AI for Software Engineering Final Project
Knowledge Distillation for Code Description Generation
By Danny Otten, Matthew Chen, Chuntian Chi, and Kate Balint

Introduction

The demand for automated code generation has grown exponentially with the development of large language models (LLMs) like GitHub Copilot and Amazon CodeWhisperer. These systems have shown remarkable capabilities in translating natural language specifications into executable code, and achieve performance that is comparable to human programmers in certain tasks. However, one significant challenge remains that the computational complexity of these models often exceeds practical resource constraints, especially in scenarios that require real-time code generation or deployment on edge devices.

Knowledge distillation offers a solution to this efficiency problem. First introduced by Hinton et al. (2015), it involves training a compact student model to mimic the behavior of a larger, more complex teacher model. The idea is to transfer the knowledge gained by the teacher model into the student model, so the student can achieve performance comparable to the teacher, but with fewer parameters and computational resources. In the context of code generation, knowledge distillation is more challenging as code generation requires maintaining strict syntactic correctness and semantic consistency.

To make training simpler, we decided to focus on only Python. In order to train each model, we need data containing both code blocks and a sample description of what they do. This is provided in the CodeSearchNet dataset on Hugging Face from Evaluating the State of Semantic Code Search, which contains a total of 503,502 Python function snippets obtained from libraries.io. They are only acquired from public open-source GitHub repositories with valid licenses for distribution to ensure that the data does not infringe on any privacy rights or copyrights. Furthermore, to ensure the code is related and relevant to real-world problems in the industry, toy projects made for fun are excluded by filtering for more popular projects. Popularity is defined by frequency of use by other projects, stars from other users, and how many times other users have forked their project, meaning they obtained a copy on their local machine. The quality of the dataset was maintained by removing duplicates, short programs, and functions that are frequently redundant across scripts or contain no important content, like __init__ or testing. The Jaccard similarity-based method from The Adverse Effects of Code Duplication was used to assess how alike distinct pieces of code were to each other and marked copies according to comparisons of their literal sets and token sets.

Implementation

We chose to implement knowledge distillation using the SetFit transformers framework due to its default provision of a Trainer and DistillTrainer, which uses the output and parameters of the teacher model to train a student model. Speaking of the models, we decided to use models from the sentence transformers library, since they are compatible with SetFit. For our teacher model, we chose all-mpnet-base-v2, which has 109 million parameters and for our student model, we chose all-MiniLM-L12-v2, which has 33.4 million parameters. Both use a pretrained Microsoft model and finetuned with a large corpus containing text from various websites.

Due to the many issues with memory detailed below, we trained several small models with varying degrees of success. For the sake of brevity, this section will discuss only the largest model and best-performing model.

For context on the unideal speed of the models, our largest teacher model took about 3 hours to train on a dataset of 7800 code-documentation pairs with a batch size of 4 for 10 epochs and 2500 steps. Since the models were training on such a limited dataset, we found that reducing the epochs and steps for training to avoid overfitting successfully improved the results for the best models. Below are the hyperparameters we used for the distillation training for each model:

Largest Model:
Teacher: 7800 train dataset, batch = 4, epochs = 10, steps = 2500
Student: 2310 train dataset, batch = 16, epochs = 10, steps = 500

Best-Performing Model:
Teacher: 3900 train dataset, batch = 1, epochs = 1, steps = 10
Student: 115 train dataset, batch = 16, epochs = 1, steps = 500

Errors, Challenges, and Limitations

SetFit and SentenceTransformers

The main challenge we faced was from using SetFit, specifically version 1.0.3, which was released on January 16th, 2024. Due to its recency, it has not been stress-tested as much as other options for training models. We experienced package collisions with certain versions of transformers and had to choose specific models that were compatible with SetFit.

Despite its built-in knowledge distillation implementation, using SetFit may have been a mistake. This is primarily because it is a framework for SentenceTransformers, which is not optimized for our task of generating code descriptions. Instead of using code or natural language tokens, this module requires an untokenized full string as input.

When we initially attempted to train this model on code and documentation string tokens, SetFit's model card creator threw the following error as it tried to split the input sentence to obtain the word count:

```
…site-packages\setfit\model_card.py", line 307, in
add_naive_word_count sample["word_count"] =
len(sample["text"].split(" ")) AttributeError: 'list' object has no
attribute 'split'
```

We edited this word count check out of model_card.py to check if the error was exclusive to the model card setup, but soon ran into another which is not so easily worked around:

```
…sklearn\utils\_array_api.py", line 380, in _asarray_with_order
    array = numpy.asarray(array, order=order, dtype=dtype)
ValueError: setting an array element with a sequence. The requested
array has an inhomogeneous shape after 1 dimensions. The detected
shape was (3900,) + inhomogeneous part.
```

In this instance, our training set contained 3900 examples. The error shows that using an array of code tokens for model input was not a viable strategy, forcing us to train with unbroken strings of python methods and documentation.

Aside from preventing the model from learning the relationships between neighboring tokens to increase its understanding of code, this aspect of SentenceTransformers has the effect of only being able to output full strings that were found in its training data rather than a combination of tokens in the output examples. This means that neither the teacher's nor student's output can be relevant to the input when making a prediction, unless this input is extremely similar or identical to an example from the training set.

Finally, SetFit seems to be highly inefficient compared to Hugging Face's default trainers, making it take a long time and put significant strain on the machines that it ran on, even completely crashing bg13 with only 30k data points. In future works, it may be a better idea to implement Knowledge Distillation with PyTorch or TensorFlow directly instead of through SetFit.

Dataset Errors

The CodeSearchNet dataset seemed like the best available dataset for documentation generation, as it includes a documentation string and documentation tokens for each method. However, it also contains this documentation *as part* of each Python code segment in the form of a method description header. This results in a completely inconsistent docstring format, as certain descriptions contain only plaintext while others contain code terms and URLs.

Had our implementation not contained the errors discussed above, the model could have noticed this pattern and began outputting the description for every test run. There are a variety of ways to mitigate this, such as making sure every description is represented by a single token (if SetFit could use tokens) or removing this description from each code block in the dataset. As it is, our models did not have the flexibility to recognize this, so it's unlikely that it influenced the final results.

Furthermore, we found out that CodeSearchNet was part of the finetuning corpus for the models we selected. However, given that CodeSearchNet was only 1 of 32 datasets and only 0.0984% of all the sentences in the corpus, the models likely had minimal memorization of the dataset. We were also likely unable to train the models for long enough or with enough of the dataset for this issue to have any effect.

Training Errors

In the process of attempting to train our model, we encountered numerous errors involving the environment, dependencies, and hardware. Our personal computers did not have sufficient GPU for a model of the size required for this project, so we elected to use the university lab machines. However, this resulted in many CUDA and GPU errors, even with the more powerful hardware. The errors we encountered on the lab machines were as follows:

```
AttributeError: 'CallbackHandler' object has no attribute 'tokenizer'
```

SetFit has an incompatibility with transformers which is solved by reinstalling transformers with a version less than or equal to 4.42.2. In our case, we chose 4.28.1 since we were using conda and conda-forge provides limited package options. [Setfit does not work with transformers dev-version with error "'CallbackHandler' object has no attribute 'tokenizer'" · Issue #564 · huggingface/setfit]

```
RuntimeError: CUDA out of memory. Tried to allocate 432.00 MiB (GPU
0; 39.38 GiB total capacity; 14.00 GiB already allocated; 283.81 MiB
```

```
free; 14.09 GiB reserved in total by PyTorch) If reserved memory is
>> allocated memory try setting max_split_size_mb to avoid
fragmentation.  See documentation for Memory Management and
PYTORCH_CUDA_ALLOC_CONF
```

This error occurred because our batch size was too high and we used the default max_steps. Setting an explicit number to the max_steps parameter of our TrainingArguments allowed the model to run with more control of its GPU expenditure.

```
…torch-env/lib/python3.9/tempfile.py", line 255, in _mkstemp_inner
     fd = _os.open(file, flags, 0o600)
FileNotFoundError: [Errno 2] No such file or directory:
'/tmp/tmp_seju4fw/port-138315.txth64h64kj'
```

We reported this error to W&M's CS Support but have not been given a definitive solution thus far. As a Techie, Kate has been investigating it further. We presume this issue has been effectively resolved within the scope of our project, since it hasn't been encountered since, but its origin is so far unknown.

```
…torch-env/lib/python3.9/site-packages/wandb/sdk/lib/sock_client.py",
line 221, in send_record_publish
```

Here, wandb inexplicably failed to upload its records to the machine. This remains an unsolved mystery, as this error did not appear in future runs of the code, but we suspect it was caused by clearing cookies and logging out of the website.

Evaluation and Results

A prediction model such as this one would be extremely unlikely to output the expected documentation exactly, so we used BLEU to evaluate the accuracy of our model. Because SentenceTransformers limits the possible results, single-gram precision is necessary to determine how well the model can identify patterns within an input code block. Before the predictions can be tested, the training data must first be encoded as embeddings and fit to the SetFit model head. Then ".predict()" can be called for the model with a Python method as input.

All models were evaluated on the same 10% subset of the test.csv dataset containing ~2200 examples created by loaddata.

We evaluated both the large model and the best performing model, and analyzed their generated results respectively. For the largest models, the student model shows a focused approach to code summarization that is primarily centered on hyperparameter optimization and model evaluation, consistently returning "Set of hyperparameters" as its predicted output, which may suggest that the distilled knowledge focuses more on

summaries of the optimization functions from the teacher model. The teacher model outputs more diverse code summaries including video processing, URL management, and various arithmetic operations. The student model displays a consistent BLEU score of 0.0 across all provided codes, while the teacher model has a slightly better average BLEU score of 0.00541. This scoring warrants training our model on larger and more diverse datasets, or areas requiring further model refinement.

For our best-performing models, the student model mainly produces two types of predicted outputs: “Wrapper function for Series arithmetic operations, to avoid code duplication.” and “convert to our native types format, slicing if desired”. While it shows some capabilities of generating understandable code summarizations, its BLUE scores hovering around 0.0 suggest room for improvement in generation quality. Again, the corresponding teacher model produces more diverse outputs, including “Split a RNN `model` in groups for differential learning rates.”, “Sub-classes to define. Return a sliced object.”, “Evaluates the model on a test dataset.”, “Provide a transactional scope around a series of operations.”, and “Utility to find a constraint name in alembic migrations”, each having BLUE scores from 0.222 to 0.0, with the average BLUE score of 0.0492. The knowledge distillation process has transferred some capabilities from teacher to student model. However, opportunities exist for enhancement, particularly in improving the student model’s BLUE scores and enabling more diverse outputs, through better knowledge transfer mechanisms. Future improvements could focus on incorporating more complex features from the teacher model while maintaining the student model’s efficiency.

Conclusion

We as a team learned a lot from this project, including what not to do. The use of SetFit made the project much more difficult than we had anticipated. If we were to attempt this project again in the future, based on our experiences this time around, we would elect to either use PyTorch or a fine-tuned transformer.

Another potential area of improvement is standardization of environments. With all the difficulties we encountered concerning the lab machines, it was necessary for us to switch machines relatively often, which involved resolving package and dependency errors over and over again. In the future, given the resources to do so, it would be extremely beneficial to have one development environment on one machine in order to not waste time on resolving these issues.

As frustrating as working on this project could be at times, we believe the process of doing so was beneficial to us all as developers. Not only did we gain a greater

understanding of how knowledge distillation works in this context, but we also learned how to resolve issues of unknown origin and work together as a team in discouraging circumstances. Overall, we feel that this was a valuable experience for our group that will better prepare us for future projects in both our educational and professional endeavors.

Resources and References

GitHub repository: https://github.com/dsotten/AI4SEKnowledgeDistillation
Processed data from CodeSearchNet: dataset
Largest teacher/student model: models_large
Best teacher/student model: models_small