# Code Documentation to C# using a Multi Model Pipeline

Muhammad Ali Qadri*
Cristian Carlos Diaz Claure*
muhammadaliq@vt.edu
cristd4@vt.edu
Virginia Polytechnic Institute and State university
Blacksburg, Virginia, USA

## Abstract

Benchmark datasets have led to some innovative ideas in the field of Deep learning. There are many deep learning models that surpass what was previously thought impossible. From natural language modeling to understanding code semantics and generating code from text.

In this paper, we seek to find how multiple models that are specialized for different tasks, can work in unison and achieve something that was previously not done by a single model. In our case, we intend on using CodeT5's text-to-code functionality to convert a given document string of some piece of code into Java code, and then use the resulting Java as input to CodeT5's code-to-code model to finally convert it to C#.

In this two-step process, we fine-tune these previously trained models to match our needs and then intend on evaluating the results based on common state-of-the-art metrics such as CodeBLEU, and BLEU. We also verify if the resulting code is compile-able, as a measure to see how well our model understands the C# syntax.

***Keywords:*** datasets, neural networks, transformers, code-to-code, CodeT5

## 1 Introduction

Machine learning and AI have seen developments that have led to the possibility of self driving cars, world champions being defeated at games like go and chess, making fake images that are indiscernible from real, and many other ground breaking advances. While all applications , this paper seeks to focus on the software engineering applications, more specifically generating code documents into live code.

Even though within each of the experiments, there were a number of different unique tasks it offers. The possibilities from combining two seem to warrant some attention as well.

We plan on trying to pipeline a process that'll convert Docstrings to C# code.

One possible permutation we noticed, that isn't currently supported from just a single model, is taking a given docstring and converting it to its equivalent C# code. Which is our goal.

## 2 Related Work

In general, natural language to programming language generation or NL-PL tasks are a hot topic these days, with new and improved state-of-the-art models breaking previous records almost every week. Due to the higher level of knowledge required for annotation than for most other NLP tasks, training datasets for semantic parsing are often tiny. As a result, models for this application frequently require the incorporation of additional prior knowledge into the architecture or algorithm. In practice, the growing reliance on human expertise obstructs automation and raises development and maintenance expenses. We found a research that looks into whether a generic transformer-based sequence-to-sequence or seq2seq model with minimum code-generation-specific inductive bias design may achieve competitive performance [1].

The paper got an astonishing 81.03 percent exact match (EM) accuracy on Django and 32.57 BLEU score on CoNaLa by utilizing a relatively large monolingual corpus of the target programming language, which is inexpensive to mine from the web.

Another interesting research was found was based on Generative Pre-trained Transformer 3 or GPT-3, which is OpenAI's third-generation language prediction model (and the successor to GPT-2) in the GPT-n series. The entire version of GPT-3 can store up to 175 billion machine learning parameters and it is part of a growing trend of pre-trained language representations in natural language processing (NLP) systems. Its child, the OpenAI Codex is even used in Gihub Copilot and is compatible with many programming languages [2].

Another interesting and very promising article and research we sought was for the Salesforce CodeT5. Based on the T5 architecture, CodeT5 is a Transformer-based approach for code interpretation and generation. It makes use of an identifier-aware pre-training goal that takes into account the key token type information (identifiers) from code. To

*Both authors contributed equally to this research.

*Conference'17, July 2017, Washington, DC, USA*
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

further use the token type information from programming languages, which are the identifiers supplied by developers, T5's denoising Seq2Seq objective is expanded with two identifier tagging and prediction tasks. A bimodal dual learning objective is utilized for a bidirectional conversion between natural language and programming language to increase natural language-programming language alignment [3].

We can clearly see a pattern here, using transformer models and pre-trained models that make use of PL insights and syntax's, is something that can build downstream tasks and specializations for previously unsought tasks. Here we discuss the more specific dataset related works that are previously done by industry-leading researchers.

mapping language to code was discussed by Srinivasan Iyer and others from the University of Washington. They published the large dataset that CodeXGlue has as part of its text-code benchmarks. However, they produced 100,000 examples known as CONCODE, based on public github repositories, for only Java related code. [4]

Most of these papers mention evaluation the performance of their models in terms of BLEU (BiLingual Evaluation Understudy) and CodeBLEU scores.[5, 6] The papers and datasets however fall short on the task that we want to accomplish - convert documentation strings to C#. We aim to use these techniques, datasets and analysis from above to guide us to solve our tasks.

## 3 Proposed Plan

Our high level plan to execute our tasks is to first use existing datasets and pre-trained state-of-the-art models to build an enhanced dataset that contains samples of a doc string, with its equivalent Java code and C# code. We use Java code of the CodeTrans dataset in CodeXGLUE [5], and feed it to through an already fine-tuned CodeT5 model [7], to get its unknown doc string. As discussed above in related work, CodeT5 is already been trained extensively on Java-C# and many other language models to efficiently handle this task. We can then stitch this new generated documentation and the existing Java-c# pair to form a set of document string, Java code and C# code. These set will then be used to, generate Java code from our doc string through another CodeT5 model, and fine tune it to achieve the maximum possible evaluation score on CodeBLEU and BLEU. We do this to make sure that the documentation generated from our dataset is closely matched with the original Java code, and hence clearly demonstrate the running example of what we want to achieve with this paper.

This new Java code output will then be evaluated against a portion of our original Java code dataset using codeBLEU [8] and BLEU. We then finally plug this new Java code to the final CodeT5 model to generate the required C# code, while fine tuning it with the original C# pairs. Since the original Java and the generated Java from the documentation string are theoretically alike, it can be hypothesised that this Java will correspond with the original C# as is. Hence, fine-tuning is done on generated Java code, and loss is measured against the original C#. This lengthy processes of fine-tuning and evaluation is done to ensure that our end result can be considered state-of-the-art and we can show a proof of concept for making multiple models run in unison.

Apart from running the models on our datasets, we also try to evaluate our pipelines efficiency by varying the documentation strings, and seeing if that has any effect on the outcome of our pipeline. We also vary our beam size for all the generation tasks on the transformers, to see if we can increase our efficiency or our performance.

To re-iterate, we used different pre-trained models of CodeT5 as our initial starting point. As a pre-processing step, we used CodeT5 tuned for Java to document generation to produce documentation string. Those documentation strings, along with the original Java, and C# formed our dataset which will be used for our main goal.

Our first model's task is to receive the documentation string, and original Java to train and improve Java production given the documentation strings. Our second model's task is to receive the generated Java, and translate it into C#. This is fine-tuned used the loss between the original C# and the produced C#.

Each of the new models are used to generate their respective outputs such as documentation string, Java and C#, and all were then used to calculate BLEU, and CodeBLEU scores with their original Java and C#.

## 4 Data/Model Description

We mainly make use of CodeXGLUE's Code2Code [5] datasets which contains pairs of equivalent Java and C# code functions and APIs. The dataset's basic aim was to port legacy software code from one programming language to another. The dataset is built from several open-source and public collections including Lucene, POI, JGit and Antlr. The dataset contains roughly 10000 training examples, with 500 validation examples and 1000 tests. This dataset is vital for ensuring our Java and C# matching, and for making sure our models get properly evaluated in each step.

To view each of our techniques and steps in detail, we have written our the following sections. The first section emphasises on the dataset we are using, with its technical specific details, and how we pre-process that dataset to make a newer, refined dataset.

The following listings, show sample Java, and C# code from our dataset.

```
public static byte[] copyOfRange(byte[]
original,
int start, int end)
{
    if (start > end)
    {
        throw new IllegalArgumentException();
    }
    int originalLength = original.length;

    if (start < 0 || start > originalLength)
    {
        throw new
        ArrayIndexOutOfBoundsException();
    }

    int resultLength = end - start;
    int copyLength = Math.min(resultLength,
    originalLength - start);
    byte[] result = new byte[resultLength];
    System.arraycopy(original, start,
    result, 0, copyLength);
    return result;
}
```

**Listing 1.** Sample Java

```
public static byte[] copyOfRange(
byte[] original,
int start, int end)
{
    if (start > end)
    {
        throw new System.ArgumentException();
    }

    int originalLength = original.Length;
    if (start < 0 || start > originalLength)
    {
        throw new
        System.IndexOutOfRangeException();
    }

    int resultLength = end - start;
    int copyLength = System.Math.Min(
    resultLength, originalLength - start);
    byte[] result = new byte[resultLength];
    System.Array.Copy(original, start,
    result, 0, copyLength);
    return result;
}
```

**Listing 2.** Sample C#

```
Creates a copy of the given byte array
starting from the given start position
and ending at the given end position.
```

**Listing 3.** Sample Documentation String

We then move onto the actual learning part with our two step model, and we dive into the deeper technical aspects of each of our models including the way it is used and what are the outputs.

For running our models and general environemnt, we used LINUX setup which as hosted on a cloud on Datacrunch.io. We made use of their high performance GPU node which consisted of a single A100 Nvidia RTX GPU which comprised of 80 GB GPU memory, with 120 GB RAM and 22 vCPUs. This on-demand setup allowed us to acheive our task with higher update rates and allowed us to efficiently try out different experiments without waiting around.

Our main code repository consists of all the saved models, datasets and shell scripts along with all our notebooks and scripts that made it possible to achieve our task.

### 4.1 Dataset and Preprocessing

Our dataset consists of code from CodeXGLEU's Code2Code [5] dataset that is mainly comprised of CodeTrans data. This dataset consists of individual files that are divided into Java and C# respectively with each having training, validation and testing. The training sample consists of 10296 records, with our validation having 500 samples and our testing data having 1000 samples.

After these basic data statistics, we downloaded the data from our the CodeXGLEU repository and passed it through the tokenizer from the huggingface library. We made use of the RobertaTokenizer that was specialized and tokenizes the data based on the pre-trained Code-T5 model. To get a much better sense of this, we will explain what the tokenization step does and how it effects our results.

The tokenization step basically uses the roBERTa tokenizer [9]. The RoBERTa tokenizer uses byte-level Byte-Pair-Encoding and is derived from the GPT-2 tokenizer. Because this tokenizer has been trained to consider spaces as token parts (similar to sentence-piece), a word will be encoded differently depending on whether it is at the start of the sentence (without space) or not. This is specially important for our code translation tasks where we need to tokenize each and every character including spaces and all brackets including angle and curly and all symbols. This is necessary to make the model learn these different tokens and then use it in generating the sequence output for any NL-PL downstream task.

We employed use of the CodeT5-base tokenizer from huggingface that was specialized in tokenizing the documentation strings and the code. As our initial analysis on the dataset, we ran the code on the dataset and found out the maximum tokens that could be generated for our dataset.

We used the training set only for this part as only that will be visible to our model when training, and the rest will be unknown. After analysis we used the token size of 256 for each of our data during preparation.

Since our models are cannot understand characters, they need to be encoded via the same tokenizer. The encoding process translates each token and special tokens into a long integer, that represents that token in the vocabulary of the tokenizer. Once this is encoded into a long integer, it would be fed into the model for training and generation task, and then decoded once the output of anything needs to be shown.

After these initial steps for the pre-processing, it was time to make our new doc-string, Java and C# pair dataset. For this task we used the Salesforces Code-T5 model [3]. Before diving into what CodeT5 is, we need to know what transformers are and how they work. A transformer model is a neural network that tracks relationships in sequential data, such as the words in this sentence, to learn context and thus meaning. Transformer models use a developing collection of mathematical approaches known as attention or self-attention to detect subtle ways that even far-flung data pieces in a series impact and depend on one another. This model was first discussed in the Attention is all you need paper from Google in 2017 [10]. Taking on from this breakthrough in the ice age of deep learning, T5 was introduced, which is based on transfer learning.

Transfer learning has developed as a powerful technique in natural language processing, where a model is initially pre-trained on a data-rich task before being fine-tuned on a downstream task (NLP). T5 is an encoder-decoder model that has been pre-trained on a multi-task mixture of unsupervised and supervised workloads, with each task transformed to text-to-text [11]. T5 can handle a wide range of jobs right out of the box by pre-pending a different prefix to the input corresponding to each activity, such as translation or summary.

Now finally CodeT5, which a pre-trained encoder-decoder Transformer model that takes advantage of the code semantics communicated by developer-assigned identifiers. Our methodology uses a unified architecture to enable both code comprehension and generation tasks at the same time, allowing for multi-task learning [3].

Hence, for converting the Java files to a document string, we dont need to make our own model, we made use of CodeT5‘s already fine-tuned model that does it for us. This will be the starting place for our next modeling. We generated the document string for each of our training, validation and test set and stitched them together to make our final dataset.

### 4.2 Learning Models

For our problem, instead of making use of our own architecture, with different blocks and layers of transformers and LSTMs, we use already existing and previously trained models. These models can be integrated with our Python code using the huggingface library, which is a vast open-source library with models and usage documentation. We can then fine-tune these models with the dataset that we have created.

We make use of CodeT5‘s fine tuned models for converting the documentation string to Java and then using that generated Java to convert it finally to C#.

### 4.3 Documentation String to Java

In essence, we used the pre-trained code generation model which was already fine-tuned on the large Concode [4] dataset. It has also been trained on the CodeSearchNet [12] which is a large corpus of code and doc string data. This model is trained from the base CodeT5 model on the documentation to Java dataset and hence, will provide a starting ground to fine-tune it on our specific dataset.

We run the training on our GPU instance with Pytorch library for 20 epochs, with a learning rate of 0.00005, and we plotted the training and validation loss per batch update. We used the ADAM optimizer to update our gradients and can see our long flat tail on the plot, which is described in the evaluations and results section. After running our model, we capture the generated Java for each of the documentation on the training, validation and testing set using the huggingface library functions, with beam search of 5 branching factor. Beam search is a decision-making method used in many NLP and speech recognition models to select the best output given target variables such as maximum probability or the next output character. Beam search, which was first employed for voice recognition in 1976, is frequently utilized in models that include encoders and decoders with LSTM or Gated Recurrent Unit modules [13].

This prepared Java is now used by the next and final model in our task, Java to C#.

### 4.4 Java to C#

Like before, we used a pre-trained code translation model which was already fine-tuned on the large CodeTrans [5] dataset. We use this model to fine-tune our downstream task of converting Java to C#.

Before actual running of the model, we shuffled the validation and training to get a better result for our model and show a good decrease in training and validation loss. Although the loss was not as great as for the other model, it will still be considered a good enough feat for this amount of a task.

We run the training on our GPU instance with Pytorch library for 5 epochs, with a learning rate of 0.00005, and we plotted the training and validation loss per batch update. We used the ADAM optimizer to update our gradients and can see our somewhat long flat tail on the plot, which is described in the evaluations and results section. After running our model, we capture the generated C# for each of the

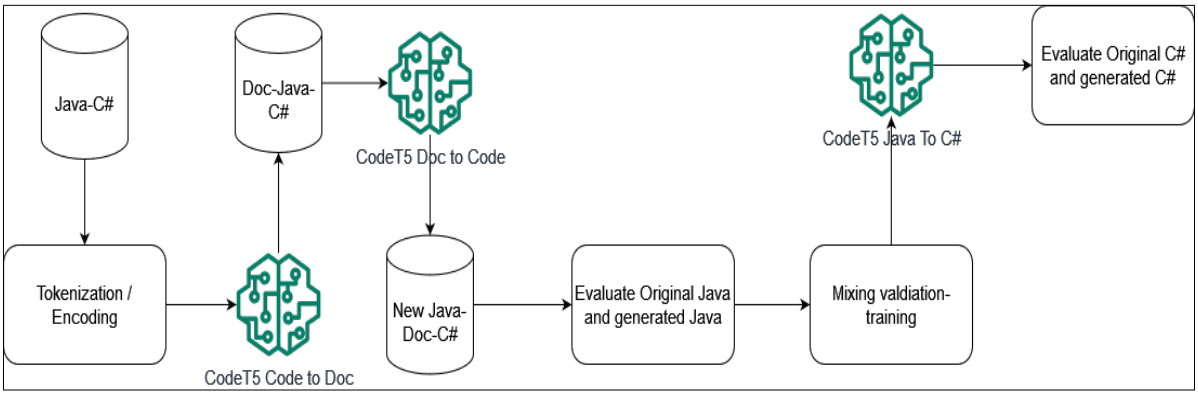



**Figure 1.** Model Pipeline

Java code on the training, validation and testing set using the huggingface library functions, with beam search of 5 branching factor [13].

This concludes our task of converting the documentation string to C# via porting it through Java code. While taking these models from end-to-end, we have our pipeline here, that nicely wraps up our entire operation in a diagram.

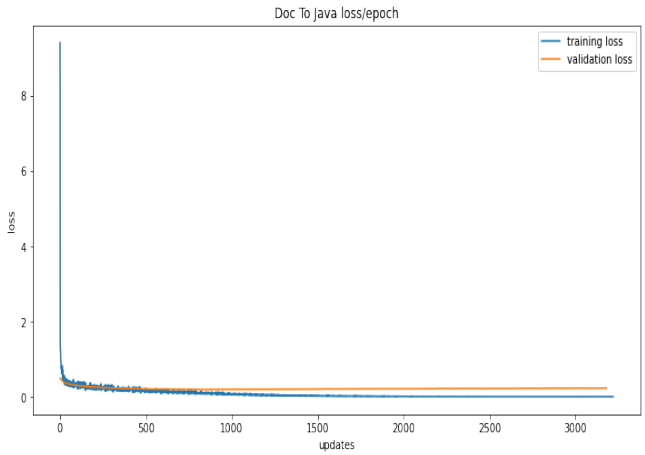
## 5 Evaluation & Results

Evaluation was performed on each step of the task with evaluation being done on the generated Java and the original Java, and generated C# and the original C#. The evaluation metrics we used were BLEU and CodeBLUE [6, 14]. CodeBLEU is an enhancement made on the BLEU score and considered vital for code PL in terms of looking at code. It considers ngram matches, syntax matches, and data flow matches to compare the reference code with the hypothesis code.

| | Train | Valid | Testing |
|---|---|---|---|
| Java BLEU | 86.2 | 31.45 | 29.9 |
| Java CodeBLEU | 89.32 | 42.29 | **42.06** |
| C# BLEU | 2.98 | 1.75 | 2.85 |
| C# CodeBLEU | 11.41 | 9.73 | **11.21** |

**Table 1.** Model Evaluation Results

**Figure 2.** Loss for Documentation to java




As seen in our Figures. The training, and validation loss displayed the desired trend for our documentation string to Java code model. That is to say that over updates, we decreased loss, and achieved a long flat tail with 3000 updates.

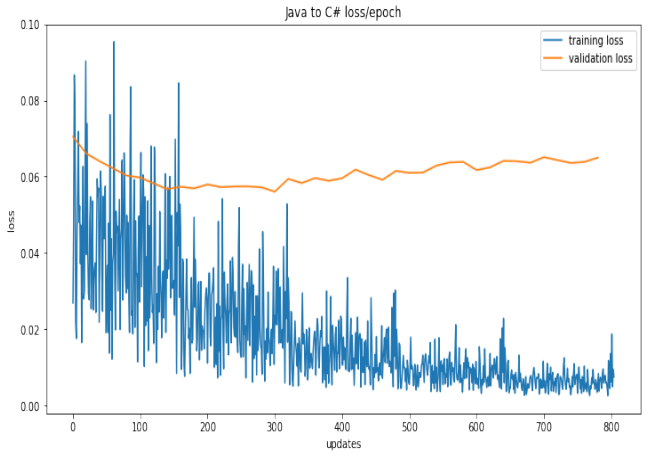
In our fig.2, training showed significant improvement, however validation loss did not improve by much, but showed a desirable trend.

After 3000 updates, we computed BLEU and CodeBLEU results as seen in table1. We achieved slightly better results on our dataset, but that's probably because ours was smaller (going from 20 to 29). With a CodeBLEU score of 41, which we thought wasn't bad, and pretty reading at least from a human perspective.

Below you'll find our figure for training and tuning on the second portion (the Java to C# portion). As you can see, this one didn't produce the desirable trend we had thought, but if we're being honest this is more or less what we were expecting. While the loss did decrease, it wasn't improving as much as the first model did over iterations.

This time, after 800 updates we generated our C sharp files, and computed CodeBLEU and BLEU. These results weren't great at all, we achieved only a BLEU score of only 4.87, and a CodeBLEU result of 11.21.

**Figure 3.** Loss for Java to C#




```
public static byte[] copyOfRange(
byte[] original, int start, int end)
{
    if (start > end)
    {
        throw new
        IllegalArgumentException();
    }

    int originalLength = original.length;
    if (start < 0 || start > originalLength)
    {
        throw new
        ArrayIndexOutOfBoundsException();
    }

    int resultLength = end - start;
    int copyLength = Math.min(
    resultLength, originalLength - start);
    byte[] result = new byte[resultLength];
    System.arraycopy(original, start,
    result, 0, copyLength);
    return result;
}
```

**Listing 4.** Sample Generated Java

```
public byte[] Create(byte[] array,
byte arrayStart, byte arrayEnd)
{
    return new byte[arrayStart
    + arrayEnd.Length];
}
```

**Listing 5.** Sample Generated C#

Based on the sample generated Java, and Sample Generated C#. It is clear that our results even to a human, aligns with the codeBLEU and BLEU results. Since our results for Java were acceptable, but our results for C# weren't as consistent.

## 6 Lessons Learned

There were many things that we learned during the course of this paper. We learned different state-of-the-art models ranging from GPT, codeBERT and CodeT5. We learned about the huggingface library and how to make use of these massive models on our downstream tasks. We also became well-versed in the fact that these models require high computational power, and learned how to deploy these and train them on on-demand cloud services. Learning how to utilize high end GPU power was a big relief for us as it improved our throughput significantly. With shorter load times and training times, we could run models in parallel and experiment more openly with hyper-parameters, as compared to with our previous setup, where we had to wait hours for just a few minor updates.

Although there is a plethora of models out there, and eventually finding which one to use for which task is a challenge in its own way. We learned how to play into the models strenghts and weaknesses and get our job done. For example, we initially went with the CodeGPT document to code model, which we tried endlessly with different decoding models but failed to get any good results. All our code results were meaningless and were'nt worth going over. We eventually experimented and made a huge improvement while utilizing the CodeT5 model.

In short while taking this massive project from end-to-end we faced many challenges that we took head on and solved one by one, and learned how an actual real-world machine learning project is taken care of.

## 7 Broader Impacts/Discussion

Our results achieved in hindsight are expected, we never anticipated the initial Java being produced from our first model to be so accurate and have a remarkable CodeBLEU score. Having ran CodeBLEU and BLEU against the initial training data, and our generated Java files, we achieved a CodeBLEU score of 74. This meant however, that the pretrained model from Java to CSharp wasn't learning much, it was correctly generating the CSharp functions.

To counter this, we created a new training and validation set, by randomly mixing and shuffling from both. Since there were around 11000 combined samples, training took 10000 and the validation set would be comprised of the rest, and we did manage get better results this way. We initially only got 10 codeBLEU but with random sampling we improved slightly, to 11. Over the same number of update steps, on a brand new model of course.

Documentation string to C# is probably feasible, but we would need to scrape the data ourselves from sources. It

would probably be easier to scrape Github pages, and get the comments to generate a dataset that way. That's what was done for the CONCODE dataset, and given the amount of public repositories, it wouldn't be a monumental task to aggregate such a dataset. It's important to note, the conversion to an intermediary language, we can't think of when that would be necessary.

In the future, developers could use plugins, and at their discretion run the tool to generate basic class files. During development there is boiler plate, for certain tasks, and this could facilitate certain design patterns. As an example, this could be used for factory pattern classes. At a business level, these types of relations are common, and if developers could be spared having to generate these files. Developers could just go straight into making use of those files, and creating their product. Anytime saved, is useful during sprints since, it'll yield greater revenues.

In a much broader perspective, we showed a proof of concept that we can make use of multiple massive models and make them work together to reach a bigger and complex goal. This paper can be used as a reference to start off a chain reaction that leads to more experimentation on usage of a combination of state-of-the-art models.

## 8 Conclusion

In conclusion, the attempt to pipeline multiple models in an attempt to yield a new functionality is doable. The results were unsatisfactory however, and our biggest problem was our training data for the Java to Csharp portion. We needed to manually generate our own dataset, and maybe then we would have achieved desirable results. We also think that it would be better to generate C# directly, instead of losing data by going to an intermediary language, and in the future the task documentation string to code, could be used to develop basic parts of a bigger product.

## 9 Work Distribution

| **Muhammad Ali Qadri** | **Cristian Diaz** |
|---|---|
| Preprocessing | Java to C |
| Document to Java | Evaluation |

**Table 2.** Work Distribution

## References


[1] S. Norouzi, K. Tang, and Y. Cao, "Code generation from natural language with less prior knowledge and more monolingual data," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, (Online), pp. 776–785, Association for Computational Linguistics, Aug. 2021.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[3] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, (Online and Punta Cana, Dominican Republic), pp. 8696–8708, Association for Computational Linguistics, Nov. 2021.

[4] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 1643–1652, Association for Computational Linguistics, Oct.-Nov. 2018.

[5] M. LLC, "CodeXGLUE code to code translation," 2022.

[6] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu, "Bleu: a method for automatic evaluation of machine translation," 10 2002.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, (Online), pp. 1536–1547, Association for Computational Linguistics, Nov. 2020.

[8] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *ArXiv*, vol. abs/2009.10297, 2020.

[9] "Roberta tokenizer."

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[11] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *CoRR*, vol. abs/1910.10683, 2019.

[12] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *ArXiv*, vol. abs/1909.09436, 2019.

[13] "What is beam search? explaining the beam search algorithm."

[14] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020.