder lacks the ability to translate. To overcome this problem, we use a Denoising Auto-Encoding (DAE) objective to train the model to encode and decode sequences. The DAE goal works similarly to a supervised machine translation approach, in which the model is trained to predict a sequence of tokens from a corrupted version of that sequence. We employ the same noise model presented by Lample [9] to corrupt a sequence. We mask, delete, and shuffle input tokens at random. A specific token denoting the outgoing programming language is the very first symbol supplied as input to the decoder. The model can encrypt a Python sequence and decode it using the Java start symbol to create a Java translation at test time. The decoder will correctly create this Java translation if the Python function as well as a legitimate Java translation are both assigned to the same latent representation by the encoder. While the encoder output is messy, the DAE goal teaches the model's "language modeling" element, i.e. the decoder has always been trained to create a correct result. [10]
- **Back translation**: In reality, only using XLM pretraining and denoising auto-encoding to create versions is sufficient. The model is never taught to do what it is supposed to do at testing phase, which is to convert function from one language to another, hence the efficiency of these conversions is usually poor. To solve this problem, we employ back-translation, which is one of the most successful ways for leveraging monolingual data in a poorly supervised situation. Back-translation was first used to boost the effectiveness of machine translation in a supervised scenario, but it has since shown to be a crucial part of unsupervised machine translation. A source-to-target model is paired with a backwards target-to-source trained model in parallel in the unsupervised situation. The target-to-source model is used to convert target sequences into origin sequences, resulting in messy source sequences that match the actual truth target sequences. The source-to-target model is then trained to recreate target sequences from incorrect source sequences created by the target-to-source model in a poorly supervised manner, and vice versa. Both model can be trained in parallel until they reach a point of convergence.

5 Evaluation and Results

The most common metrics used for source code are BLEU scores and reference match. The BLEU (Bi-Lingual Evaluation Understudy) method assesses the quality of machine-translated text through one natural language to another. The key notion underpinning BLEU is that quality is defined as the correlation between a machine's output and those of a person: "the closer a machine translation is to a professional human translation, the better it is." [13] BLEU was among the first metrics to claim a strong correlation with human quality judgments, and it is still one of the most used automated and low-cost measures today.With BLEU scores used to find the the syntactical overlap of tokens between

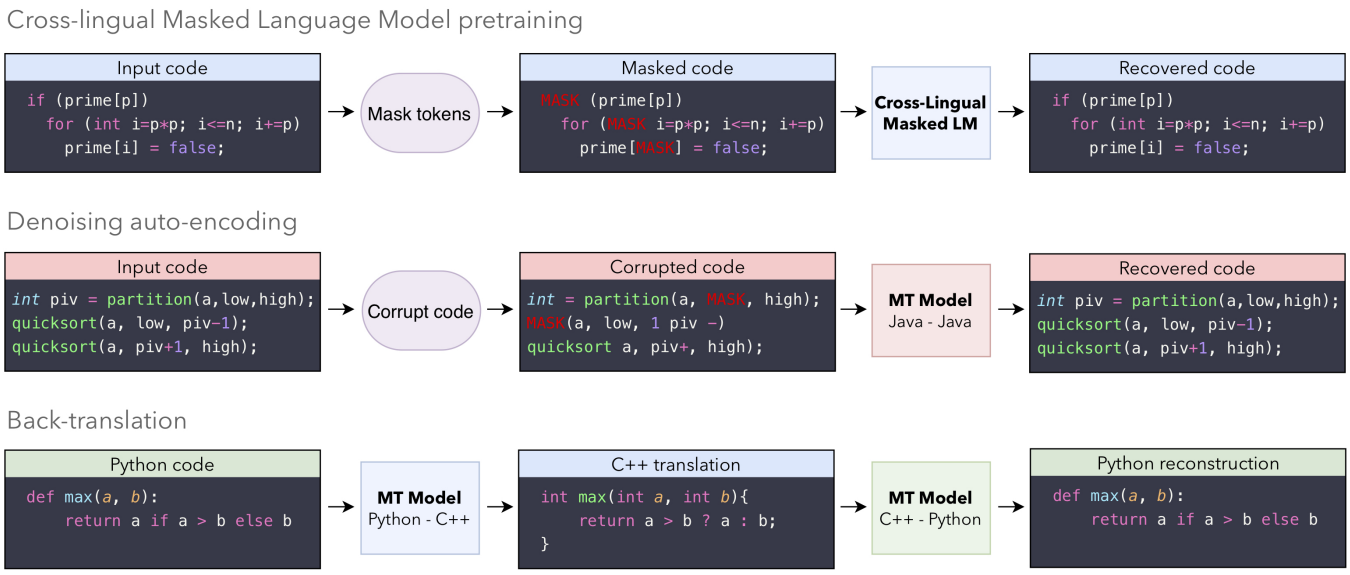

Figure 3. Illustration of the three principles of unsupervised learning by Transcoder [10]

the target source and the generated source code. While reference match is a metric that calculates the percentage of generation that perfectly match the ground truth. These two metrics do perform well for natural text and help us evaluate how well the generation model is performing. There is one issue with these models is that they do not take into account the result/output of the code itself. Adding another metric to this would be to consider the computational accuracy of the generations by the model. This takes into the account the actual code itself and will classify the generation as correct if the output is same to output of the code of the target.

Let's take a lot at the BLEU scores for both the state-of-art Transcoder model and the models that we have run. Table 2 clearly shows that our CodeT5 model has outperformed the CodeBERT and even though with limited GPU resources has given a tight competition to Facebook's base Transcoder model.

Model	Details	BLEU Score
Transcoder	Greedy decoding	68.1
CodeT5	Encoder-Decoder	52.91
CodeBERT	bimodal	23.0

Table 2. BLEU scores comparison

5.1 Comparison of CodeBERT and CodeT5

We do want to point out that we have tried numerous different parameters and are displaying the best results obtained. We had to downgrade our CodeBERT model because of the limited GPU resources available with us (using Colab Pro and ARC). The following draws a comparison between the suggested parameters and the ones we used for training our CodeBERT model :

- Train steps : 100000 → 100
- Evaluation steps: 5000 → 5
- Max source length : 512 → 256
- Max target length : 512 → 256
- Beam Size : 5 → 3
- Batch Size : 16 → 4
- Learning rate: 0.00005 → 0.005

This is the prime reason we feel that CodeBERT hasn't performed to it's best. Table 3 shows the different parameters that might have caused the performance difference between the two models.

Parameter	CodeT5	CodeBERT
Model Type	codet5-base	codebert-base
Learning Rate	0.00005	0.005
Beam Size/ Patience	5	3
Batch Size	4	4
Source Length	320	256
Target Length	256	256
Epochs	3	100

Table 3. Parameter comparison between CodeBERT and CodeT5

When used for the code summarization task, CodeBERT needs an external decoder that does not benefit from the pre-training. Currently, most approaches just use standard NLP pre-training methods to source code, treating it as a sequence of tokens similar to natural language (NL). This essentially

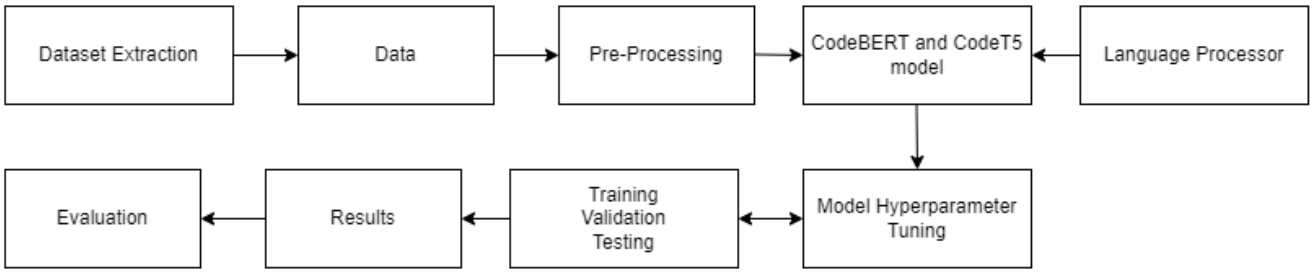

Figure 4. Workflow Diagram

overlooks the programming language's (PL) rich structural information, which is critical for fully comprehending code semantics. This is one of the significant reason why CodeT5 has outperformed CodeBERT.

We also have the following parameter values for CodeT5 when it translates code from Java to Python (multiplied by 100) :

- Ngram: For effective approximation matching, n-grams can be employed. A series of items can be embedded in a vector space by converting it to a collection of n-grams, allowing it to be matched to other sequences in an effective way.
  Score = 52.907
- Weighted Ngram : A more improved way on n-gram matching which reduces training data mismatch in cross-domain language model estimation
  Score = 53.352
- Syntax Match : The proportion of sub-trees retrieved from the candidate program's abstract syntax tree (AST) that match the sub-trees in the reference programs' AST is called Syntax Match (SM).
  Score = 59.763
- Dataflow Match: The ratio of the number of matched candidate data-flows to the total number of reference data-flows is known as Dataflow Match (DM).
  Score = 45.589

As described in the workflow diagram mentioned in the Figure 4, we have initially brainstormed on different topics in the software engineering domain. We came across the Transcoder, CodeBERT and CodeT5 models. CodeBERT and CodeT5 had performed code-to-code translation on Java to C#, both semantically similar languages. The Facebook's Transcoder model did conversion interchangeably among Java, C++ and Python. We wanted to evaluate how these models would perform on Java to Python (not so similar languages). We did the necessary language parsing, model tuning, tokenization and evaluation of these models and performed our analysis as we mentioned in the project proposal.

6 Broader Impacts/Discussion

The automation of many different software engineering tasks is paramount to help the productivity of software developers across the world. Automating these tasks will help elevate some of the menial tasks done by developers in the day-to-day world which can then help them concentrate on other intensive tasks. With the availability of these large scale tasks from open source repositories like GitHub, Leetcode, Geeks for Geeks etc. Helps create an opportunity to develop and train models to solve these menial tasks. These pre-trained models like CodeBert and CodeT5 are great baseline models on which research can be built upon and can be used to help solve these tasks with state of the art performance. Also with the reassurance of deep learning algorithms has helped create better machine learning models which understand the programming language in a better way both syntactically and semantically in nature.

To use these models specifically for code translation tasks are a good baseline models. As most of these models are pre-trained to help understand many different programming languages so they understand the syntactic and semantic nuances of the programming languages. This code translation task will help alleviate a lot of time and effort being put in by developers when they are performing the conversion of legacy systems into modern code. These pre-trained models can even be used for the automatic bug detection of the modern systems to help eliminate the manual steps that are involved in this process. Removing the manual parts of this process can help developers develop systems using the latest languages and tools that can be used with latest security updates and feature building capability. The source code translations can be used for other tasks other than conversion of legacy systems. These systems can be used to convert source code for a functionality developed by another team in a different language to the required programming language. Helping developers to achieve code sharing across various teams and create a code sharing environment. While reducing the cost and efforts for knowledge transfer, development and testing to help teams reuse code and functionalities across different teams.

Therefore the thinking that the use of deep learning tools in software engineering will not only help developers and companies save time and money as seen in the various tools implemented by the current day research. These tools will help create more robust, secure and better applications by developers while also eliminating the menial tasks they face during the development process. Helping them create better quality solutions with reduced cost. With these advantages of deep learning in software engineering see the software development world to continue moving forward with deep learning techniques to solve the problems faced by software developers in the world today.

We also considered the feedback from the proposal reviews. We incorporated the negatives pointed out in the feedback (like typos, data set description) and made sure that we have exhaustively covered the points to our knowledge.

7 Conclusion

In this paper, we presented the data, work and analysis of the application of CodeBERT and CodeT5 models on code-to-code translation from Java to Python programming language. We also compared the results to Facebook's Transcoder baseline and see how our models fared against them particularly using the BLEU evaluation metric. We compared the intrinsic pre-training tasks, internal architecture, underlying techniques of these models and made an assessment of their performance on code-to-code translation. We looked into a myriad of factors from the ethical concerns of using CodeT5 to the fine-tuning of these models to produce the desired results. We discussed the broader impact of our project in the domain of deep learning research on software engineering tasks. In our future work, we would like to tune the hyperparamters and run on a powerful GPU and then evaluate it's performance against state-of-art models like PLBART, etc.

8 Work Flow Distribution

Arnav Chhabra

1. Completed half of the literature survey
2. Implemented CodeT5 for Java to Python code translation end-to-end.
3. Wrote the introduction and abstract for the report.
4. Wrote the detailed description regarding the CodeT5 model in the report.
5. Contributed to the evaluation and results regarding the CodeT5 model.
6. Wrote the Broader Impacts/Discussion section of the report.

Harsha Vardhan Reddy

1. Completed half of the literature survey
2. Implemented CodeBERT for Java to Python code translation end-to-end.
3. Wrote the dataset section of the report.
4. Wrote the detailed description regarding the CodeBERT model in the report.
5. Contributed to evaluation and results regarding the CodeBERT model.
6. Wrote the conclusion section of the report.

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. AVATAR: A Parallel Corpus for Java-Python Program Translation. https://doi.org/10.48550/ARXIV.2108.11590

[3] Jason Evans. 2011. The HipHop Virtual Machine. https://engineering.fb.com/2011/12/09/open-source/the-hiphop-virtual-machine/.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. https://doi.org/10.48550/ARXIV.2002.08155

[5] The Haxe foundation. 2006. Haxe. https://haxe.org/.

[6] Qiyuan Gong. 2018. leetcode. https://github.com/qiyuangong/leetcode.

[7] Abel Avram Google. 2010. J2ObjC. https://www.infoq.com/news/2016/02/j2objc/.

[8] Ludovic Denoyer Guillaume Lample, Alexis Conneau and Marc'Aurelio Ranzato. 2018. Unsupervised machine translation using monolingual corpora only. https://arxiv.org/pdf/1711.00043.pdf.

[9] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2017. Unsupervised Machine Translation Using Monolingual Corpora Only. https://doi.org/10.48550/ARXIV.1711.00043

[10] Lowik Chanussot Guillaume Lample Marie-Anne Lachaux, Baptiste Roziere. 2020. Unsupervised Translation of Programming Languages. https://doi.org/10.48550/arXiv.2006.03511

[11] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1026–1037. https://doi.org/10.1109/ASE.2019.00099

[12] Nayuki. 2011. Project-Euler-solutions. https://github.com/nayuki/Project-Euler-solutions.

[13] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. 311–318.

[14] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[15] Bjarne Stroustrup. 1995. A History of C++: 1979 1991. https://www.stroustrup.com/hopl2.pdf.

[16] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. https://doi.org/10.48550/ARXIV.2109.00859