CS230

Setting a Benchmark for Representation Learning of Source Code

Weichen Zhao
Arista Networks
Santa Clara
weichenz@stanford.edu

Kaviarasan Selvam
Department of Computer Science
Stanford University
kselvam@stanford.edu

Abstract

Representation Learning for source code is a relatively new field with very little formalization. This project aims to set a benchmark for representation learning of source code through two parts. First, we looked into reproducibility in this sub-field of Deep Learning by reproducing state-of-the-art papers for various tasks. Second, we delved deeper into setting a benchmark for the variable naming task, using the concept from Allamnis et al.[6] to learn program representations for Python source code, we implement the variable misuse task.

1 Introduction

The problem of Representation Learning of source code pertains to how we can represent source code to facilitate tasks like variable naming and variable misuse in source code. Today, very little has been done to formalize such research. Very much like the ImageNet paper for the Computer Vision community, this project aims to lay the foundations for setting up a rigorous benchmark and an industry standard for various tasks that use Deep Learning for source code. This project could have long-term benefits for Integrated Development Environments (IDE). The technology of using representation learning for source code, when reached a mature stage, will enable the development of robust recommender systems for IDEs. The recommender systems will be vital for tasks like suggesting better naming conventions for variables and identifying misused variables, which will ultimately lead to more bug-resistant code.

Since there is some state-of-the-art work being done in this sub-field of Deep Learning, it is important to compile them and evaluate their reproducibility. This will not only set the industry standard to beat, but also hold authors accountable if the claims in their papers are not reproducible. Besides pooling together the most up-to-date research work, this project also aims to explore some of the challenges of setting a benchmark for a task. We decided to delve deeper into the task of variable misuse (VarMisuse), which checks if we are using the correct variable at a position. We adopted the method from a state-of-the-art paper that carried out VarMisuse on C# source code, and tried to implement our Deep Learning model to carry out the same task on Python source code. We decided that this project will be important to illustrate how such models can be standardized to carry out VarMisuse irrespective of the programming language.

Since this project has 2 main parts, reproducing state-of-the-art papers and modifying one of them to take Python context graphs, the Related Work section will cover all the work that was done on reproducing papers, and the rest of the report will cover the work that was done in implementing Python version of the Microsoft paper[6] for the VarMisuse task.

CS230: Deep Learning, Winter 2019, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related work

Our work compiles a list of state-of-the-art papers that has the potential to be used as industry standards for their respective tasks. We started off with compiling a list of papers pertaining to Machine Learning for source code. This list of papers that were compiled can be found in the Filtered Paper tab here: [10].

We then evaluated the feasibility of reproducing the papers by identifying the availability of the data set and the source code for each paper. Since we were only interested in evaluating approaches that used Deep Learning, we also filtered out papers that used non-Deep Learning approaches like n-grams. We skimmed through all the papers to understand their approach and evaluate their reproducibility. From over 100 papers in the field, we only managed to identify 7 papers which had a chance of being reproduced easily using a Deep Learning approach. For these 5 papers, we made another spreadsheet to classify the papers by the different tasks they were trying to accomplish. The tasks are usually done on the representation of source code produced from training a Deep Learning model. Identifying the tasks that each of these papers were trying to accomplish was important because we would have to set performance benchmarks based on tasks. This list can be found in the Classified By Task tab here: [10].

After identifying the papers, we tried to reproduce them to verify their achievement claims before calling it a benchmark to surpass. Surprisingly, some of these papers are really difficult to reproduce. The Context2Name[1] paper, which attempted to use Recurrent Neural Networks (RNN) to predict natural identifier names for variables in minified code, had a very heavy-weight network that was still training after 2 weeks. After verifying that the model saved checkpoints, we stopped the training to run the evaluation step, only to be disappointed by the code getting stuck in an infinite loop.

The code2seq paper[2] attempted to predict function names and function summaries. The paper used a LSTM-based autoencoder model to encode ASTs and decode them to produce a sequence of tokens representing the the function names and summaries. This model, which was also big, produced big tensors that were too big for the p2.xlarge AWS GPU instance that we were granted. Due to the limitation on the AWS credit, we tried to request a for a bigger instance from AWS when we were sure that we had some balance to exhaust, but were not given the correct one in time to train the network. This hardware limitation also made us realize the importance of specifying the hardware requirements when publishing a paper and when setting a benchmark. If the authors of the papers used the most advanced GPUs to carry out their experiments, it would be unreasonable for people with limited resources to attempt reproducing the paper. In addition, a paper that has very high hardware requirements might not be suitable as a benchmark for people to measure their performance.

For the rest of the three paper, we were able to reproduce two of them within reasonable amount of accuracy. We got 72.9% semantic distance score compared to 79.12% mentioned in the Neural code comprehension[3] paper, and 77.6% for classification accuracy compare to 81.1% mentioned in the Type inference[4] paper. We didn't have enough time to reproduce the loop invariants paper[5] before midterm.

3 Methods

The core idea of our model comes from Allamanis et al.[6], which learns program representations as graphs. As mentioned in the beginning of their paper, most recent works on learning code representation have tried to transfer natural language methods and does not capitalize on the unique opportunities offered by code's known semantics.

To model the variable usage context, we used Gated Graph Neural Network (GGNN), which consists of Gated Recurrent Units (GRU), to process graph based AST node relationship information. In order to leverage the semantics, we build our own Python ASTs parser to extract features and relationships, including long distance ones, between terminals and non-terminals. For each minibatch, GGNN layer will generate embeddings for each node in the AST tree with size of 64. We then select the embeddings of variable nodes and feed them to task specific layer. For variable misuse task, we have two fully connected layer as well as a softmax layer to generate a multi-class single-label variable choice prediction. The output is a 2-D array of shape [variable count, variable vocabulary size]. The correct variable choices is represented as the class id (vocabulary id) per variable. Therefore, we choose the softmax cross entropy loss function to optimize. Our model is available at Github[13].

```
if a < b:
    return 1
else:
    return 2
```

Figure 1: A code snippet

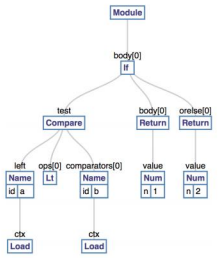

Figure 2: Corresponding AST

4 Dataset and Features

In the original paper, the data set used was C# source code[8]. The ASTs for the source code were converted into context graphs with a number of custom edges like LastLexicalUse, GuardedBy, GuardedByNegation and VariableUsageContexts. The construction of some of these edges were explained in the Programs as Graphs section in the paper. For example, for the source code shown in Figure 1 with the corresponding AST shown in Figure 2, the data set will contain a GuardedBy edge from the Name node that has id: a to the the Compare node. After the graph was created, some nodes were selectively removed from the context graph so that the model could be trained to predict the missing node labels. These graphs with 'HOLES' were then further processed to produce a pickled file to fed to the training pipeline.

To extend this method to Python, we tackled many challenges to produce a similar context graph from Python ASTs. We downloaded roughly 4 GB python source code from 38 Github repositories, containing 20855 files. For pre-processing, we implemented a Python AST parser[12] to construct a context graph with the minimum edge types that were needed to train the model: The Child, NextToken, GuardedBy and VariableUsageContext edges. The context graphs are then processed further to produce two vocabularies of all tokens and variables from source code. The Python AST parser generated 3.6GB training data, containing 2.17 million variable context info, as well as many other AST node relationship edges. Similar to the paper we referenced, the vocabulary id based data is then converted to pickled and tensorized versions of the graphs, which were fed as input to our model. As for the data allocation, we separated them into training and testing set in 9:1 ratio.

5 Experiments results and discussion

We found that reproducing papers is a really hard task, which further supports the need to have consistent reproducible benchmarks. From a hardware perspective, some networks are very big and are trained on powerful GPUs, making it harder to reproduce when one does not have similar computing power. For example, in the Context2Name paper, the authors trained the model on a 1080 Ti GPU, which was not available to us on AWS. Therefore, the model trained for almost 2 weeks on the p2.xlarge instance that was available. The code2seq paper is also another good example of how reproducibility efforts can be hampered by computing resources. When trained on a p2.xlarge instance, the model training consistently stopped with a Out Of Memory (OOM) error when trying to produce a very large tensor.

From a software perspective, we found many instances of code that just didn't work when the repository is cloned from GitHub. Some papers like Context2Name didn't specify requirements or have a requirements.txt file to indicate all the libraries that had to be downloaded. The Context2Name repository also had code that couldn't find the right files because of a lack of knowledge of the file path. Scenarios like this can occur when authors use GitHub merely as file storage and don't use the same file structure when training a Deep Learning system.

When trying to reproduce and reuse the model given as part of the Allamanis paper[6], we faced different issues. With this paper, we tried to reproduce the results with the sample C# data set that was given to make sure the the whole pipeline works. When using the sample data set with 15 C# ASTs, the code in the GitHub repository for the graph2seq model did not work. After some diagnosis, we figured out that the sample data set had too few samples to fit a 3000-word vocabulary that was hard-coded. When using the full C# data set that was referenced in the paper, we found that the

context graphs in the full data set had some different edges than the context graphs in the sample data set. The tensorizing step for the full dataset failed repeatedly because it couldn't find edges like SymbolLabels and VariableUsageContexts. Furthermore, the full data set also had new edges like slotTokenIdx and SlotDummyNode that were not explained anywhere on the data set website or in the paper. This discrepancy between the sample and full data set posed an issue when trying to verify the claims made by the paper. When we tried to train the model with the sample data set, we also found a method that had a missing parameter that had to be figured out.

In order to evaluate the possibility of establishing a full pipeline, we stuck with using the small sample data set and tuned the vocabulary size and acceptance threshold hyperparameters. The vocabulary size was trimmed from 3000 to 30 so that when generating a sequence to predict a variable name, the vocabulary indices with the highest probability will come from a number between 1 and 30. We also reduced the acceptance threshold, the number of times a token has to appear in a source to be accepted into the vocabulary, from 5 to 1. Since the sample data set is small, the number of times that each token appears is also small, leading to almost no tokens being accepted into the vocabulary.

With this change in hyperparameters, we were able to get the model to train to completion. However, the model had a very high training loss of 5.12. This bad performance was further reflected in the testing phase that produced an accuracy score of 46.7%.Since the model had a high bias, our first inclination was to increase the complexity of the network by increasing the number of layers. We increased the number of hidden layers in the decoder from 2 to 4. Unfortunately, increasing the number of hidden layers did not improve the performance of the network. There is a possibility that the network was not learning anything from the small sample data set size of 15 C# ASTs. While we established that the sample data set has the right format of input for the model, we need to dedicate more work get a reasonably sized data set with the desired format so that we can confidently evaluate the bias and variance in the model.

For variable misuse task, we run our model on AWS p2 instance. We tuned different hyperparameters to get better results. The initial turnings include increase default learning rate to overcome the effect of learning rate decay; increasing the number of fully connected layer as well as GGNN layer to generate better embeddings and better fit the node representation to variable choice. Apart from the tuning experience, we also figured our some implementation issue we came across. After we fixed the learning rate problem, the training and validation accuracy kept increase to almost 100%, which we have never seen in any literature. After a long time of tuning our model to avoid over fitting, we took a close look at the GGNN layer we adopt from the Allamanis et. al paper[6]. As it turn out, the model apply back propagation and update embeddings for both training and validation process. For the results from our model, we achieved 84.29% training accuracy as well as 87.63% testing accuracy with 4 layers of GGNN network with embedding size 64.

6 Conclusion/Future Work

This project exposed some of the difficulties of setting a benchmark for the relatively immature field of representation learning for source code. We found many papers written about this subject matter, but only few seemed reproducible, and even those had different issues that deterred our efforts to reproduced the results claimed in the paper.

For our current model of downstream task, we have only selected embeddings for variable nodes, which limits us to use fully connected layer to produce variable choice prediction. Given we also have other info about a particular node, we could also extract the embeddings and use those to improve the prediction accuracy. For example, the variable context info (adjacent token) can be feed to CNN/RNN to produce more reference for the last fully connected layer.

This project is a step in the right direction in setting up the benchmark. However, there is still a lot of work to be done. For a start, we should get in touch with the authors of the shortlisted papers to discuss the issues that we faced and how we can solve those issues. We can some up with suggestions for them to make changes to their GitHub repositories so that the Deep Learning community can benefit from them. For the Allamanis paper, we can also request for a much more reliable data set to train their model and verify their claims. Regardless of the performance claims that the papers make, we should make sure that the results are reproducible before they can be included in the benchmark. We should also make sure that we document all the necessary libraries and hardware that will be needed to reproduce each paper. The end goal for this initiative would be to have a website

with benchmark papers for each task, and links to the repositories and corresponding data sets. If possible, we should try to come up with one or two data sets that can be used for a multitude of tasks. By starting this initiative to set up a benchmark, we hope that the Deep Learning for source code community will pick up where we left off and work on setting industry standards for this exciting field

7 Contributions

We would like to thank two people who have mentored us throughout this project. Michele Catasta, a postdoctoral researcher in the Stanford SNAP group who is very passionate about Deep Learning for source code, came up with the original idea to set up this benchmark for the research community and the industry. He reached out to us and got us onboarded with his vision very quickly as he had a clear plan on how to execute his initiative. Michele also got into contact with the Miltiadis Allamanis to get the source code for the Learning to Represent Programs with Graphs paper.

We would also like to thank Dylan Bourgeois, a Masters thesis student also in the Stanford SNAP group, for his incredible support throughout this project. He assisted us in debugging code and introduced us to many different tools to visualize and debug graph neural networks (GNN). He was very knowledgeable about Representation Learning on source code and gave us code that we could use to produce and parse python ASTs (add citation).

References

[1] Bavishi, Rohan, Michael Pradel, and Koushik Sen. *Context2Name: A deep learning-based approach to infer natural variable names from usage contexts*. arXiv preprint arXiv:1809.05193 (2018).

[2] Alon, Uri, Omer Levy, and Eran Yahav. *code2seq: Generating sequences from structured representations of code*. arXiv preprint arXiv:1808.01400 (2018).

[3] Ben-Nun, Tal, Alice Shoshana Jakobovits, and Torsten Hoefler. *Neural Code Comprehension: A Learnable Representation of Code Semantics*. In Proceedings of the Conference on Neural Information Processing Systems (NIPS), 2018.

[4] Hellendoorn, Vincent J., et al. *Deep learning type inference*. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2018.

[5] Si, Xujie, et al. *Learning loop invariants for program verification*. In Proceedings of the Conference on Neural Information Processing Systems (NIPS), 2018.

[6] Allamanis, Miltiadis, Marc Brockschmidt, and Mahmoud Khademi. *Learning to represent programs with graphs*. In International Conference on Learning Representations (ICLR), 2018.

[7] Raychev, Veselin, Pavol Bielik, and Martin Vechev. *Probabilistic model for code with decision trees* ACM SIGPLAN Notices. Vol. 51. No. 10. ACM, 2016.

[8] Microsoft Research *https://www.microsoft.com/en-us/download/details.aspx?id=56844*

[9] Python AST Visualizer *https://vpyast.appspot.com/*

[10] Google Doc for Filtered Papers *https://docs.google.com/spreadsheets/d/1ZblWUAMgh1T80ElDZtawC-khBmONbOTYEqTP06qID3M*

[11] Python AST generator *https://github.com/dtsbourg/codegraph-fmt*

[12] Python AST parser *https://github.com/09zwcbupt/AST2Graph*

[13] Variable misuse model *https://github.com/09zwcbupt/CS230-VarMisuse*