CodeT5++: A Pre-trained Programming Language Model for Code Summarization Task*

Mahesh Arumugam
Section 1 / W266
aumahesh@berkeley.edu

Anshuman Awasthi
Section 5 / W266
awasthianshuman@berkeley.edu

Abstract

There has been considerable research in building pre-trained models for programming language tasks, such as CodeBERT and CodeT5, that enable several downstream tasks, including code summarization, generation, and translation. In this paper, we focus on the task of automated code summarization that translates Python source code into a natural language *docstring*. Towards this end, we propose CodeT5++, extensions to CodeT5 where we introduce novel pre-training tasks that capture relevant source code features most useful in code summarization tasks. Specifically, we pre-train the model to (1) predict masked return values of Python functions, (2) detect whether a docstring and source code pair is an accurate representation of the function, and (3) predict masked function names of Python functions. Subsequently, we fine-tune the models for the code summarization task and evaluate the performance using a smoothed BLEU-4 score, a precision-based metric applicable in translation tasks. Finally, we analyze how the pre-training steps help improve the summarization tasks.

1 Introduction

Docstrings in programming language source code provide a way to document a function, a class, or a module. It enhances the readability of the source code and provides a context before the actual implementation for better understanding. However, more often than not, programmers write such docstrings as an afterthought -if at all and ignore their correctness. Though there are general guidelines to writing good docstrings, most docstrings do not adhere to these guidelines strictly. As a result, source code often tends to be undocumented or poorly documented. Rather than the authors providing clear documentation of the source code, it is often left to other programmers or maintainers of the source code to figure out the code's assumptions, intentions, and functionality.

To eliminate programmer fatigue and improve documentation, automatically generating docstrings from source code is essential. Towards this end, first, we observe that source code provides sufficient context through logical constructs (such as conditions and iterations), meaningful identifier names, input arguments, and return values. As a result, machine learning algorithms can leverage such information and enable automatic programming language (PL) to natural language (NL) translation.

Contextual embeddings of tokens (or words) exist for natural language texts that facilitate downstream tasks such as summarization, translation, and question-answering. Such contextual embeddings of tokens (e.g., Devlin et al. (2019) and Brown et al. (2020)) are pre-trained using a large corpus of unlabeled data (e.g., Wikipedia). The success of such contextual embeddings and transformer-based neural network architectures (Vaswani et al., 2017) for sequence-to-sequence learning motivated the extension of such architectures for programming languages. One such model is CodeT5 (Wang et al., 2021), which is based on T5's (Raffel et al., 2020) encoder-decoder transformer architecture. CodeT5 enables various programming language tasks, including code summarization, translation, completion, and unit-test generation.

In this paper, we propose CodeT5++, extensions to CodeT5's pre-training tasks that learns to generate better natural language docstrings (summarizations). Our work builds on top of CodeT5 by adding new pre-training objectives: (1) masked return values prediction, (2) corrupted docstring detection, and (3) masked function names prediction.

* Original proposal title: *AutoDoc: Human Readable Docstrings from Source Code*.

In Masked Return Values (MRV) prediction, the pre-training task masks return values (constants, literals, identifiers, or expressions) and predict these values. In Corrupted Docstring (CDS) detection, the pre-training task randomly corrupts docstrings and then predicts which docstring and source code pairs accurately represent the function. Finally, in Masked Function Names (MFN) prediction, the pre-training task masks the names of the functions and predicts them. These pre-training tasks aim to capture the salient aspects of the source code that are most meaningful in a code summarization task.

In summary, the contributions of this paper are:

- We fine-tune CodeT5 on the Code Docstring Corpus (Barone and Sennrich, 2017) dataset and compute a smoothed BLEU-4 (Lin and Och, 2004) score, which is the evaluation metric adopted by CodeT5 and other pre-trained programming language models for translation tasks. A precision-based metric such as BLEU (Papineni et al., 2002) is suitable for translation/summarization tasks, where we would like our models to generate adequate and fluent summaries.

- We propose CodeT5++, a pre-trained programming language transformer-based model built on top of CodeT5. CodeT5++ pre-trains on new objectives: MRV, CDS, and MFN. We pre-train for these tasks on Code Search Net's (Husain et al., 2019) summarization dataset for Python functions.

- We fine-tune and evaluate CodeT5++ on the Code Docstring Corpus dataset.

2 Background

Pre-trained programming language models. Research on contextual embeddings for natural languages has achieved great success (e.g., BERT (Devlin et al., 2019) and GPT (Brown et al., 2020)). Recent work attempts to extend the architectures of pre-trained language models to programming languages. For example, CuBERT (Kannade et al., 2020), CodeBERT (Feng et al., 2020), and GraphCodeBERT (Guo et al., 2021) build on the BERT's (Devlin et al., 2019) pre-trained transformer-based architecture. CuBERT employs the same pre-training objectives as BERT. In CuBERT, a sentence is the shortest sequence of consecutive lines that constitutes a valid statement. The pre-training dataset consists of a massive corpus of Python programs collected from GitHub[1]. Similar to CuBERT, CodeBERT uses a transformer-based architecture that is pre-trained using masked language modeling and replaced token detection. The pre-training data is the Code Search Net (Husain et al., 2019) dataset. Extending CodeBERT, GraphCodeBERT uses data flow information and learns the code structure that enables downstream tasks such as search, clone detection, translation, and refinement.

In addition, recent work also extends GPT for programming language tasks, especially for code completion (e.g., Svyatkovskiy et al. (2020)).

Encoder-decoder architectures. BART (Lewis et al., 2020) generalizes BERT, GPT, and other pre-training approaches by combining a bidirectional encoder and an auto-regressive decoder. BART is pre-trained to retrieve the original document from a corrupted input. BART is effective for text generation and comprehension. Hence, models such as PLBART (Ahmad et al., 2021) work on extending BART (Lewis et al., 2020) for downstream tasks such as code summarization and programming language conversion.

T5 (Raffel et al., 2020) introduces the idea of a unified framework that converts all text-based natural language problems into a text-to-text format using a text-to-text transformer model. Inspired by T5, approaches such as PyMT5 (Clement et al., 2020), Mastropaolo et al. (2021), CodeTrans (Elnaggar et al., 2021), and CodeT5 (Wang et al., 2021) extend T5 for programming language tasks. Unlike CodeT5, the other approaches do not utilize features unique to the source code (e.g., identifiers).

CodeT5 is identifier aware and trained to recover the correct identifiers during pre-training. The pre-training objectives of CodeT5 include: (1) masked span detection, where random spans of arbitrary lengths are masked and predicted during training, (2) masked identifier prediction, where all identifiers are masked and predicted during training and (3) bimodal dual generation that translates docstrings to code snippets and vice versa. It is pre-trained on Code Search Net (Husain et al., 2019) dataset along with two additional datasets of C/C-Sharp from BigQuery[1] for code summarization, generation, refinement, translation, and defect detection tasks.

[1] https://console.cloud.google.com/marketplace/details/github/github-repos

3 Datasets

The two datasets we primarily use in pre-training and fine-tuning are the Code Search Net (Husain et al., 2019) and the Code Docstring Corpus (Barone and Sennrich, 2017).

Pre-training. We use the summarization set of Python functions from the Code Search Net to pre-train CodeT5++. This dataset includes bimodal and unimodal data, with approximately 250K examples for pre-training and 14K for validation. Note that we sample 5000 examples from the validation set for validation.

Fine-tuning. To fine-tune CodeT5++ for the summarization task, we employ the training set of the Code Docstring Corpus dataset[2]. The dataset consists of 109108 triples of Python function declarations, docstrings, and function bodies. Also, the validation set includes 2000 examples.

4 Methods

The design and implementation of CodeT5++ are relatively straightforward. CodeT5 is already pre-trained on the Code Search Net (Husain et al., 2019) dataset. In CodeT5++, we introduce novel pre-training tasks as illustrated in Figure 1.

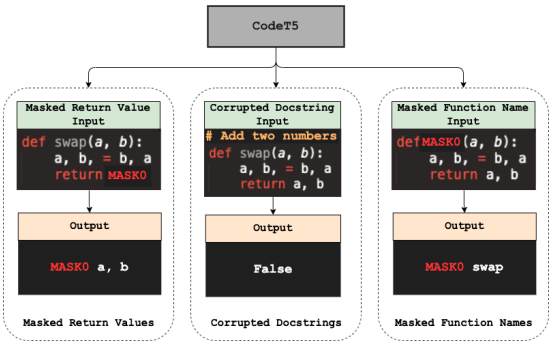

Figure 1: Pre-training tasks of CodeT5++.

4.1 Masked Return Values (MRV)

In trying to understand what a Python function is doing, it is essential to know what that function is returning as it helps identify the outcome of the function. As a result, it helps in generating a better docstring. To achieve this objective, we pre-train CodeT5++ on the masked return value prediction task. In this phase, we mask the return values (constants, literals, identifiers, expressions) and let the training algorithm learn and predict what these return values should be for a given Python function.

4.2 Corrupted Docstring (CDS)

To better understand a valid docstring, we implement another pre-training phase, where we pre-train CodeT5 on a dataset containing randomly corrupted docstrings for half of the dataset. During training, the model learns and predicts if a Python function and the corresponding docstring match or not. This task enables the model to generate reasonable natural language texts.

4.3 Masked Function Names (MFN)

First, we observe that function names provide valuable input in understanding and summarizing a Python function. Therefore, in this pre-training objective, we try to leverage that information. Specifically, we mask the function's name and train the model to predict it. We note that this training objective is complementary to the masked identifier prediction (MIP) objective in CodeT5 (Wang et al., 2021), where all identifiers are masked (not just the function's name). In MFN, we keep the function body intact and let the model determine the correct function name given its entire context. The model learns to identify the essential attribute of a function that is most helpful in summarization tasks. On the other hand, in MIP, the model tries to learn all at once. While MIP is more applicable in downstream tasks, such as code translation and code completion, we believe MFN is more suitable for code summarization tasks and natural language to programming language translation.

4.4 Fine-Tuning and Evaluation

We fine-tune the model for code summarization using the training set of the Code Docstring Corpus (Barone and Sennrich, 2017). We evaluate the performance of our design using the smoothed BLEU-4 score (Lin and Och, 2004) on the Code Docstring Corpus test set. BLEU (Papineni et al., 2002) is based on a modified n-gram precision (p_n). This metric captures *adequacy* and *fluency* aspects of machine translation. In addition, similar to CodeT5, we employ a smoothing technique (proposed in Lin and Och (2004)) that adds one to n-gram hits and total n-gram count for $n > 1$. As a result, candidate docstrings with less than n words still get a positive smoothed BLEU score, as long as there is a match in lower n-grams.

[2] https://github.com/EdinburghNLP/code-docstring-corpus/tree/master/parallel-corpus

5 Results

Table 1 summarizes our results on the Code Docstring Corpus dataset.

Methods	Smoothed BLEU-4	
	Validation	Test
Baseline		
CodeT5	N/A	**9.19**
Fine-tuned CodeBERT	9.84	7.35
Fine-tuned CodeT5	**16.24**	7.44
CodeT5++		
+MRV	**16.49**	**17.34**
+CDS	15.42	16.45
+MRV,CDS	15.88	16.69
+MFN	15.96	16.64

Table 1: Smoothed BLUE-4 scores (Lin and Och, 2004) for models on Code Docstring Corpus (Barone and Sennrich, 2017). Note that CodeT5 baseline model is not fine-tuned and, hence, the smoothed BLEU-4 on validation set is not applicable.

5.1 Baseline Results

We perform three baseline evaluations. First, we evaluate CodeT5 for the summarization task. We consider this result our absolute baseline as the model is not fine-tuned. We get a smoothed BLEU-4 score of 9.19.

In addition, we evaluate two baseline models after fine-tuning them on the Code Docstring Corpus training set. The first baseline model with fine-tuning is CodeBERT, and the smoothed BLEU-4 score for CodeBERT is 7.35. The second baseline model with fine-tuning is CodeT5, and it performs marginally better with a smoothed BLEU-4 score of 7.44. However, this result is surprisingly low compared to our absolute baseline model. Since CodeT5 is a pre-trained model, we believe fine-tuning is *overfitting* the model to the Code Docstring Corpus training dataset. From Table 1, we observe that the fine-tuning CodeT5 performed better on the validation set than the test set.

5.2 CodeT5++ Results

CodeT5+MRV. First, we consider CodeT5+MRV, where we extend CodeT5 by adding a pre-training objective of masking the return value in the Python function and training the model to predict it. Once the pre-training is complete, we fine-tune the model using the train set of the Code Docstring Corpus dataset. This model achieved a smoothed BLEU-4 score of 17.34 on the test set. As we anticipated, this pre-training task helped the model understand what the Python function does, and the docstring generation task improved performance.

CodeT5+CDS. Next, we consider CodeT5+CDS, where we add the Corrupted Docstring pre-training objective to CodeT5. After fine-tuning on Code Docstring Corpus dataset, this model achieved a smoothed BLEU-4 score of 16.45 on the test set. This result is considerably better than the BLEU score of our baseline models. As we had suspected, corrupting approximately 50% of the docstrings and training the model to understand what is a valid docstring for a Python function and what is not improves the model's ability to generate a docstring for a Python function. However, CodeT5+CDS did not perform better than CodeT5+MRV. We interpret this to mean that a model that understands the objective of Python functions generates better docstrings than a model that can understand what makes a valid docstring.

CodeT5+MRV,CDS. Since pre-training CodeT5 on MRV and CDS individually yielded much better performance results, we consider CodeT5+MRV,CDS, where we pre-train the CodeT5 model on both these objectives. CodeT5+MRV,CDS achieved a smoothed BLEU-4 score of 16.69 on the test set. Although this model performs better than CodeT5+CDS, it does not perform as well as CodeT5+MRV. This result validates our conjecture that the task of understanding the objective of Python functions has a higher weightage than the task of understanding what a valid docstring looks like for code summarization tasks.

CodeT5+MFN. As discussed in Section 4.3, we expect CodeT5+MFN to utilize the function body to generate good summaries without relying on the function's name. Like other models, we fine-tune CodeT5+MFN on the Code Docstring Corpus dataset. The model achieved a smoothed BLUE-4 score of 16.64 on the test set. While this is an improvement over the CodeT5+CDS model, contrary to our expectation, this did not perform better than CodeT5+MRV. MFN objective is unique in that it tries to capture the signal from the function body, whereas the MRV objective captures the signal from the function's return values. Therefore, we expect MFN to perform better in other test sets and, more importantly, when combined with other pre-training objectives (including MRV).

Combining pre-training objectives. Due to lack of time, we did not perform modeling with combinations of pre-training objectives (except for MRV and CDS). We shall take this up as future work.

6 Discussion

Consider a function that determines whether a number is prime, as shown in Figure 2. We test two functions with same function body: *isPrime* and *unique*. The second function is named deliberately to ensure that the function name does not provide enough information to construct a good docstring.

```
def {isPrime | unique}(n):
    if n <= 1:
        return False
    if n == 2:
        return True
    for i in range(2, math.ceil(math.sqrt(n))+1):
        if n % i == 0:
            return False
    return True
```

Figure 2: Determine if the number is prime or not. Two different function names: *isPrime* and *unique*.

All models generate a satisfactory docstring for *isPrime*. On the other hand, all models except CodeT5+MFN generate "*Return True if n is unique.*" CodeT5+MFN generates a more accurate summary: "*Return True if n is a valid prime.*" for *unique*. We attribute this to the pre-training objective used in CodeT5+MFN. MFN lets the model learn the name of the function from the structure of the function body. While CodeT5 is pre-trained using a masked identifier prediction (MIP), it still did not generate a good docstring for *unique*. During training, MIP masks all identifiers, including the function's name. As a result, the model does not have enough context to distill a function's name or description. Thus, in this example, the MFN objective is helpful.

On the other hand, CodeT5+MRV also fails to generate a valid docstring (though it has a better BLEU score). In this particular case, due to the overlap of common n-grams between the docstring generated by CodeT5+MRV and a reference docstring for *unique*, it would not be penalized much. Such scenarios possibly help CodeT5+MRV have a slightly better BLEU-4 score.

As another illustration, consider the swap function shown in Figure 3 with four variations in function names: *swap*, *xyz*, *blah* and *reorder*.

```
def {swap | xyz | blah | reorder}(a, b):
    a, b = b, a
    return a, b
```

Figure 3: Swap function. Function names are *swap*, *xyz*, *blah*, or *reorder*.

Table 2 shows the docstrings generated by our models. All models generate a reasonable docstring for *swap*. However, none of the models generate a good docstring for the other variations. For example, CodeT5+MRV generates "*Same as a, b, except that first element is returned.*" for *blah*. And, CodeT5+MRV,CDS generates "*Reorder the elements of a and b in ascending order*" for *reorder*. We believe that when there is insufficient context (or the source length is very small), the models resort to using features that could indicate what the function may be doing. The actual function body is tiny in the example presented in Figure 3 and, hence, the models may rely on the function's name to generate docstrings. Since most programmers would not name their functions arbitrarily, the models will usually generate reasonable docstrings.

7 Conclusion

We have presented CodeT5++ by introducing novel pre-training tasks that capture relevant source code features most valuable in the docstring generation. Specifically, we proposed masked return values (MRV) prediction, corrupted docstring (CDS) detection, and masked function names (MFN) prediction. Furthermore, we fine-tuned and evaluated our models on the Code Docstring Corpus dataset. Finally, we used smoothed BLEU-4 scores, a precision-based metric applicable in translation tasks, to compare our new models to the baselines. Our novel training tasks helped improve the BLEU-4 score. We also observed that a model pre-trained solely on MRV performs better than a model pre-trained solely on CDS (or MFN) or a combination of MRV and CDS. Based on this result, we believe that understanding the objective of Python functions is more important than understanding what a valid docstring looks like for code summarization tasks.

While adding new pre-training tasks and fine-tuning CodeT5++ for multiple programming languages is a natural extension to this work, we also expect additional follow-up work. For example, we would like to explore how CodeT5++ can help in other downstream tasks, such as source code translation across different languages and natural language to programming language code generation. We also want to investigate unit-test coverage and automatic unit-test case generation for source code, as this helps unburden programmers, from code documentation to unit testing.

Model	*swap*	*xyz*	*blah*	*reorder*
Baseline				
CodeT5	*Swap two values.*	*Returns the two points a and b.*	*Return the two elements a and b.*	*Reorder two lists.*
Fine-tuned CodeBERT	*Swap two arrays.*	*.. todo:: WRITEME*		*.. todo:: WRITEME*
Fine-tuned CodeT5	*Swaps two numbers.*	*Return the xyz components of a and b.*	*Same as a, b, but returns the result.*	*Same as a, b, but returns the result.*
CodeT5++				
+MRV	*Swaps two lists of numbers.*	*Returns the two elements of a, b, in the same order as a, b.*	*Same as a, b, except that the first element is returned.*	*Reorder the elements of two lists.*
+CDS	*Swaps two lists.*	*Returns the xyz of a and b.*	*Same as a, but with a different value.*	*Reorder two lists of numbers.*
+MRV,CDS	*Swaps two lists of numbers.*	*Return the two elements of a and b.*	*Same as a, b, except that the result is a, b.*	*Reorder the elements of a and b in ascending order.*
+MFN	*Swaps two lists of numbers.*	*Returns the two elements of the xyz tuple.*	*Same as a, b, but returns the result.*	*Reorder the two lists of values.*

Table 2: Docstrings generated for swap function. Note that CodeBERT did not generate any docstring for *blah*.

References

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*, pages 2655–2668, Online. Association for Computational Linguistics.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. *In Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065, Online. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicone's code through self-supervised deep learning and high performance computing. *CoRR*, abs/2104.02443.

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. *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. *In Proceedings of 9th International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Aditya Kannade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embeddings of Source Code. *In Proceedings of the 37th International Conference on Machine Learning*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto,

and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347.

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 of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

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):1–67.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. *IntelliCode Compose: Code Generation Using Transformer*, page 1433–1443. Association for Computing Machinery, New York, NY, USA.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

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. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

A Appendix: Datasets

Table 3 shows the datasets we used for pre-training and fine-tuning tasks.

Dataset	Train	Validation	Test
Code Search Net	251820	5000 (13914)	14918
Code Docstring Corpus	109108	2000	2000

Table 3: Code Search Net (Husain et al., 2019) and Code Docstring Corpus (Barone and Sennrich, 2017) used to pre-train and fine-tune our models. Note that we only use the summarize/Python dataset from Code Search Net for pre-training our models. And, we only use the parallel-corpus dataset[3] from Code Docstring Corpus for fine-tuning and evaluating our models.

Preprocessing of Code Search Net dataset. For CodeT5++ pre-training tasks, we construct datasets for various pre-training objectives. Masked return values dataset masks all return values (constants, literals, identifiers, expressions). The corrupted docstring dataset corrupts the docstrings of half of the training set. Finally, the masked function names dataset masks the name of the functions.

Preprocessing of the Code Docstring Corpus dataset. Since we are interested in generating docstrings for Python functions, we must input a concatenation of the function declarations and function bodies while fine-tuning the model. Furthermore, the data uses many unique tokens and escape characters to denote certain aspects of the data (such as newlines and whitespaces). We must process this data as required before inputting it to fine-tune the model.

[3] https://github.com/EdinburghNLP/code-docstring-corpus/tree/master/parallel-corpus

B Appendix: Training

CodeT5++ uses the same architecture as CodeT5, with 12 layers, 768-dimensional hidden states, and 12 attention heads in the decoder. We train CodeT5++ in Google Cloud Platform on a virtual machine with 2 NVIDIA Tesla T4 GPU instances. Table 4 shows the hyper-parameters used for pre-training and fine-tuning. Each epoch in pre-training takes around 200 minutes for training and around 17 minutes for validation. Likewise, each epoch in fine-tuning takes around 110 minutes for training and around a minute for validation. The calculation of the smoothed BLEU-4 score on the test set takes around 20 minutes.

Parameter	Pre-training	Fine-tuning
Epochs	3	3
Learning rate	0.00005	0.00005
Train batch size	16	16
Validation batch size	N/A	8
Max Length	512	N/A
Max source length	N/A	256
Max target length	N/A	128

Table 4: Hyper-parameters used in pre-training and fine-tuning.

C Appendix: isPrime and unique

Table 5 shows the docstrings generated by our models for *isPrime* and *unique* functions shown in Figure 2.

Model	*isPrime*	*unique*
Baseline		
CodeT5	*Check if n is prime.*	*Return True if n is unique.*
Fine-tuned CodeBERT	*Returns True if n is prime.*	*Return True if n is unique.*
Fine-tuned CodeT5	*Returns True if n is prime.*	*Return True if n is unique.*
CodeT5++		
+MRV	*Returns True if n is prime.*	*Return True if n is unique.*
+CDS	*Returns True if n is prime.*	*Return True if n is unique.*
+MRV,CDS	*Returns True if n is prime.*	*Return True if n is unique.*
+MFN	*Returns True if n is prime.*	**Return True if n is a valid prime.**

Table 5: Docstrings generated for isPrime function.

D Appendix: Supplemental Material

For pre-training, we extended the pre-training scripts for T5 using PyTorch Lightning[4]. In addition, we extended CodeT5[5] by forking and adding new pre-training objectives. Also, we wrote scripts to generate MRV, CDS, and MFN datasets from the Code Search Net dataset. Finally, the code and scripts to pre-train and fine-tune CodeT5++ are available in our Github repository[6]. In addition, all our pre-trained and fine-tuned models are available in Google Cloud Storage[7].

[4] https://github.com/manueldeprada/Pretraining-T5-PyTorch-Lightning

[5] https://github.com/salesforce/CodeT5

[6] https://github.com/aumahesh-mids/w266-summer-2022-project

[7] GCP Cloud Storage bucket: gs://mrv-cds/models