Question Answering for Code Summarization

Alex Cheng
Department of Statistics & Data Science
Carnegie Mellon University
Pittsburgh, PA 15213
abcheng@andrew.cmu.edu

Liz Chu
Department of Statistics & Data Science
Carnegie Mellon University
Pittsburgh, PA 15213
echu2@andrew.cmu.edu

Abstract

Code documentation and summarization is a critical part of software maintenance and integrity. Recent research has shown promising results in the automatic machine generation of documentation by leveraging large language models. In this work, we take the novel approach of reformulating the text generation task as a question answering task, and also provide a novel dataset for question answering code summarization. We find that CodeT5+ overall performs better with our given architecture, but our limitations significantly hinder our results in the losses, F1 scores, and sample results we provide.

1 Introduction

Documentation is a problem that many software engineers and computer scientists face within the lab and at their workplace. Recently, the advent of large language models (LLMs)[4], [20], [14] [12] have provided many advancements in this field. These models first pre-train on large amounts of code in a cornucopia of programming languages to gain an understanding of code grammar and syntax from large code bases such as Github. From there, the model is further fine tuned onto the specific downstream task [21] to achieve better results.

Even though quantitative measures reflect good performance from LLMs, a qualitative glance at these models show that the answers generated are a often very varied and lack structure. Furthermore, because the training data is based on real documentation rather than ideal documentation, this causes models to deviate from structures outlined in their respective programming language style guidelines and documentation.

To alleviate this, our team decided to take the novel approach of recontextualizing this problem as a question answering task, as well as creating a novel dataset for future use for other researchers. The dataset would contain various questions about different aspects and qualities of each function in order to generate robust and complete documentation. To formalize, given a piece of code C and a set of prompts P, we want to generate quality responses that are truthful to the piece of code, using our dataset **Code Question Answering Dataset (CoQuAD)**[1] and our transformer-based model [2].

2 Background

In our previous approach, we followed baseline models by implementing a vanilla Bi-LSTM to act as an encoder-decoder on a Python dataset. The model has the advantage of being lightweight and easy to understand, but suffers from the vanishing gradient issue, especially with large code or documentation chunks. The results of our baseline approach is detailed in our Results section.

[1]`https://huggingface.co/datasets/aalexchengg/codesearchnet_qa`
[2]`https://github.com/lizchu413/617project`

3 Related Work

Code understanding and summarization has been a popular task the last few years, with very recent models heavily utilizing the transformer architecture [1], [10] based on self-attention [18] to achieve performances that past architectures such as a LSTM [16] [17] or a RNN could not. Recent work has also focused on deviating away from using an abstract syntax tree (AST) to represent the code's hierarchical structure, and instead learning the hierarchical features within the model's latent space. Some important models that have come out are Microsoft's CodeBERT [8], which utilizes BERT [7]'s encoder-only architecture to discover features of the code in the latent space. Salesforce's CodeT5 and CodeT5+ models [19], [20], based on the T5 architecture [15], utilized both an encoder-decoder architecture, with separate parts of the architecture activated during specific tasks in order to have one uniform model for several code-related tasks.

These prior works have often viewed code summarization as a text generation task, where given a piece of code as the input, the documentation would be generated by the decoder as the output. There has also been a significant amount of work in pretraning techniques; while CodeBERT focuses on the standard BERT pretraining tasks of masked language model (MLM), and next sentence prediction (NSP), CodeT5 augments pretraining with several new tasks based on T5 pretraining and their own novel pretraining techniques, such as span denoising, text-code contrastive learning, and causal language modeling (CLM). However, this may lead to incomplete or unintelligible responses, as the documentation often tries to create a one sentence summary of the code rather than a complete and robust description of possible confusing factors of the code. Thus, we opted for a question answering approach.

Transformers have also been used for question answering tasks [13], as well as cross-lingual question answering [2], which enables the transformer to have an understanding between multiple languages and answer queries in different languages.

Lastly, while there are few datasets that provide text-code pairs [3] [6], the most widely used dataset is CodeSearchNet [11], which provides parallel examples from six languages (Ruby, Python, Java, Javascript, Go, PHP), generated from both human-annotated annotations as well as distantly generated annotations. The opposite could be said about raw-code data - the Github code dataset [3] contains 115M files from 32 different programming languages, and comes out at almost 1 terabyte of data, making pretraining of code LLMs extremely robust.

4 Methods & Model

4.1 Dataset Creation

To create CoQuAD, we first used the CodeSearchNet dataset to access parallel code and documentation instances. Since there are several instances of bad documentation, we decided to throw out all training examples where the documentation length $l < 4$ and the number of lines of code $n > 5$. This would ensure that our dataset was quality controlled to some extent. From there, we followed Berkeley the Library Guide guidelines for good code documentation points. Specifically, we wanted to answer these questions:

1. What does the function do?
2. For each function parameter p, what is p?
3. What does the function return?

To do this, we used rule based methods to first extract the parameters out from the function code string. In the case that the rule base parsing failed, we would omit the training example. From there, we would use the documentation for each training instance to generate answers to each question. Only answers that had a specific marker indicating its information would be included; if a parameter was not explicitly mentioned in the documentation, we would not include it. An example parse can be found in Figure 1: notice how since there is no special indicator for parameter b, we choose to not include any questions about it as we have no labels.

[3] `https://huggingface.co/datasets/codeparrot/github-code`

Code	Documentation

`def add(a, b):`
`return a + b`
	`adds two numbers`
	`@param a first number`
	`@return summation`

Questions Generated
What does the function do? What is a? What does the function return?

Figure 1: Dataset parsing example

After processing our data into a new question answering dataset, we have augmented our dataset to include extra information about each function, and as such have also increased the number of training examples overall. The table in Figure 2 will provide some insight into the features of our dataset.

CSN	Our Data	Language
457,461	637,959	Python
578,118	1,142,916	PHP
496,688	1,048,713	Java
138,625	167,917	Javascript
346,365	337,759	Go
53,279	89,327	Ruby
2,070,536	3,424,591	Total

Figure 2: Number of training examples in each dataset

Question	Count
What function does	1,970,827
What param p is	738,126
What function returns	715,638

Figure 3: Question Distribution

4.2 Model Selection

We decided to select CodeT5+ and GraphCodeBERT [9] to conduct our experiments. We chose these two models because they are both transformer based architectures, but take very different approaches in how they understand the code data.

4.2.1 CodeT5+

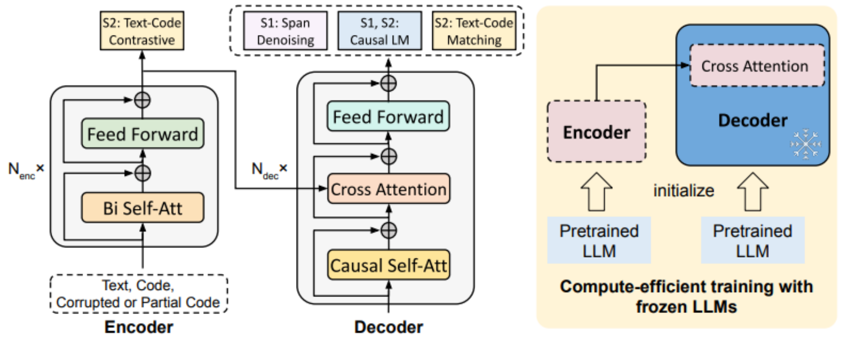


Figure 4: CodeT5+ Architecture. While not relevant to our current task, CodeT5 is also pretrained using off-the-shelf LLMs to increase computational efficiency.

CodeT5+ contains both an encoder and a decoder and follows the T5 architecture, which can be seen in Figure 4. Both the encoder and the decoder are trained during pre-training, but only specific parts are used during inference. Pre-training is also split up into two parts - the former is solely unimodal code data for code understanding, while the other is bimodal text-code paired data for documentation generation and translation into English. CodeT5+ is also built to have all tasks as text input and generated text output, which provides flexibility and malleability in how training data is given.

4.2.2 GraphCodeBERT

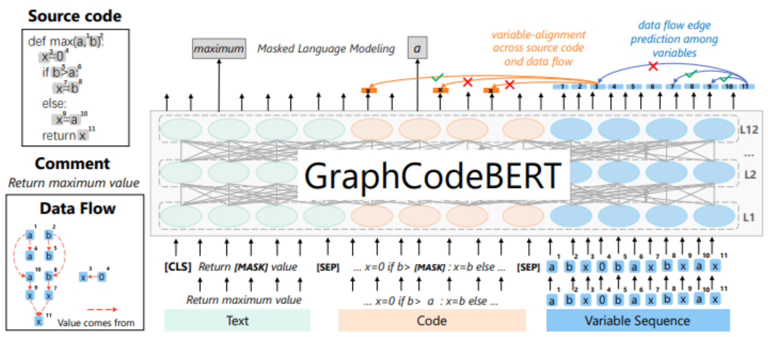


Figure 5: GraphCodeBERT Architecture

GraphCodeBERT is a version of CodeBERT that attempts to retrieve the hierarchical information of a piece of code through semantic information rather than an AST. GraphcodeBERT pre-training involves the usual MLM task involved with BERT, as well as two other structure-involved tasks to discover graphical features of the code. This architecture, which can be found in Figure 5 is an encoder-only architecture, as its focus is on finding the features rather than generating new outputs.

4.3 Model Training

With our processed dataset CoQuAD, we first generate tokenized sequences. To treat the QA problem as a Seq2Seq problem, we prepend the question to our context as our input sequence as seen in other summarization to QA models [4]. Thus, if our function

```
def add(a, b):  return a + b
```

had the question "What does this function do?", our input sequence (before tokenizing) would be

"question: What does this function do? context: `def add(a, b):  return a + b`".

Our result sequence would simply be the answer to the question. In the previous example, our output sequence (before tokenizing) would be

"Adds two numbers."

We do this for all splits of the dataset: train, validation, and test.

Then, for each experiment, we use a pre-trained model (either CodeT5 or CodeBERT) in a Trainer object (from the `transformers` library) to train our model on the tokenized training dataset. We run all the models for 8 epochs on 4000 training examples and 100 validation examples (due to memory and time limitations). We also limited the maximum number of tokens of any sequence to be 120 tokens to prevent overfitting. Other than loss, we also compute F1 scores on the validation dataset. (A common QA metric is EM, or exact match, but given our limited training set size and time trained, the validation EM was always zero and we thus omitted it from our metric reports.) We finally evaluate the models and save all losses and F1 scores with builtin functions from the Trainer class.

[4] `https://huggingface.co/MaRiOrOsSi/t5-base-finetuned-question-answering`

4.4 Experiments

With our baseline training loop constructed, we conducted experiments by varying:

- Base Model: {CodeT5 [20], CodeBERT [8]}
- Learning Rate: {5e-4, 5e-5, 5e-6}
- Weight Decay: {0.001, 0.0001, 0.0}

Thus, we have a total of twelve experiments ran (all on 4000 training examples and 100 validation examples for 8 epochs), six per base model. Our training batch size was 8 and our evaluation batch size was 4, as computing the metrics took substantially more memory during training time. Again, due to time and memory limitations, we were unable to run these models for longer than 8 epochs or with larger batch sizes. On average, the time to train with this batch size and dataset size on 3 RTX 2080 GPUs was around 40 minutes, and it would take 64 hours to train on the entire dataset for three epochs.

We chose to use these two baseline models as they are pre-trained on existing code data. CodeT5+ specifically, created by Salesforce, has been used for code summarizing in the past, which suggests that it may be useful for a modified code summarization task (QA). CodeBERT was also pretrained on NL-PL (natural language-programming language) pairs in six languages, all of which our training dataset have (Python, Java, JavaScript, PHP, Ruby, Go).

5 Results

For each of the twelve experiments, we ran 8 epochs and recorded the training and testing average cross entropy loss per epoch. Plots of the loss graphs for each model can be found in the appendix, and the final F1 scores we achieved can be found in Figure 6. As mentioned previously, QA tasks normally use F1 and EM (exact match) as metrics, but given that our training time was so short and we had so few training examples, our EM's were always zero and was thus an inadequate metric to use.

F1, calculated as follows, is a common metric for prediction tasks as it uses both precision and recall.

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

	LR = 5e-4	LR = 5e-5	LR = 5e-6	WD = 0.001	WD = 0.0001
CodeT5	0.0848	0.0867	0.0871	0.1101	**0.1102**
CodeBert	0.0083	0.0273	0.0083	0.0259	**0.0282**

Figure 6: Final F1 Scores

6 Discussion & Future Work

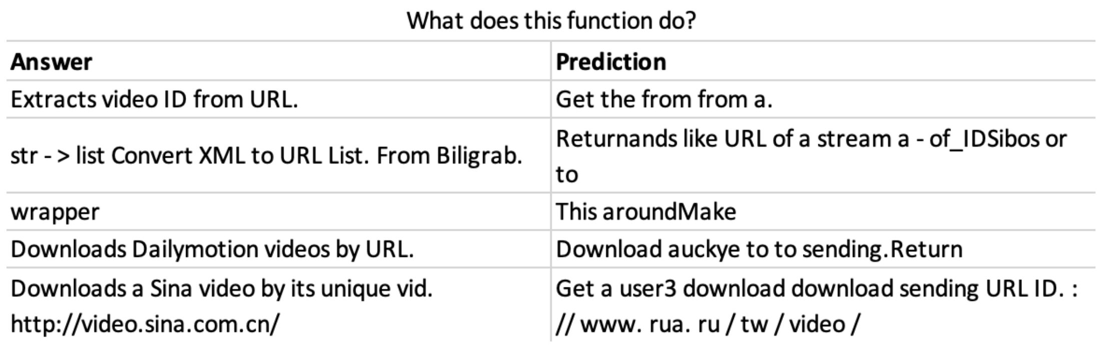
6.1 Analysis

Before we discuss how our models performed in our experiments, we revisit the baseline model presented in the midway report, which used a vanilla Bi-LSTM to act as an encoder-decoder. This time, we pass in our current dataset and keep most of the parameters the same when possible. We ran this vanilla Bi-LSTM on a training set of 4000 examples, 100 validation examples, and 100 test examples for 8 epochs, again limiting all examples to 120 tokens. With the baseline model, we achieved a final F1 score of **0.8832**, which is substantially higher than all of our more complex models.

Our baseline model surprisingly outperforms all the experimental models we trained. This is probably due to the pre-training of our transformer based models - since our baseline has only seen data from CoQuAD, it is bound to overfit to the training data and perform well, while our transformer-based architecture has already trained on millions of examples outside of CoQuAD and would require significantly more training to achieve state of the art results, which unfortunately could not be achieved with our current computational abilities.

Furthermore, although we had high F1 scores with our baseline model, the predictions our model gave were extremely generic and seemed to overfit (with most of the responses to "What does this function do?" being "Returns the given number of the given class."). Further analysis of this baseline could be done with better metrics (say, ROUGE or BLEU scores) given more time.

Figure 7 shows some of the predictions that our default model (CodeT5, learning rate of 5e-5, no weight decay) made. Function bodies were omitted for space.



What does this function do?

Answer	**Prediction**
Extracts video ID from URL.	Get the from from a.
str - > list Convert XML to URL List. From Biligrab.	Returnands like URL of a stream a - of_IDSibos or to
wrapper	This aroundMake
Downloads Dailymotion videos by URL.	Download auckye to to sending.Return
Downloads a Sina video by its unique vid. http://video.sina.com.cn/	Get a user3 download download sending URL ID. : // www. rua. ru / tw / video /

Figure 7: Default Model Prediction Examples

Some qualitative observations we can make on these results are as follows:

- *Action verbs are present and for the most part, correct.* We see that in the 3 of the five observations, the model returns "Get" or "Download" as its first token.
- *The model often struggles with URLs.* The last example shows this clearly — although our model was able to distinguish that there is an URL involved, the URL returned does not make sense (with Russian and Taiwanese domain names for a function that downloads Chinese videos).
- *Our golden data's quality must be taken with a grain of salt.* Our third example's true answer is just "wrapper," which doesn't tell us anything about what the inner function contains.

All of these results could be due to our limited training time, training set size, and the flawed, real-world nature of the dataset itself. This also correlates to some of the results we got, as we notice that our F1 scores are pretty low.

6.2 Limitations and Future Work

A large limitation that we had, as previously mentioned, was computational power. We were unable to run large batch sizes, large models, and were also unable to utilize the entire dataset, which definitely influenced our findings. Another limitation that we had was that there was not an even representation of questions, languages, and code lengths, which may cause certain training instances to not perform as well. Some ways to mitigate this are stratified sampling, or exponentially smoothed weighting (as mentioned in mBERT[7]). Another interesting idea is to have dynamic sampling distributions for code length, where the model first learns easier examples before building up to learning mostly harder examples, and then eventually settling on the true distribution of examples.

Lastly, there is still a significant amount quality control needed for the dataset itself, as there are still many labels in the dataset that could be considered low quality or incorrect. However, due to time constraints, we were unable to parse or filter out these examples.

References & Citations

References

[1] Wasi Ahmad et al. "A Transformer-based Approach for Source Code Summarization". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, July 2020, pp. 4998–5007. DOI: 10.18653/v1/2020.acl-main.449. URL: https://aclanthology.org/2020.acl-main.449.

[2] Akari Asai et al. "One Question Answering Model for Many Languages with Cross-lingual Dense Passage Retrieval". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 7547–7560. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/3df07fdae1ab273a967aaa1d355b8bb6-Paper.pdf.

[3] Antonio Valerio Miceli Barone and Rico Sennrich. *A parallel corpus of Python functions and documentation strings for automated code documentation and code generation*. 2017. arXiv: 1707.02275 [cs.CL].

[4] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[5] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179.

[6] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. "On the Coherence between Comments and Implementations in Source Code". In: Aug. 2015, pp. 76–83. DOI: 10.1109/SEAA.2015.20.

[7] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[8] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL].

[9] Daya Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. 2021. arXiv: 2009.08366 [cs.SE].

[10] Juncai Guo et al. "Modeling Hierarchical Syntax Structure with Triplet Position for Source Code Summarization". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 486–500. DOI: 10.18653/v1/2022.acl-long.37. URL: https://aclanthology.org/2022.acl-long.37.

[11] Hamel Husain et al. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. 2020. arXiv: 1909.09436 [cs.LG].

[12] Shuai Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021. arXiv: 2102.04664 [cs.SE].

[13] Khalid Nassiri and Moulay Akhloufi. "Transformer Models Used for Text-Based Question Answering Systems". In: *Applied Intelligence* 53.9 (Aug. 2022), pp. 10602–10635. ISSN: 0924-669X. DOI: 10.1007/s10489-022-04052-8. URL: https://doi.org/10.1007/s10489-022-04052-8.

[14] Erik Nijkamp et al. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=iaYcJKpY2B_.

[15] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG].

[16] Yusuke Shido et al. *Automatic Source Code Summarization with Extended Tree-LSTM*. 2019. arXiv: 1906.08094 [cs.LG].

[17] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 3104–3112.

[18] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[19] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL].

[20] Yue Wang et al. *CodeT5+: Open Code Large Language Models for Code Understanding and Generation*. 2023. arXiv: 2305.07922 [cs.CL].

[21] Fuzhen Zhuang et al. *A Comprehensive Survey on Transfer Learning*. 2020. arXiv: 1911.02685 [cs.LG].

Appendix

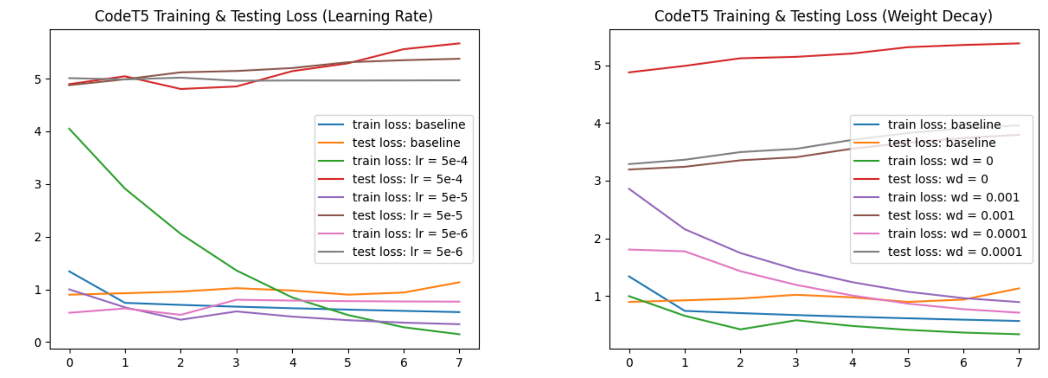


Figure 8: CodeT5 Experiments: Train & Test Losses. We note that our test loss is significantly higher than our training loss for every experiment, but overall, a learning rate of 5e-6 performs best and a weight decay of 0.001 performs the best.

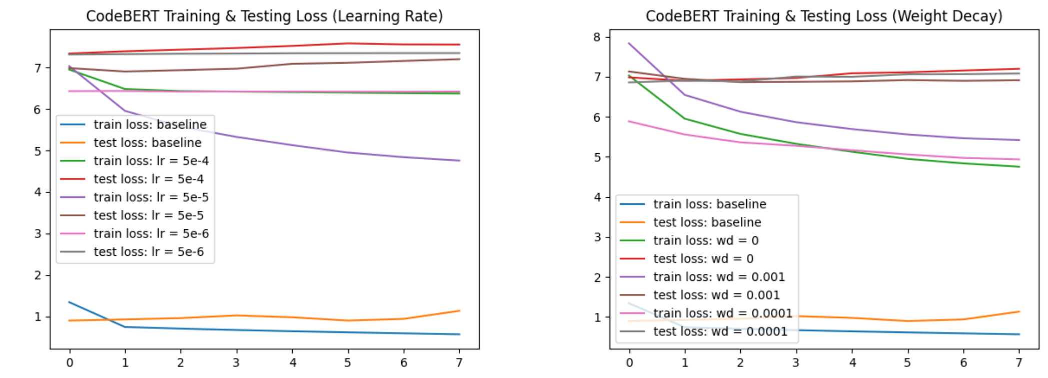


Figure 9: CodeBERT Experiments: Train & Test Losses. Similar observations can be made with the CodeBERT experiments, except a learning rate of 5e-5 performs the best instead of 5e-6.

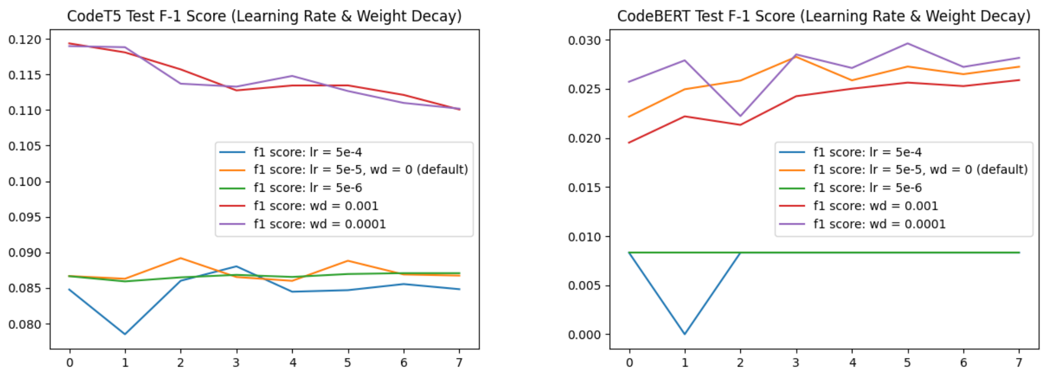


Figure 10: CodeT5 & CodeBERT Experiments: F1 Scores. Overall, CodeT5 has better F1 scores than CodeBERT, with weight decay of 0.0001 having the highest in both CodeBERT and CodeT5.