Replication of "code2seq: Generating Sequences from Structured Representations of Code" with New dataset (Python)

Soheil Changizi
changizs@myumanitoba.ca
University of Manitoba
Winnipeg, MB, CA

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Finding an approach to gather meaningful features can benefit us in many applications such as code summarization, documentation, retrieval, and creation. Using Code2Seq, we can model the relationship between source code and natural language by extracting natural language sequence from a fragment of source code. Seq2seq models inspired this approach by using an encoder-decoder architecture to generate meaningful sequences from raw source code. This approach encodes a code snippet in its abstract syntax tree (AST) as a series of compositional paths, and it employs attention to pick the relevant paths during decoding [5].

The paper has tested their approach with two applications, including code captioning and code summarization. The dataset that they have used was gathered from top starred GitHub repositories in C# and Java languages. The amount of data they used to get promising results was massive, making this approach ineffective to new programming languages.

In many scenarios, collecting sufficient training data is often expensive and time-consuming, or in some cases, impossible. However, we may be able to apply what we have learned to other programming languages. This approach, known as transfer learning, focuses on transferring knowledge across domains, and it is a promising machine learning methodology for solving the above problem.

Transfer learning has shown promising results in many fields of deep learning, such as computer vision [20], reinforcement learning [18], and sentimental analysis [14]. Using networks trained on millions of existing data, we can leverage this knowledge to speed up the training process for new applications, programming languages, or even updating our previous models.

In this project, a new dataset on a new programing language (Python) has been collected and went through the cleaning process. All the files are from top-stared non-forked GitHub repositories. Methods were taken from files, and their names were used as labels for this assignment. In the last data preprocessing stage, AST paths have been extracted from these methods and stored in 3 separate train, validation, and test files.

This project aims to determine whether we can use the new Python dataset to utilize transfer learning in Code2Seq architecture. First, a model has been trained on the Java dataset from the paper. This model will be used as our pre-trained model to transfer knowledge from Java to Python.

Since we are training on different programming languages, our embedding layers may not contain all the tokens. I have replaced the non-existing tokens and set their weight to the initial value using Xavier's method to mitigate this issue. Then, I have only trained the embedding layers of the pre-trained model on the new Python dataset.

Next, I have done a factorial experiment to see which part of Code2Seq architecture can generalize across programming languages. I have divided the architecture into three parts: embedding layers, encoder, and decoder. Since I have already trained the embedding layers, I ran this experiment twice for encoder and decoder. Every time I have frozen the weights of the two of three parts and trained the rest of the model.

From this experiment, I have concluded that the decoder has more transferable knowledge than the encoder. This is while the embedding layers must be trained due to missing tokens and keywords. Then, in the next step, I finetuned the pre-trained model to train on my Python dataset. I have also trained a model from scratch on the same dataset to compare the precision, recall, and rouge scores.

However, after several epochs, the average loss plateaued, and the model did not improve further. I have tried boosting the model architecture to mitigate this issue by adding an LSTM layer to the decoder. I have transferred all the trained weights from the previous model right at the plateau point. I improved the model's precision, recall, and rouge score by doing this.

In conclusion, I have found that Code2Seq does not work well in the Python language compared to Java and C#. Since in the paper, they have also tested Code2Seq with a dataset (small-java) with the same amount of training samples, it is unlikely that the poor performance results from a small dataset. However, by finetuning and boosting GPT-C models, I improved all the metrics on the test samples. Also, by using a pre-trained model on a large dataset, we can accelerate the training process for other programming languages. The source code for this project have been forked from the original GitHub repository and it is available to the public.

https://github.com/cocolico14/code2seq

2 RELATED WORKS

2.1 Code Embedding

Allamanis et al. have used probabilistic models to do code captioning and retrieval. They have shown their approach on two retrieval tasks: retrieving source code snippets given a natural language query and vice versa is promising [4]. However, they have only tested their approach on a small dataset, and the way they represent features is by using the bag of words, which may not capture the local dependencies in a code snippet efficiently.

Later on, Allamanis et al. have presented an approach for name suggestion for methods and classes based on a neural probabilistic language model. Their model learned semantically similar tokens through embeddings and produced names that had not appeared in the training corpus [1]. However, they have used a log-bilinear model, which needs to search through word embedding representations to predict the next word in the target sequence. Also, using a model as simple as a linear combination of context words may not produce accurate results. As a result, their approach cannot produce quality suggestions promptly.

Iyer et al. have proposed CODE-NN to do code summarization and retrieval. They have used Long Short Term Memory (LSTM) networks with "attention" trained on C# code snippets and SQL queries from StackOverflow [13]. However, they only use token-level information without syntactic paths like Code2Seq, which can degrade the performance in more extended code snippets.

Allamanis et al. also presented an approach based on attention mechanisms for code summarization. However, they have pointed out that previous attentional architectures were not explicitly constructed to learn translation-invariant features locally. To mitigate this issue, they have offered to use convolution on the input tokens to detect such features in a context-dependent way [3]. Nevertheless, the size of convolutional windows is dependant on the context and can not be as informative as syntactic paths.

Bastings et al. presented a practical and straightforward approach to incorporating syntactic structure in machine translation. They have utilized graph-convolutional networks (GCNs) to generate representations of words responsive to their syntactic neighbors by predicting syntactic dependency trees of source sentences [9]. Allamanis et al. have also extended their Convolutional Attention Network by using Gated Graph Neural Networks. Their model now could capture long-range dependencies among variables or functions in a code snippet [2].

Rabinovich et al. presented Abstract syntax networks (ASNs) as an extension of the standard encoder-decoder framework that uses a modular decoder with submodels assembled to generate ASTs natively in a top-down manner. The decoding procedure for every given input uses a dynamically selected mutual recursion between the modules to mimic the recursion's call graph. They have used a decoder model with several submodels, each connected with a particular AST grammar construct and called as needed in the output tree [15]. Their approach can be paired with Code2Seq to seamlessly make the end-to-end learning process. Nonetheless, it will make the overall model too complicated and slow to train.

Alon et al. established code2vec in a prior paper [7], which similarly used AST paths as input for the encoder. Paths, unlike code2seq, are embedded as solid context vectors and are not compositional. As a result, the model overfits to a subset of the paths that have been observed frequently enough during training. When AST pathways are analysed node-by-node with LSTMs, the encoder can generalise even if the paths aren't identical (e.g., a For-node in one path and a While-node in the other).

Brockschmidt et al. proposed a generative code model that directs the generation technique using the semantics of partially created programs. Using graph neural networks, they have augmented partial programs to obtain a graph to construct an exact representation for the partial program. They have demonstrated that using this method; they can construct short but semantically interesting statements from sparse context data [10]. However, more research is needed to evaluate if this approach can be used in more practical contexts like code generation and review.

2.2 Transfer Learning

Deep learning has recently gotten much attention from researchers, and it has been successfully applied to a lot of different fields. However, it does come with some challenges. Deep learning is challenging in fields like robotics that rely on limited environmental feedback rather than precisely classified instances [18]. There are also cases like activity recognition where the only data source is human experts, and creating a large dataset can be cumbersome [11].

However, gathering a large data set is not a challenge in some cases. Instead, finding domain-specific data is difficult or nearly impossible. For instance, millions of images and text corpus are available on the internet, while finding data for a specific domain can be tricky. Using acquired knowledge via transfer learning could be one way to overcome the shortage of domain-specific data. Many researchers use transfer learning in computer vision [20], sentiment analysis [14], and reinforcement learning [18].

Tan et al. define transfer learning and categorize them across four approaches in their survey. Due to costly data collection and annotation, constructing a large-scale, well-annotated dataset is difficult. The hypothesis that the training data must be independent and identically distributed (i.i.d.) with the test data helps us employ transfer learning to overcome the problem of insufficient training data. The four categories suggested by Tan et al. are Instances-based, Mapping-based, Network-based, and Adversarial-based [17]. For this project, I have followed the Network-based approach in which it reuses the partial of network pre-trained in the source domain. However, we will also discuss Adversarial-based later in this section to propose an alternative approach. A more recent survey by Zhuang et al. connects and systematizes the existing transfer learning research and summarizes the mechanisms and strategies

of transfer learning comprehensively. The main focus of this survey is on homogeneous transfer learning. Homogeneous transfer learning approaches are developed and proposed to handle situations where the domains are of the same feature space. It also suggests strategies to avoid the negative transfer, which is when the target learner is negatively affected by the transferred knowledge [23].

Yao and Doretto propose one possible solution to address the problem of negative transfer. A boosting framework for inductive transfer learning when knowledge from multiple sources can help because the chance to import knowledge from a source related to the target increases significantly [21]. I have partially used their approach by boosting a trained model with new LSTM layers in the decoder.

2.3 Recent Works

Adversarial learning methods are a potential way to build robust deep networks that can generate complicated samples in various domains. They can also help recognize when the domain shifts and dataset bias. Tzeng et al. proposed Adversarial Discriminative Domain Adaptation (ADDA), a framework that combines discriminative modeling, untied weight sharing, and a GAN loss. Their discriminative modeling can handle significant domain shifts while exploiting a GAN-base loss [19]. The most recent work of Yefet et al. studies the adversarial examples with slight mutation to the input code snippet of models such as code2vec and GNNs, which results in the prediction of choice from these models. They present Discrete Adversarial Manipulation of Programs, a new approach (DAMP). DAMP derives the desired prediction from the model's inputs while keeping the model weights constant and using gradients to change the input code slightly [22].

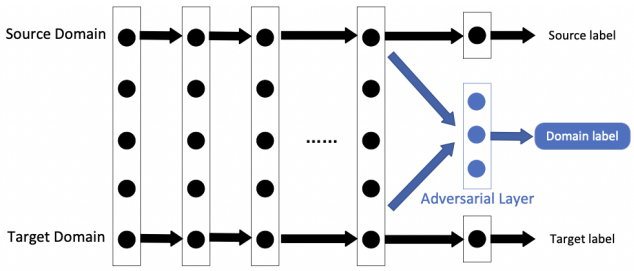

Figure 1: Adversarial-based deep transfer learning. The performance of the adversarial layer will affect the transfer network to find features with more transferability [17].

David and colleagues propose a new method for predicting procedure names in stripped executables. Static analysis and neural models are used in their approach. They employ static analysis to acquire augmented representations of call sites, then use the control-flow graph (CFG) to encode the structure of these call sites, and finally, while attending to these call sites, construct a target name [12].

Table 1: Keywords used in repository search and number of python files downloaded

Keyword	Count	Keyword	Count
Machine Learning	5k	Algorithm	18k
Distributed Computing	6k	Python	80k
Blockchain	4k	API	9k
Web Application	7k	Data	17k
Image Processing	6k	Processing	13k
Genomics	3k	Internet	7k
Security	16k	System	14k
GUI	8k	Artificial	13k

Alon et al. recent work is on generating the missing code within a more extensive code snippet. This is similar to code completion, but instead of predicting a single token at a time, it can predict complex expressions. It also utilizes ASTs by decomposing it into a product of conditional probabilities over its nodes and learning these conditional probabilities[6].

3 DATA COLLECTION

SRILab collected a dataset of parsed Python ASTs and used it to train and evaluate their DeepSyn tool project. The Python code snippets were gathered from GitHub repositories by deleting duplicate files, project forks (copies of other existing repositories), and removing obfuscated files keeping only ASTs with at most 30'000 nodes in them. They used the Python AST parser available in Python 2.7 for parsing. Their dataset is divided into training files (100,000) and testing files (50,000) [16].

I followed the same steps to expand their dataset and gather more Python files. Using GitHub Crawler, I have downloaded top-stared non-forked Python repositories. Repositories have been selected from various fields, as shown in Table 1. Since there were some overlapping projects across these keywords, I removed all the duplicated repositories.

3.1 Data Cleaning

After extracting all python files, I have removed all the empty files and files with sizes less than 1KB. Also, some files did not have any significant code, like configuration files. These files roughly consisted of 26k of python files that I have gathered. To find all the duplicated files, I calculated the md5 hash to spot similarities between them. This step resulted in removing around 106k files.

Next, I have run a script to pass all the python files from the 2to3 tool. Using this tool allowed me to convert them to Python3 and normalize the code convention. Moreover, I could spot all the files with compilation errors and remove them in this process, approximately 3k files in total.

As I mentioned before, SRILab also gathered a dataset of ASTs from Python code snippets. They have included their code in which extracted ASTs from raw source code. I have used the same AST parser to extract ASTs from python files that I have collected from GitHub. Like their approach, I only kept trees with less than 30000 nodes. As a side note, at first, due to memory limitations, I only kept trees that had between 70 and 7000 nodes. However, I was able to fix the memory leakage later on.

3.2 Data Preprocessing

In this step, I have pulled all the tree paths with less than eight nodes in them to generate training samples. The length of paths can be altered based on the scope of block statements in the code. To capture deep relations between statements, we need paths with longer lengths. However, this requires us to use more LSTM layers in the decoder to pull this off.

Besides extracting them, the necessary tokens are also added to the paths. These tokens include start and end of statement tokens and tokens for padding. Each sample includes six nodes along the path, two terminal nodes, and the target token. Since the task is code summarization for this project, the target is function names.

3.3 Result

In the end, by merging ASTs from what I have gathered and the SRILab 150k Pyhton dataset, I got 200k files in total. I have used 132k file for training and the rest for testing. After path extraction, I ended up with 1023848 training samples, 256321 samples for validation, and 265337 samples for test. The size of my dataset is roughly a third of the java-med dataset from the paper.

4 METHODOLOGY

As mentioned before, this approach utilizes the AST of source code snippet in a given language. Terminals are the tree's leaves, and they usually relate to user-defined values that represent code identifiers and names. Nonterminals are non-leaf nodes representing a limited number of language structures, such as loops, expressions, and variable declarations.

Given the AST of a code snippet, all pairwise paths between terminals are considered and represented as sequences of terminal and nonterminal nodes. Since a code snippet can contain an arbitrary number of such paths, several paths have been sampled as the representation of the code snippet. In every training iteration, several new paths are sampled afresh to avoid bias by providing regularization, which has improved the testing results. An example of AST of a code snippet is illustrated in 2.

The model is based on GPT-2 with modifications to take AST paths as input. The encoder does not interpret the input as a flat sequence of tokens. Instead, the encoder creates a vector representation for each AST path separately, and the decoder uses the averaged encoded feature vectors to generate the target sequence.

```
int countOccurrences(String str, char ch) {
   int num = 0;
   int index = -1;
   do {
      index = str.indexOf(ch, index + 1);
      if (index >= 0) {
         num++;
      }
   } while (index >= 0);
   return num;
}
```

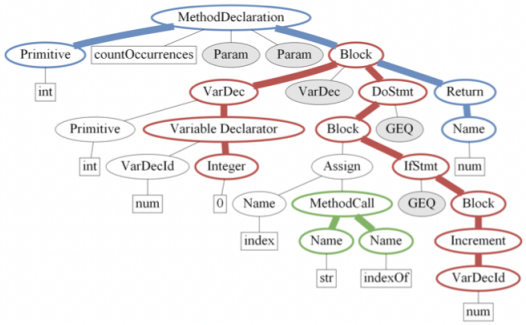

Figure 2: An example of AST for a Java method, different paths highlighteing the terminal nodes and statements in between.

Given a set of AST paths, the goal is to create a vector representation to feed our model. Each path consists of several nodes and two terminal tokens. The way we will map nodes and terminals will be slightly different. However, like GPT-2, we will use contextual embedding for both of them to convert words into feature vectors.

As for the path representation, we will pass the words through an embedding matrix. The weights for this matrix will be updated during the training process. Then the entire sequence will be encoded using a BiLSTM. In the end, the final states of the forward and backward pass will be concatenated. As for the token representation, we split the tokens into subtokens by leveraging the snake case convention in Python code. These subtoken will be passed through a learned embedding matrix and get summed up to be later concatenated with path representation.

This combined representation will be for a single path. Several paths will be encoded in parallel at each training step and pass through a dense layer for the decoding step. These paths will be sampled randomly, resulting in reduced variance and a more stable learning process.

The order of the random pathways is ignored, unlike in traditional encoder-decoder models. Each path is encoded separately to initialize the decoder's state, and the combined representations are aggregated by mean pooling. Finally, the decoder constructs the output sequence while attending to all combined representations

Table 2: code2seq model outperforming previous PL-oriented and NMT models [5]

Model	Java-small			Java-med			Java-large		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
Paths+CRFs [8]	8.39	5.63	6.74	32.56	20.37	25.06	32.56	20.37	25.06
code2vec [7]	18.51	18.74	18.62	38.12	28.31	32.49	48.15	38.40	42.73
ConvAttention [3]	50.25	24.62	33.05	60.82	26.75	37.16	60.71	27.60	37.95
2-layer BiLSTM	42.63	29.97	35.20	55.15	41.75	47.52	63.53	48.77	55.18
code2seq	50.64	37.40	43.02	61.24	47.07	53.23	64.03	55.02	59.19
Absolute gain over BiLSTM	+8.01	+7.43	+7.82	+6.09	+5.32	+5.71	+0.50	+6.25	+4.01

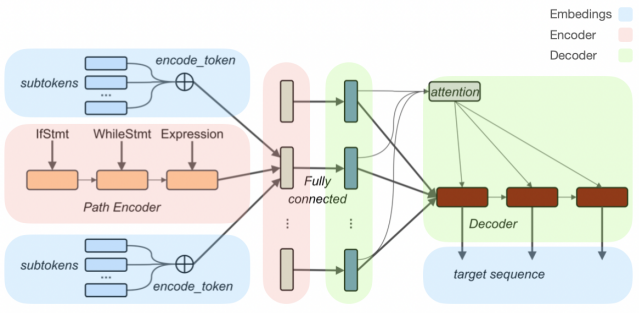

Figure 3: Each AST path is encoded as a vector in the GPT-C model, and the average of all k paths is used as the decoder's initial state. While attending to the k encoded paths, the decoder generates an output sequence.

by selecting the distribution over these k combined representations using an attention mechanism.

To investigate the impact of techniques used in Code2Seq, they have done an ablation study. They have tested the variation of their model on the java-med dataset by removing certain aspects. Their first experiment was only to encode the terminal nodes and ignore the rest from the paths. The outcome was still better than the baseline, and they claim that more LSTM layers can mitigate this degradation in performance.

Removing the decoder has lost more than one-third of the overall performance. Despite the methods' brief names, this reveals that the method name prediction task should be approached as a sequential prediction, not with a single softmax layer. They also found that not splitting tokens and not having tokens reduces the score significantly, demonstrating the need to record both subtokens and syntactic routes.

Finally, the no random experiment indicates that choosing k different paths anew on each training iteration, rather than using the same sample of paths from each example throughout the training, adds data-level regularisation, improving the model's performance.

My contribution to this project is to try this approach on a new programming language. I have collected a dataset on ASTs extracted from Python code snippets and trained the model. However, I could not achieve the same performance for the Python language. To get better performance, I tried to transfer already learned knowledge on a large corpus of data to gain an advantage for training my new model. I studied different parts of the architecture to find out how to finetune the model for transfer learning. Moreover, by boosting the architecture, I achieved better performance. In the next section, I will answer three research questions that I have studied throughout this project and share my results.

5 RESEARCH QUESTIONS

5.1 Which part of this architecture can be generalized across languages?

To answer this question, I will first define the notion of transferability. By observing the rate of accuracy improvement of a pre-trained model on a new dataset based on a group of trainable variables, we can decide the transferability. I have separated the GPT-C architecture into embedding, encoder, and decoder sections and ran an experiment to check the transferability of each of these sections.

The pre-trained model published by the paper's authors was not trainable, so I had to train the model on the java-med dataset from scratch. I did not choose the java-large dataset (nearly 16M training samples) because the memory provided by colab was not enough. I have trained the model for 15 epochs (as shown in figure 4) which took three days to train, and saved the model for my experiment. Due to differences in keywords and statements between Java and Python languages, some keywords were missing from the embedding matrices. So, I have replaced the missing words and reinitialized the weights based on Xavier's method. Then, I retrained the embedding matrices for five epochs and saved this model as my initial pre-trained model for the rest of the experiment. Looking at the accuracy trend during training, it was clear that embedding layers were transferable to some extend. After epoch three, the accuracy plateaus and does not improve any further, which means knowledge from the Java dataset is not enough on its own.

Next, I froze all the trainable variables related to the embedding layers and decoder to test the transferability of the encoder. There is a steady increase in accuracy this time, unlike the embedding

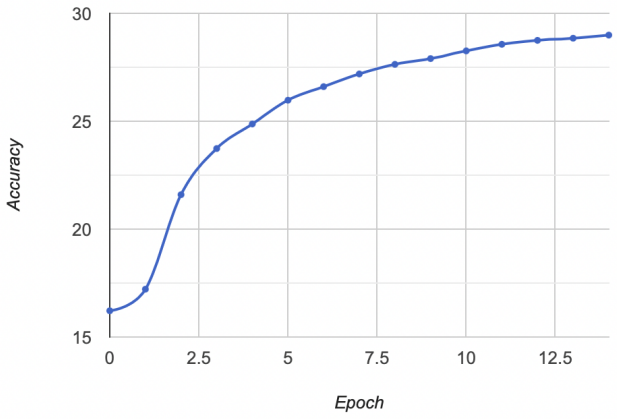

Figure 4: The accuracy of code2seq model trained on java-med data set which later on used for the transferability experiment.

layer. I have repeated this test for the decoder, which resulted in slightly less accuracy than the previous experiment.

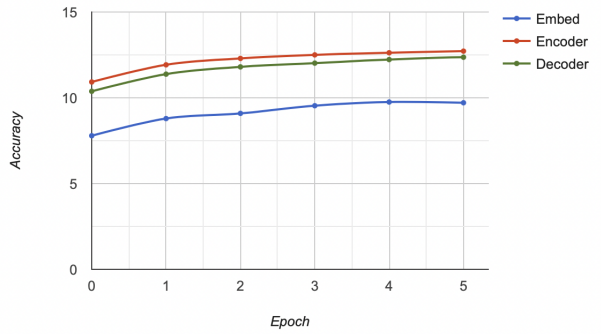

Figure 5: Comparison of accuracy of transferability experiment on three different sections of GPT-C architecture. The embedding section used the java-med pre-trained model while the other two used the trained embedding model after five epochs.

In conclusion, the decoder has the most transferability between decoder and encoder. As shown in Figure 5, the model where every weight but encoder weights are frozen achieved higher accuracy. This means that decoder weights transfer more helpful knowledge from the source domain (Java-Med) to the target domain (Python199k). However, due to the slight value difference (0.093%), one can argue that this is noise, and both sections have the same transferability.

Table 3: Comparison of accuracy, recall, precision, and rouge metrics between three models. They are all tested on Python199k test set

Model	Acc	Rec	Prec	Rouge-2	Rouge-l
Python150k	14.243%	24.988%	36.082%	4.111%	30.702%
Python199k	14.209%	23.848%	42.654%	4.128%	32.922%
Fine Tuned	15.535%	26.484%	42.766%	5.832%	35.229%

5.2 Can this architecture help transfer learning?

The main paper tested their model on C and Java, but SRILab already provided an AST dataset on Python. The authors have tested this dataset on their model in their GitHub repository. However, the performance is not as good as Java or C# counterparts for code summarization.

Since they have made their java-med dataset which consists of 4M training samples, one can leverage the existing data to train a model for a new programming language. I used Network-based deep transfer learning for this project, which reuses the pre-trained partial network in the source domain (Java-med), including its network structure and connection parameters, and transfers it to a part of the model used in the target domain (Python199k).

As for the baseline, I have trained on both SRILab's Python150k dataset and my Python199k dataset. However, I have trained from scratch in both of these baseline models for 20 epochs and did not use pre-trained models. The validation set for Python150k and Python199k will be slightly different (Python199k has more validation samples), but I have tested all three with the same testing samples.

After discovering the transferability of each section in GPT-C architecture from the previous experiment, I came up with this fine-tuning configuration (trained on Python199k). I have set the learning rate of embedding layers to 0.001, encoder to 0.005, and decoder to 0.01. I have trained this network using the pre-trained model (5 epochs on the embedding network with java weights) for 20 epochs.

In conclusion, by transferring knowledge from a pre-trained model on the Java-med dataset, we achieved better performance in all metrics on the test set. This method improved not only the performance of our model but also sped up the convergence time (in terms of the number of epochs to reach a decent accuracy) compared to training from scratch.

5.3 Can we improve the accuracy by changing the architecture?

As shown in figure 6, all performance metrics start to plateau and do not improve. The reason behind this might because of negative transfer. It can happen for various reasons, including the relevance of the source and target domains or the learners' ability to locate transferable and valuable information across domains.

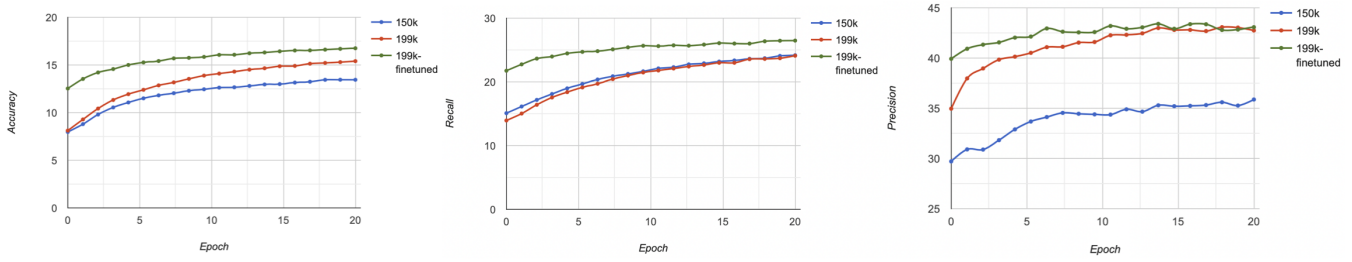

Figure 6: Comparison of accuracy, precision, and recall between three models. Validation set is slightly different between Python150k and Python 199k datasets.

According to Yao and Doretto, getting knowledge from numerous sources will raise the chances of identifying one source close to the target and reduce negative transfer. They have also proposed that boosting methods such as AdaBoost can also reduce the impact of negative transfer. Due to the lack of resources, I can only add more layers to boost my model and continue the training to overcome this issue.

At first, I wanted to use stacked BiLSTM as an encoder and add more LSTM layers to the decoder. However, the GPU memory could not handle stacked BiLSTM, so I ended up only adding more layers to the decoder. I have continued training from epoch 27 while using this new architecture.

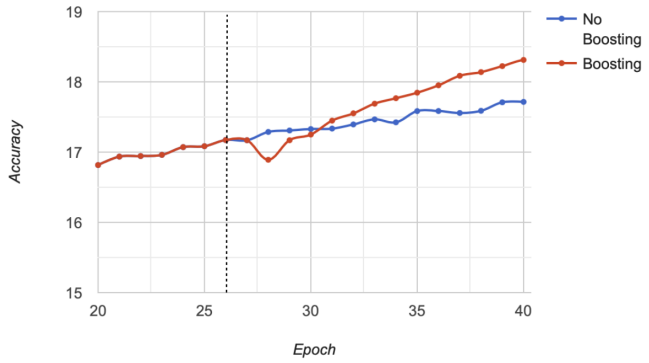

Figure 7: Comparison of accuracy boosting and not boosting. The boosting happens right at epoch 27.

I have also continued training with the previous model until it reached 40 epochs to compare these two approaches. As shown in figure 7, the accuracy decreases for two epochs but instantly gains momentum and improves over the rest of the training epochs. However, the performance gain in the test set is not that significant.

6 THREATS TO VALIDITY & LIMITATIONS

Unlike the result reported in the paper, this approach did poorly on Python language. There are several possible reasons for the unsatisfactory performance. First, we will discuss the probable reasons,

Table 4: Comparison of accuracy, recall, precision, and rouge metrics between boosting and no boosting. They are all tested on Python199k test set

Model	Acc	Rec	Prec	Rouge-2	Rouge-l
No Boosting	16.423%	27.605%	42.430%	5.832%	35.229%
Boosting	16.793%	27.651%	43.618%	6.074%	36.074%

and then we will state the limitation of my approach and this model.

First of all, this project was only tested on a limited Python dataset. While there are nearly 1M training samples in our collected dataset, it might not be enough. We can compare our dataset with Java-small since they have roughly the same training samples. Code2Seq on java-small achieved 50.64% precision, 37.40% recall, and an F1 score of 43.02%. This is while our best model achieved 43.62% precision, 27.65% recall, and an F1 score of 33.85%.

Another possible reason could be the quality and variety of the collected database. Java-small was collected from only 11 relatively large Java projects. In contrast, my database was collected over a variety of keyword searches. This is when larger datasets such as java-med or java-large are considering variety after gathering over 1000-9500 repositories.

Lastly, it could be that this approach needs extensive hyperparameter tuning. I have used the same configuration for Java training. The main reason for not changing the configuration was to transfer the pre-trained weights seamlessly. This is also one of the first limitations of using transfer learning. To transfer pre-trained weights from a source domain to a target domain, we must ensure that both source and target architectures are compatible.

Most of the time same configurations do not apply to both models. For instance, based on the size of the dataset, it might not be feasible to use large batch sizes. Changing the batch size means that some weights must be discarded, which means losing transferable knowledge.

Another limitation is the issue of negative transfer since the source domain, and target domain might not be in the same feature space.

As mentioned in the related work section, there are new ways to mitigate this issue. However, they need more computational power and complicate the training process.

7 CONCLUSION

In conclusion, I discovered that Code2Seq does not perform as well in Python as in Java and C#. It is doubtful that the poor performance is due to the short dataset because the publication also evaluated Code2Seq using a dataset (small-java) with the same number of training samples. However, as mentioned in the previous section, the dataset's quality might be inferior, or the models need more hyperparameter tuning.

Moreover, by finetuning GPT-C models, I improved all the metrics on the test samples. Also, by using a pre-trained model on a large dataset, I have accelerated the training process for my dataset. This also shows the transferability of encoder and decoder variables. To mitigate the negative transfer issue, I boosted the model's architecture and improved all metrics.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740 http://arxiv.org/abs/1711.00740

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. PMLR, 2091–2100.

[4] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 2123–2132. https://proceedings.mlr.press/v37/allamanis15.html

[5] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. *CoRR* abs/1808.01400 (2018). arXiv:1808.01400 http://arxiv.org/abs/1808.01400

[6] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2019. Structural Language Models for Any-Code Generation. *CoRR* abs/1910.00577 (2019). arXiv:1910.00577 http://arxiv.org/abs/1910.00577

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. *CoRR* abs/1803.09473 (2018). arXiv:1803.09473 http://arxiv.org/abs/1803.09473

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. *SIGPLAN Not.* 53, 4 (jun 2018), 404–419. https://doi.org/10.1145/3296979.3192412

[9] Jasmijn Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Sima'an. 2017. Graph convolutional encoders for syntax-aware neural machine translation. *arXiv preprint arXiv:1704.04675* (2017).

[10] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2018. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490* (2018).

[11] Diane Cook, Kyle D Feuz, and Narayanan C Krishnan. 2013. Transfer learning for activity recognition: A survey. *Knowledge and information systems* 36, 3 (2013), 537–556.

[12] Yaniv David, Uri Alon, and Eran Yahav. 2019. Neural Reverse Engineering of Stripped Binaries. *CoRR* abs/1902.09122 (2019). arXiv:1902.09122 http://arxiv.org/abs/1902.09122

[13] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[14] Ruijun Liu, Yuqian Shi, Changjiang Ji, and Ming Jia. 2019. A survey of sentiment analysis based on transfer learning. *IEEE Access* 7 (2019), 85401–85412.

[15] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[16] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (oct 2016), 731–747. https://doi.org/10.1145/3022671.2984041

[17] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A survey on deep transfer learning. In *International conference on artificial neural networks*. Springer, 270–279.

[18] Matthew E Taylor and Peter Stone. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10, 7 (2009).

[19] Eric Tzeng, Judy Hoffman, Kate Saenko, and Trevor Darrell. 2017. Adversarial discriminative domain adaptation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7167–7176.

[20] Mei Wang and Weihong Deng. 2018. Deep visual domain adaptation: A survey. *Neurocomputing* 312 (2018), 135–153.

[21] Yi Yao and Gianfranco Doretto. 2010. Boosting for transfer learning with multiple sources. In *2010 IEEE computer society conference on computer vision and pattern recognition*. IEEE, 1855–1862.

[22] Noam Yefet, Uri Alon, and Eran Yahav. 2019. Adversarial Examples for Models of Code. *CoRR* abs/1910.07517 (2019). arXiv:1910.07517 http://arxiv.org/abs/1910.07517

[23] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.