

- Synopsis
 - As part of this project, I tried to see if incorporating AST information would improve the performance of automatic documentation generation in a Transformer model. The use of structural information in the modeling of programming languages is currently a popular research topic, so confirming or contradicting the conclusions of the original paper would be useful to the research community.
 - The scope decreased as I originally intended to compare with a dataset with fewer data leakage issues, but I was unable to get a Python dataset that was of sufficient size, had AST representations extracted in a usable way, and worked with the system I was working with. I planned to use Python 150k
 - There would have been more novelty under my original scope (evaluating against improved methods data processing methods), but the work in this project at least partially reproduces the results of a paper on a recent novel subject (Transformers in source code modeling). An additional improvement in this project is making the AST modeling portion more readily available out of the box, which was not the case in the original repository.

- RQs

- 1: Does incorporating AST help?
 - The model performance didn't really change versus the values the authors reported for the baseline, which is consistent with the claims of the paper that their attempts to incorporate AST were not beneficial. However, there are several caveats to this which I will discuss.


| Model | BLEU | METEOR | ROUGE-L |
|-------------------------------------|-------------------------|--------|---------|
| AST Model | 43.94 (as of epoch 170) | 25.93 | 53.23 |
| Base Model | 43.41 | 25.91 | 52.71 |
| Full Model | 44.58 | 26.43 | 54.76 |
| Dual Model (different benchmark) | 42.39 | 25.77 | 53.61 |

- Methodology (and methodological issues)
 - I ended up using their provided AST dataset. Their experiments with ASTs were not well documented, though from what I saw on the author's responses on github to a similar question, experimenting with ASTs was a low priority for them. They did provide a mostly complete subset of their Java dataset with ASTs extracted, which is what I used.
 - The other major issue with my methodology is that because I ended up using one of their datasets, and given that the data collection practices they followed are known to have issues with data leakage, this model also was trained on leaky data. This came up in an interesting way in one of my earlier test runs, which I discuss below.
 - To elaborate on the data leakage issues, the main problem is that the data is split at a method level and not at a repository or even source file level; i.e. different methods from the same source file may be in the train, dev

(validation), and test sets. There may be similar methods in a given source file, so this risks exposing your model to test data while training.

○ Example:

- Prediction: "copy the contents of the given byte array to the given outputstream . closes the stream when done ."
- Reference: "copy the contents of the given byte array to the given outputstream . leaves the stream open when done ."
- Code:

```
94
95  /**
96  * Copy the contents of the given byte array to the given OutputStream.
97  * Leaves the stream open when done.
98  * @param in the byte array to copy from
99  * @param out the OutputStream to copy to
100  * @throws IOException in case of I/O errors
101  */
102 public static void >>copy(byte[] in, OutputStream out) throws IOException {
103     Assert.notNull(in, "No input byte array specified");
104     Assert.notNull(out, "No OutputStream specified");
105
106     out.write(in);
107 }
```

- [https://code.yawk.at/org.springframework/spring-core/5.1.5.RELEASE/org.springframework/util/StreamUtils.java#org.springframework.util.StreamUtils%23copyToByteArray%28java.io.InputStream%29](https://code.yawk.at/org.springframework/spring-core/5.1.5.RELEASE/org.springframework.util/StreamUtils.java#org.springframework.util.StreamUtils%23copyToByteArray%28java.io.InputStream%29)

- While the body of the summaries are nearly identical, they differ at the end in a contradictory manner which in the real world could have serious consequences (mistakenly thinking that a stream was closed when it was still open could cause a host of issues). I was curious as to how the prediction could come so close to the reference, and yet chose the opposite meaning of the second clause. Looking at the original class, there are multiple methods with similar behavior, signatures, and documentation as is common in Java. The training set actually has examples with "copy the contents of the given byte array to the given outputstream . closes the stream when done ." This example also highlights some of the issues with metrics like BLEU; I found this example while I was searching in the output file for high BLEU score samples to spot check that the model was making reasonable predictions. It only takes a very small change in the sequence to produce something with significant semantic differences.
- Another observation was that the AST model was very slow compared to the text only model. My experiments with running the model on text-only Python took about the same time per epoch but was using a 10-20x larger dataset. This matches both the claims of the author and the mechanics of how Transformers scale with sequence length. In regards to the latter, Transformers encode a sequence as a series of pairwise mappings between elements of the sequence, thus it is $O(n^2)$ with respect to sequence length. According to the authors of the original paper, the average sequence length for Java increased from 120 to 172 when going from text to AST; the increase in computational effort could be even more pronounced depending on the distribution of sequence lengths.
- 2: How do their results change given "better" data
 - I wasn't able to address this question. There were some issues with regards to transparency here. I planned to use a Python dataset originally. While they describe

their data collection and processing procedures in a fair amount of depth, they do not actually follow them for Python. The authors use a dataset from a benchmark they compare to, but I wasn't all that clear on that dataset's processing. One of the authors explicitly says that they used that other group's data instead because they had trouble replicating the the performance of the benchmark model on their own dataset.

- Deliverables

- github

- <https://github.com/rd1013/6156-course-project>
 - Milestones and documentation: https://github.com/rd1013/6156-course-project/tree/robert_changes/6156-project-documentation

- Reuse:

- Data

- Python 150k
 - <https://www.sri.inf.ethz.ch/py150>
 - Corresponding docstrings are not readily available. Chirkova et al. used this dataset to evaluate various Transformer configurations. I was wondering why they didn't use it for generating documentation, since the Ahmad et al. paper ended up being their main point of comparison. This is probably the answer.
 - I may try to augment this at some point with the docstrings and experiment on this set. The dataset has links to the repositories of the original code, but many of those did not have docstrings, so it may end up being much fewer than 150k files.
 - University of Edinburgh python + docstring
 - <https://github.com/EdinburghNLP/code-docstring-corpus>
 - All three datasets from Ahmad et al.
 - <https://drive.google.com/drive/folders/1Mx0xEPZfQzb5h0z753XV-JgoWUuxiuKZ>

- Code

- Original project code
 - <https://github.com/wasiahmad/NeuralCodeSum>

- Papers

- Original paper on using Transformers for source code summarization
 - <https://arxiv.org/abs/2005.00653>
 - Ahmad et al.
 - A survey of Transformers applied to source code
 - <https://arxiv.org/abs/2010.07987>
 - Chirkova et al.
 - Paper describing methodological issues in source code summarization
 - <https://arxiv.org/abs/1904.02660>
 - Leclair et al.

- Self-assessment:

- Challenges:

- that weird division bug in the code that I haven't had time to deep-dive

- authors mention the code is tested; however, the code is really weird; there's a mix of:
 - "from __future__ import division"
 - single / division for integer division a la python 2.X
 - explicit statement in readme that users should be on python 3.6+
 - I don't think modern DL libs like pytorch even support 2.X
- <https://github.com/wasiahmad/NeuralCodeSum/blob/master/c2nl/translator/bam.py#L131>
- Difficult to replicate results
 - I started with trying to use a python dataset, but they didn't actually use their own data processing for python. They ended up asking another group for their python dataset.
- Difficulties with finding a reasonable way to pass AST into Transformer
 - Underestimated the difficulty of encoding the AST in a way that a Transformer can represent
- Starting the coding earlier would have given me more time to adapt/pivot
 - E.g. I considered a last minute switch to a tool that used this sort of model, but decided against it; something like a plugin for automatic documentation generation
- Accomplishments
 - As far as accomplishments, I do think I was able to contest the claims of the original paper that AST information was not useful, though as I mentioned, with the data leakage issues unresolved, and the models not operating on the same dataset, it's difficult to make a strong claim.
- Learnings
 - I think messing around with model architectures to get something that works ended up being a lot harder than I expected.
 - The longevity of some of these methodological issues was interesting as well. This was something that came up both in this project as well as in various papers I looked at. Even if everyone knows that there are issues with doing something one way, people will still follow that technique to allow for easy comparisons with prior works. I ended up in the exact same situation.
 - Somewhat random learning
 - I ran into an interesting and unexpected parallelism overhead; heat. I was running a 2 GPU system, but my case/cooling setup actually wasn't able to cool two GPUs running at 100%. I had one card running at about 70% due to thermal throttling and the other at 100%. This ended up causing dual-GPU performance to barely improve on single-GPU performance with ~2:45 per epoch vs. ~3:10 per epoch. To elaborate as to why the performance gains were negligible, there are two ways in which you can parallelize an ML model over multiple GPUs; data parallelism and model parallelism, the former being more common and simpler to implement. Data parallelism is basically keeping an identical model across devices but having different mini-batches of data on each device to parallelize computing the gradients. Since the snapshots of the model (typically?) need to be synchronized, your training speed is limited by the slowest device.
 - Course changes

- I think in a hybrid/online setting, some sort of friendly cold-calling could improve engagement (I remembering you saying there was some monkey being passed around in your physical course?). Being a bit hypocritical here since I rarely used my camera, but I think being more insistent on camera usage also would help with that. On the other hand, I would like the flexibility of hybrid/online optional (I feel this way about work too).
- I also do think it would've been funner to work in a group, so I do regret going solo. Anyways, all in all I really enjoyed this class, thanks!