Variable Name Prediction with Code-Transformer

Advanced Machine Learning project report, 2021-2022

Raffaele Pojer

July 6, 2022

Abstract

With code-transformers [Zügner et al., 2021] demonstrate that by combining source code with its abstract syntax tree (AST), it is possible to obtain better results using only language-agnostic features, outperforming all the other competitors in predicting the function name of multiple languages. In this project, the aim is to use and adapt this transformer to the task of predicting the variable names. Even if guessing the name of the variable is a complex assignment, the results are similar to the ones obtained in the original paper, suggesting us the proof of work.

1 Introduction

In [Zügner et al., 2021], the authors combine two complementary data representations of programs: the source code and its abstract syntax tree to learn its joint representations.
Before that, [Hellendoorn et al., 2020] also explore the combination of Context and Structure using a Graph Relational Embedding Attention Transformer (GREAT) in a language-specific setting. The original paper used 5 different programming languages for the task of code-summarisation. In this project, the problem is restricted only to one language to predict the name of variables inside Python code.
Variables prediction can be utilised for different aspects, e.g. to reduce space and improve efficiency, like the client-side code JavaScript which can be minimised in order to save space and improve speed. Several works have been done to reconstruct minimised code, [Bavishi et al., 2018] use recurrent neural networks to predict all the variables, function name of minimised JavaScript code obtained great results compared to the static method created for the same task. This project demonstrates how this transformer can predict variable names of code using pre-trained models. Since the authors prove the language-agnostic features of the transformer, it is very likely that results obtained for one language can be transferred also to other languages (e.g. JavaScript for code minimisation).

1.1 Program synthesis

This project is related to the field of program synthesis, the task of generating code or program automatically that satisfies some user constraints. It's a very challenging field, with numerous applications that can help the end-user (technical and non-technical) to explore, maintain and design code. Not surprisingly, the structure of the code, being much more general than the syntax, helps in performance and generalisation. Understanding the variable name means also understanding the whole context of the code and this can be done by learning meaningful program embeddings. With [Zügner et al., 2021], the authors focus on the point that code can be better represented with its structure and the semantics learned together using a transformer.
One famous example of how program synthesis is used is GitHub Copilot[1], which aims to generate pieces of code according to some high-level user specifications.

2 Problem definition

Provided the model, the dataset and some specific binaries, with this code-transformer, the authors demonstrate that it is possible to predict the function name of multiple languages: Python, Go, Ruby, JavaScript and Java. The authors show how the benefits of combining the context and

[1] https://copilot.github.com/

structure lead to better results and settings the state-of-the-art for this task. The generalisation improves substantially by providing directly the structure of language. This structure helps in long-ranging dependencies between tokens in the source-code.
The transformer select the part of the input thanks to the self-attention, the equation for a single head is

$$Attention(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{1}$$

All this information is then combined in the self-attention operation of the transformer (eq. 1). The attention is then adapted from the formulation of [Dai et al., 2019] and [Yang et al., 2019].

$$\mathbf{A}^{rel}_{i,j} = \mathbf{Q}_i^T\mathbf{K}_j = \mathbf{E}_i^T\mathbf{W}_q^T\mathbf{W}_k\mathbf{E}_j + \mathbf{E}_i^T\mathbf{W}_q^T\mathbf{W}_r\phi(r_{i\to j}) + \mathbf{u}^T\mathbf{W}_k\mathbf{E}_j + \mathbf{v}^T\mathbf{W}_r\phi(r_{i\to j}) \tag{2}$$

Where $\phi(r_{i\to j})$ is the relative distance between token i and token j in the sequence, $\boldsymbol{u}$, $\boldsymbol{v}$ are learnable bias vectors, and $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_r$ are the projection matrices of the query, key and relative distances, respectively.
Given the distances of the source code and the distances of the AST, it is also possible to infer the variable names in a specific code snippet.

2.1 Distances

The combination of Structure and Context is done by mapping the AST-tokens and the source-tokens. By looking at the range of each node, tokens are combined with the nearest node of the AST. The pairwise relations between the AST nodes are then embedded with the token distances in the self-attention of the transformer.

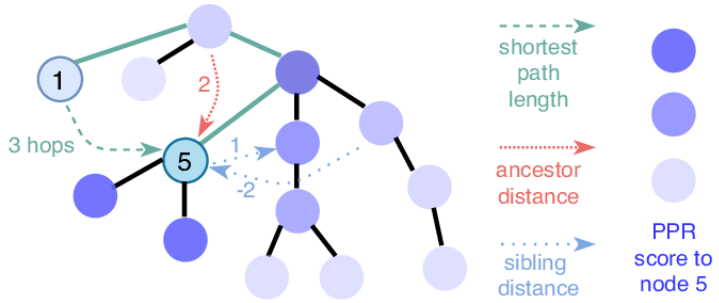

Figure 1: Structure distances taken from [Zügner et al., 2021].

2.2 Architecture

The transformer takes inspiration from the XLNet architecture [Yang et al., 2019] in the attention computation to use only language-agnostic features. These features can be computed directly from the AST (e.g. sibling shortest path, ancestor shortest path, personalised PageRank). To all the transformer decoders there is a pointer network [Vinyals et al., 2015]. The authors use this network to enhance the predictions by pointing at positions in the input space (pointer networks use attention weights as a pointer to the input space). That improves results for less frequent tokens, and enables the models to predict tokens that are not present in the vocabulary (e.g. combinatorial search problems).

2.3 Datasets

The dataset used for this task is the CodeSearchNet Challenge [Husain et al., 2019], the same as the original paper. This dataset contains multiple programming languages: Python, Javascript, Go and Ruby.
The problem was restricted to predicting only Python code thus we removed all data referring to languages which are not Python. In CodeTransformer for the Java languages, it was used another dataset containing only Java code. Also, this dataset was not included in this project. Only for

	Samples per partition		
Dataset	Train	Test	Val.
CSN-Python	412,178	22,176	23,107

Table 1: Samples in the Python partition

the Python code, the size of the dataset is about 460K samples in total.

3 From code-summarisation to variable name prediction

Transformers are well known for their promising accuracy, and for this reason are now the *de-facto* standard for many domains, in particular Natural Language Processing or Computer Vision. However, in order to obtain such resources, they need to optimise lots of model parameters requiring high-level computational and memory resources. Due to limited resources, the main objective was to train the transformer with an already pre-trained model on the task of code-summarisation. Since the goal was limited only to predicting Python variables, the model taken was the one trained only on one language.
Models, code and configuration was made freely accessible by the authors on GitHub[2] .
In order to apply transfer learning for this task and not retrain the entire transformer from scratch, the size of the embedding of the transformer trained for code-summarisation needs to be the same as for the variable prediction. In the next sections, it will be explained more in detail why the size of the embeddings can be different if the task changes. In order to train a model on a multi-language settings, there would need much more preprocessing on the extraction of variables for the diverse languages.

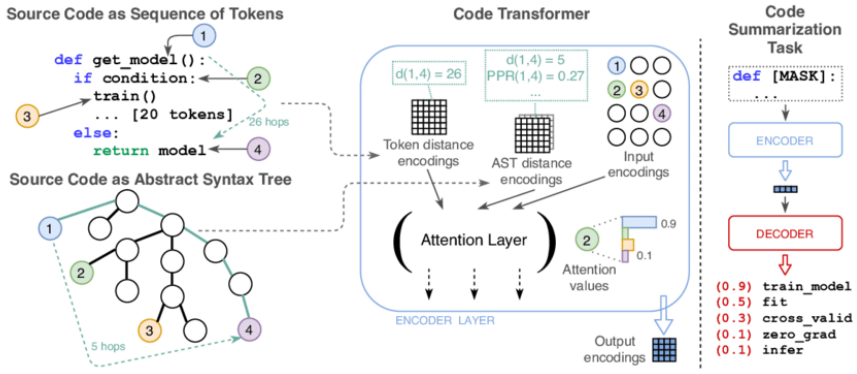

Figure 2: How each sample is preprocessed to be embedded in the self attention of the transformer, taken from [Zügner et al., 2021].

3.1 Preprocessing

To train the transformer, the samples need to be preprocessed through a multi-step phase. The first phase consists mainly of the preprocessing of the text. The variables are tokenised and each token can have multiple sub-tokens e.g. `dataFromWeb` became `[data', 'from', 'web']`, the same was happening with the name of the functions. During this process, all the numbers, strings and indentations are masked with special tokens. After the textual preprocessing, the actual preprocessing for the code-transformer begins. Originally there were two steps (1 and 2), but for the extraction of variables, there was needed another preprocessing (Stage 0).

3.1.1 Stage 0

In Python, variables can be multiple things and can be easily seen as objects, functions, etc., thus to have a more accurate, clean and "controllable" definition of variable, it is better to look directly at the Abstract Syntax Tree of the code. In order to generate the AST, code-transformer uses Semantic[3], an open-source software that produces AST for multiple languages.
In our scope, a variable is something that has an assigned value (e.g. `fooVar = something`), function arguments and asynchronous function variables. Variables that do not have an alpha numeric name are not also considered (e.g. underscore or asterisk [`_ , *`] in for loop).
Unfortunately, Semantic does not have simple approaches to walk inside the structure generated,

[2] https://github.com/danielzuegner/code-transformer
[3] https://github.com/github/semantic

if not by using other open-source programs to query the tree[4]. For this reason, the code-snippets were preprocessed also by another AST parser[5]. This AST parser is from the standard modules already present in Python, and is used for walking in the generated tree much easier. Unfortunately, this parser uses the compiler of Python in order to work, and if there are present some errors in the code, the AST is not generated. Semantic, on the other hand, is more robust and fails much less (indentation problems and some minor issues do not affect Semantic). In the next stage, the vocabulary is constructed from the token analysed in this stage, and if during the Python AST conversion some code-snippets fail, the risk is to loose some words that were not previously lost. To be sure that during the next step, the code-snippet are the same as in the original work, and not less for the AST parser, it was needed to adjust some examples.
All the samples of the dataset are passed first to the Python AST parser to check if some examples fail. If there is no variable inside, code error problems or other minor issues that break the parsing, a *dummy variable* is added. For each code-snippet fail, a new line of code is carefully placed by a script, after the function firm and arguments as the fist line in the body of the function. The line added is a `self = self`, in this way also the "logic" of the program is preserved or not ruined.

Table 2 shows the number of code-snippets that are parsed correctly by the Python AST. The number of samples that fail are less than the 1.5% of the total samples. This numbers also does not consider the cases where also Semantic fails during the generation of AST. When a failure is detected, the new *dummy variable* is added to the code for making sure that at least that variable is present in the code.
The example Listing 1 is a typical error that can cause the failure of the parsing by the Python AST: the indentation is not correctly done and it is not possible to extract any variable from it.

```
def CrossEntropyFlat(*args, axis:int=-1, **kwargs):
        "Same as `nn.CrossEntropyLoss`, but flattens input and target."
            return FlattenedLoss(nn.CrossEntropyLoss, *args, axis=axis, **kwargs)
```

Listing 1: One example found in the dataset that fails with the Python AST. The comment and the *return* are in two different indentation, this can be a typical error in Python.

All the samples of the dataset are preprocessed by this phase and saved in the same format as the original dataset. The following stages make use of this new dataset for constructing all the necessary elements for this task.

	Samples per partition		
Dataset	Train	Test	Val.
CSN-Python (parsed)	407,242	21,896	22,762

Table 2: Number of samples correctly parsed by the Python AST

3.1.2 Stage 1

This step of preprocessing is mainly focused on the generation of the AST for each specific language. Step 0 makes sure to generate a code-snippet that does not fail during the parsing by Python AST. All the code snippets are first examined by Semantic. The authors use a specific version of this software that produces a readable graph of the AST in a `.json` format. For reasons not well stated by the developers, newer versions do not have this option of generating an AST graph. All the code-snippets that do not fail during the AST generation, are then mapped to the corresponding text token in the nearest range. Each sample in the dataset comes with the function, the function name, the docstrings and other proprieties not needed.
Since variable names are not present, before passing to the next stage of preprocessing also the variable extraction needs to be done. With this parser, the temporary AST is generated, and all the variables are found (if before the parser fails, only the *dummy variable* is "detected"). With the list of the variables inside the code, only one variable is chosen randomly for each code snippet.

[4] https://github.com/github/semantic/issues/674
[5] https://docs.python.org/3/library/ast.html

Original	Masked variable	Predicted

`def ensure_directory(path):`
`  dirname = os.path.dirname(path)`
`  if not os.path.isdir( dirname ):`
`  os.makedirs( dirname )`
	`def ensure_directory(path):`
	`  a = os.path.dirname(path)`
	`  if not os.path.isdir( a ):`
	`  os.makedirs( a )`
		dir

`def to_bytes( text , encoding='utf-8'):`
`  if not text :`
`  return text`
`  if not isinstance( text , bytes_type):`
`  text = text .encode(encoding)`
`  return text`
	`def to_bytes( a , encoding='utf-8'):`
	`  if not a :`
	`  return a`
	`  if not isinstance(a, bytes_type):`
	`  a = a .encode(encoding)`
	`  return a`
		value

`def read_data(self, f):`
`  file = open(f, 'r',`
`  ↪ encoding='utf-8').readlines()`
`  r_file = []`
`  for line in file:`
`  tmp = line .strip().split()`
`  r_file.append(tmp)`
`  self.voc.add_word_sent(tmp)`
 	
`  return r_file`
	`def read_data(self, f):`
	`  file = open(f, 'r',`
	`  ↪ encoding='utf-8').readlines()`
	`  r_file = []`
	`  for a in file:`
	`  tmp = a .strip().split()`
	`  r_file.append(tmp)`
	`  self.voc.add_word_sent(tmp)`
	
	`  return r_file`
		line

`def send_command(self, command ,`
`↪ as_list=False):`
`  action = actions.Action({`
`  'Command': command ,`
`  'Action': 'Command'},`
`  as_list=as_list)`
`  return self.send_action(action)`
	`def send_command(self, a ,`
	`↪ as_list=False):`
	`  action = actions.Action({`
	`  'Command': a ,`
	`  'Action': 'Command'},`
	`  as_list=as_list)`
	`  return self.send_action(action)`
		command

`def init_weights(self):`
`  stdv = 1.0 /`
`  ↪ math.sqrt(self.hidden_size)`
`  for weight in self.parameters():`
`  weight .data.uniform_(-stdv,`
`  ↪ stdv)`
	`def init_weights(self):`
	`  stdv = 1.0 /`
	`  ↪ math.sqrt(self.hidden_size)`
	`  for a in self.parameters():`
	`  a .data.uniform_(-stdv, stdv)`
		parameter

Table 3: Variable predicted with examples. In most of the cases, the variable predicted has some meaning in the context of the code. In this table it is reported how the masking is made during the interactive notebook (e.g. by replacing every instance of the variable with the letter a). The fist column is the original function, in the second the function with the masked variable and in the last column the predicted name of the variable.

3.1.3 Stage 2

The last phase of preprocessing begins with the construction of the vocabulary using the examples seen. All the words that occur less than 100 times are substituted with the `<unk>` token. Then it is computed all the pair-wise relations between the nodes in the generated AST. For reducing the size of the sentence, all punctuation tokens are removed (the authors say that it does not affect the performance other than by only slowing down the training). All the code-snippets longer than a pre-defined constant are removed (train: 512, test and validation: 1000). For this phase, the only thing to adapt was the new elements referring to the variable name and it was removed all the elements referring to the function name.
The computation time for all these three phases was made using the CPU on a cluster in the University servers and took several hours to be complete.

3.2 Dataclass adaptation and samples management

A great effort was made to make all the pieces of the transformer work with the new parameters carried: the variables names and the variables names' embeddings. For this reason, the different class was rewritten or slightly changed to make sure all the samples do not throw errors during training time.

3.2.1 Masking

One of the critical points for this project was to understand where and how the transformer infers the function name or the variable name during training. By inspecting the code, the tokens for the function name, or in this case for the variable name, are found inside the list of tokens of the code snippet. These tokens are then masked by the pointer network during the inference time. Differently from the task of function-name prediction, the tokens can appear multiple times in the body of the function and need to be masked during training.

4 Results

The results obtained with the code transformer were on a pre-trained model. This model was trained with Python on 500,000 iterations. Table 4 shows the results for the three models. It is clear to see how the code-transformer performs much better than the other competitors for the variable prediction task. It is also worth mentioning, that the pre-trained models for GREAT and XLNet were not used for this task, and probably more training time would increase the score for all the models.
The results obtained with the code-transformer are similar to the results obtained for the function name predictions: this encourages the validity of the work done. It is reported also the result of one training without all the instances of the same variable masked. In this specific setting, only one instance of the variable is masked. The other tokens referring to the same variable are not masked and can be seen by the transformer, allowing it to understand better the missing piece of code. The last row of results shows the substantial differences in hiding, or not, all the tokens of the same variable. The transformer can see the already seen tokens.

Table 3 shows some examples of variables predicted by the transformer. These examples were predicted in an interactive jupyter notebook, to make sure that we hide all the variables in the inference time, all the variables were "masked" by the letter *a*. Even if the difficulty of predicting the exact name for a variable of the examples is quite complex, the predicted variables maintain some meaning in the context of the function.
The first example the variable *dirname* was masked in every part of the function with *a*. The predicted word was not the same as the original but kept the meaning of the name. In this case, the prediction is still counted as wrong. A more in-depth analysis can be done on the distance between the predicted and original token, but it was chosen to keep the same as in the original work both to take away complexity and to have a more obvious comparison with the previous one. In the last row of the table, there is another interesting example to see. Instead of predicting the word *weight*, it predicted the singular of the word parameters (by truncating the final s), a typical behaviour in Python code.

	Python			
Dataset	Precision	Recall	F1	Top5
code-tranformer	**37.47**	**35.44**	**35.94**	**46.72**
GREAT	22.70	20.12	20.52	31.90
XLNet	31.23	28.76	27.31	39.89
code-tranformer (with no hiding variables)	54.36	54.32	54.0	85.62

Table 4: Results obtained with the 3 different architectures. In the last row it is reported the results obtained by training the transformer without masking all the variables instances to the pointer network.

5 Conclusion and future work

This project proved the power of this architecture in combining the structure and the context of source code also for the variable prediction task. The results obtained reach similar results in the task of code-summarisation, yielding improvement compared to its competitors that have only monolingual features. Surely, there are other ways to predict variable names and achieve higher results: one of the most important is better adapt the extraction of variables for other languages. New variables extracted from different languages will help in one of the most critical points of this project: the limitation of a specific language (Python). We do not explore the real benefit of a language-agnostic model in this context of variable prediction.
Also, it will be nice to see multivariable predictions inside the code and predict more than an instance of a single variable. All of these points can be addressed as possible future works to explore.

References

[Bavishi et al., 2018] Bavishi, R., Pradel, M., and Sen, K. (2018). Context2name: A deep learning-based approach to infer natural variable names from usage contexts.

[Dai et al., 2019] Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860.

[Hellendoorn et al., 2020] Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. (2020). Global relational models of source code. In *International Conference on Learning Representations*.

[Husain et al., 2019] Husain, H., Wu, H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

[Vinyals et al., 2015] Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

[Yang et al., 2019] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

[Zügner et al., 2021] Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., and Günnemann, S. (2021). Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*.