BERT for source code

Andrei Gusev
Higher School of Economics
Saint Petersburg
Email: andy.gusev@list.ru

Dmitrii Orekhov
Saint Petersburg National Research University of Information Technologies, Mechanics and Optics
Saint Petersburg
Email: d.i.orekhov@gmail.com

Maksim Vinnichenko
Higher School of Economics
Saint Petersburg
Email: max.kvant@gmail.com

***Abstract*—Machine learning on source code is an actively developing field of research. Programmers would highly benefit from machine learning empowered tools capable of program repair, smart autocompletion, semantic code search and many other things. As in many machine learning areas the quality of data representation is crucial to the performance of trained models. Following the success of BERT in NLP a number of works took an attempt at adapting unsupervised pretraining on sequential data to the domain of source code. CuBERT and CodeBERT showed decent results in a number of different tasks. In the current work we further explore capabilities of CuBERT by applying it to several previously unexplored fine-tuning tasks: move method refactoring and method name prediction. In move method refactoring we explore aggregation techniques applied to general purpose embeddings obtained during the masked language model task and obtain promising results which need further verification. In method name prediction we fine tune sequence-to-sequence model initialized with CuBERT layers and compare its performance with a randomly initialized Transformer of the same architecture and tf-idf baseline. CuBERT-based model outperforms the explored alternatives.**

I. Introduction

For machine learning applications source code can be represented as sequences of tokens [1], abstract syntax trees [2] or flow graphs [3]. While graph approaches can be better in capturing program structure and semantic relations, effective text processing methods have been developed for the natural language processing domain. The naturallness hypothesis [4] suggests that source code has the repetitiveness and predictability properties as well as natural languages and thus NLP methods are applicable for source code.

These NLP methods include models for creating contextualised text embeddings that can exploit the large amounts of unlabeled text data available by pretraining on unsuperwised tasks [5], [6], [7]. Using these text representations produced state-of-the-art results in various tasks with not enough data available to train from scratch.

BERT model [7] has recently been applied to SE tasks. [8], [9]. We first verified the results on variable misuse task, which is defined as determining whether a wrong variable has been referenced anywhere in a function. Then we apply CuBERT to two new tasks: move method refactoring and method name prediction task. In move method refactoring task the model should find feature envy code smell and propose a refactoring that would solve it. The name prediction task is a kind of a code summarization task: given code from a body of a method or a function the aim is to predict its name.

In this work, we adopt the CuBERT model for 2 new tasks: move method refactoring and method name predictions. We report high accuracy in move method refactoring prediction. For method name prediction, we show that pre-training outperforms transformer and tf-idf baselines.

II. Background

A. BERT in SE

In recent works BERT model have been successfully applied to software engineering domain. BERT models pretrained on large sets of data improved performance on various downstream tasks.

The CodeBERT [8] is aimed at multimodal tasks. It was trained on a dataset consisting of both code (in 6 programming languages) and corresponding natural language text fragments and then applied to two tasks involving both source code and natural language: code search by natural language description and code probing, a task of selecting a token for a masked position from a limited set of tokens.

The CuBERT [9] model is pretrained separately on Python and Java datasets. In order to still keep some structual information while representing programs as sequences of tokens the authors introduce special tokens for syntax elements such as scope begining and end. They fine-tune the Python model on five classification tasks which include variable misuse detection, wrong binary operator detection, swapped operands detection, function-docstring mismatch detection, exception type prediction and a one pointer prediction task, which is variable misuse prediction and repair. Authors compare fine-tuned CuBERT models with BiLSTM and Transformer, CuBERT outperformed them in all the tasks.

B. Move Method refactoring

Machine learinng has been applied to code smell detection including feature envy detection. In [10], a synthetic dataset is created for training deep neural networks. In [11], a variational autoencoder over AST is applied to extract features for smell detection.

Kurbatova et al. [12] propose an approach to move method refactoring recommendation over code2vec path-based representation. They move methods to some classes these methods can belong to instead of the original classes. Then methods and

classes are separately vectorized, reprenentations of vectors and methods are concatenated. The problem is stated as binary classification of positive (the method originally belonged to the class) and negative (the method was moved to the class during dataset generation) method-class pairs.

C. Method name prediction

Method name prediction is the task of extreme code summarization and used as a benchmark both for new kinds of architectures and reprenentations in the area of machine learning on source code. Sequence-to-sequence models with an autoregressive decoder are usually used to generate names, but as a baseline method nearest neighbor search in the space of code embeddings is also being used[13]. Allamanis et.al.[13] use CNN with attention mechanism to predict method names for Java. Alon et.al.[14] evaluated the quality of their path-based code representation on this task too and Python is among many languages they benchmark on, as in our work. One of the most recent works that touched on method name prediction is ContraCode[15], a self-supervised algorithm for learning task-agnostic semantic representations, it was evaluated on the task of predictiing names for JavaScript methods.

D. Variable misuse

In recent work, the problem of variable misuse (varmisuse) [16] was proposed. A variable misuse error occurs when a variable other than the correct one for a specific location is used. These errors may occur in practice, e.g. a developer copy-pasted similar code but forgot to rename all occurrences of a variable in a fragment.

Varmisuse problem was used to evaluate the CuBERT model in the original paper [9]. Given a function and the task is to predict whether there is any location containing a varmusiuse error. They report state-of-the-art 95.49% accuracy on this task.

III. Proposed approach

A. Move method refactoring

We follow the framework of [12] replacing code2vec vectorization with CuBERT contextualized embeddings. We found fine-tuning to be too expensive on this task given the computational resources we had access to so we limited ourselves to executing the model to get embeddings. The first way of vectorizing methods and classes is to pass all the lines through BERT and take the embeddings of <CLF> token. Then we either use the averages of line reprenentations as the vectors of methods and classes to concatenate them and pass to a classifer or pass them to a model that accepts sequences of vectors. In order to improve performance, methods longer than 100 lines and classes longer than 1000 lines were skipped. For methods we also applied another way of vectorization by passing to the model their bodies truncated to the context size.

B. Method name prediction

Data:

We use Python subset of CodeSearchNet dataset [17] for the task of method name prediction. Dataset consists of function-docstring pairs. Train, validate and test sets have 412178, 23107 and 22176 examples respectively.

Preprocessing:

We preprocessed all the data as follows:

1) Removed docstrings from fuctions
2) Replaced every occurence of a function's name in its own code with a placeholder character '_' to prevent data leak
3) In some rare cases such preprocessing broke Python syntax, such training examples were discarded ($\approx 0.01\%$ training examples)
4) Tokenized with CuBERT tokenizer

Models:

1) CuBERT:
   Since fine-tuning the whole 24 layer BERT-Large model would be too time-consuming, we initialize a smaller sequence-to-sequence model with CuBERT weights. Namely, we use Transformer architecture with a 6 layer encoder and a 2 layer decoder. Encoder is initialized with 6 lower layers of CuBERT and decoder is initialized with lower 2. Embedding layers of both the encoder and the decoder are taken from CuBERT too. We used a version of CuBERT model with context size of 512 tokens.
   To prevent both overfitting and catastrophic forgetting 4 lower layers of the encoder and its embedding layer are frozen. We use a relatively large batch size of 512 examples, Adam optimizer and linear warmup learning rate scheduler with subsequent linear decay, maximum learning rate is 0.00001. Learning rate warmup takes 0.15 of total training time. Model had been training for 7 epochs until early stopping occured.
2) Transformer:
   To estimate the impact of pretraining we train a separate sequence-to-sequence transformer model of the same architecture, but initialized randomly. We use the same optimization hyperparameters as in CuBERT fine-tuing, but increase the learning rate up to 0.0005.
3) tfidf:
   As in [13] we use nearest neighbor search by cosine similarity in tf-idf space as a simple baseline to test our models against. Tf-idf features are computed on the training corpus tokenized with CuBERT tokenizer. It is important to note that this approach has an inherent disadvantage of not being able to predict names that are not present in the training set. In our test dataset only 29% of unique function and method names are present in the train sample.

C. Variable misuse prediction

Data:

We use the variable misuse classification dataset from CuBERT paper. The dataset is publically available on google cloud. It consists of Python functions. The dataset is balanced: for each function with varmisuse error there is the corrected version without varmisuse error and vice-versa. It contains 700708, 8192, 379400 train, validation, and test examples respectively.

Preprocessing:

The dataset was preprocessed with CuBERT tokenizer in order to represent each item as a list of tokens. Then we randomly shuffle the train set.

Fine-tuning:

For fine-tuning, we use 1-epoch pre-trained CuBERT model. We tried the following parameters: batch size 256, optimizers: SGD, Adam, LAMB, learning rates $10^{-4}, 5 \cdot 10^{-5}, 10^{-5}, 5 \cdot 10^{-6}, 10^{-6}, 10^{-7}$, $context\_length = 512$. We gradually warm up the learning rate on first 10% training examples.

Fine-tuning on the whole dataset did not work at all. Thus, we tried fine-tuning on the first 8000 examples of the training set. This size allows us to run experiments much faster. This allowed us to find bugs in our pipeline.

IV. Evaluation

A. Move method refactoring

1) Data: We used the dataset from [12]. For testing the data uploaded by authors was used and the training part was generated from the same source code repositories (the training dataset is not completely the same because the states of the repositories have changed). For each method that can be moved to another class a true data point is generated of the method and its original class with this method removed. Also, a false data point is generated for each class the method can be moved to. The data points for positive pairs are repeated as many times as many negative classes are found the corresponding methods.

2) Results: SVM, gradient boosting and multilayer perceprton were tried as classifers for averaged vectors. For sequences, LSTM, CNN and simple attention were tried. MLP and CNN preformed best, so we report their results.

The f1 scores for classifying pairs are shown in Table I. They show sequential representation to perform better than averaging and no significant difference between averaging line embeddings and passing method to the model.

In Table II refactoring recommendation f1 scores are presented. The way they are calculated is similar to those of [12]: refactoring is recommended if for any pair with given method the model output exceeds 0.5 and the class with highest probability is chosen. Precision is defined as the ratio of number of correcly recommended refactorings to the total number of refactorings recommended and recall is defined as the ratio of number of correcly recommended refactorings to the number of methods moved. The f1 scores are averaged across classes.

TABLE I
MMR PAIRWISE F1

	Averaging vectors	CNN over sequence
Per line	0.76	0.79
Method embeddings	0.75	0.81

TABLE II
MMR RECOMENDATION F1

	Averaging vectors	CNN over sequence
Per line	0.87	0.82
Method embeddings	0.86	0.82

3) Validity threat: Although the f1 scores are higher than in [12], the comparison is not completely correct. There is a possibility of data leakage in the way we removed methods from classes by simply cutting away the method body: method calls in its original class are not preceeded with an object reference and this difference is present in the model input.

B. Method name prediction

To generate names with sequence-to-sequence models we use beam search with a beam width of 5. For tf-idf model names of the 5 most similar functions from the train set are taken. Top-1 metrics consider only the candidate with the highest score, which is log probability in the case of sequence-to-sequence models and cosine similarity in the case of tf-idf. Top-5 metrics are the maximum from the 5 candidates. Precision, recall and f1 scores are computed for single tokens of the function names. Precision describes a fraction of tokens present in the predicted name, which are also present in the true name. Recall describes a fraction of ground truth token, which model managed to predict. According to all of the metrics CuBERT-based model outperforms Transformer and tf-idf. Also it is worth mentioning that CuBERT learns faster than a randomly initialized model - it reached approximately 0.09 exact match only after the first epoch.

C. Variable misuse prediction

Here we discuss the results of our fine-tuning experiments on the variable misuse classification task.

First, we evaluated our pipeline by passing the varmisuse model [9] from CuBERT paper through it. Their model showed AUC 0.973, which is similar to the performance in the original paper.

When fine-tuning on the whole training dataset we were getting accuracy near 50%. The model was not learning any dependency from the data. In hindsight, we suspect that it was due to random shuffling of the training set.

Later on, we removed shuffling from our pipeline and ran an experiment on the first 8000 examples from the training set. After two epochs we got 89.7% accuracy on the test set.

V. Conclusion

We investigate the performance of pre-trained contextualized embeddings for source code introduced in the CuBERT

TABLE III
METHOD NAME PREDICTION METRICS.

Model	Exact match		Precision		Recall		F1	
	top1	top5	top1	top5	top1	top5	top1	top5
CuBERT (2+4)e2d	0.09	0.164	0.261	0.433	0.228	0.407	0.236	0.41
Transformer 6e2d	0.045	0.082	0.158	0.3	0.136	0.277	0.142	0.28
TF-IDF	0.036	0.052	0.095	0.184	0.092	0.175	0.09	0.173

Precision, Recall and F1 are computed for unigrams.

paper [9]. For this purporse, we verify the result of CuBERT on variable misuse task and adopt it for two new tasks: move method refactoring recommendation and method name prediction.

For the variable misuse task, the provided fine-tuned model was loaded giving the same results as in the original paper. But we failed to reproduce fine-tuning ourselves due to bugs in our fine-tuning pipeline.

On the move method refactoring task, promising results were obtained, but their validity is questionable.

Fine tuned model for method name prediction outperforms randomly initialized Transformer and TF-IDF baselines. Tf-idf approach obviously suffers heavily from its inability to come up with new names, resulting in it having lowest metrics out of all the approaches. Unfortunately, we did not find any works dealing with method name prediction tasks on Python subset of CodeSearchNet datasets. Our accuracy (exact match) is lower than the ones obtained in the work of Alon et.al. [14] for Python, but they dealt with a different dataset. At the same time our metrics are higher than the ones obtained in ContraCode [15] for JavaScript subset of CodeSearchNet, which may suggest that our results are somewhat reasonable. Similarly to the CuBERT [9] paper, we show that pre-trained models can outperform baselines.

Our code is available at https://github.com/maxkvant/bert-on-source-code

REFERENCES

[1] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1073–1085, 2020.

[2] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Convolutional neural network over tree structures for programming language processing," 09 2014.

[3] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[4] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *Proceedings - International Conference on Software Engineering*, pp. 837–847, 06 2012.

[5] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[6] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *NAACL-HLT*, 2018.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *ArXiv*, vol. abs/2002.08155, 2020.

[9] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," Vienna, Austria, 2020.

[10] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[11] M. Hadj-Kacem and N. Bouassida, "Deep representation learning for code smells detection using variational auto-encoder," *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2019.

[12] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of move method refactoring using path-based representation of code," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 315322. [Online]. Available: https://doi.org/10.1145/3387940.3392191

[13] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, 2016, pp. 2091–2100.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[15] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[16] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[17] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.