Contextual Program Token Prediction

Scott Kovach
dskovach@stanford.edu

December 13, 2019

1 Introduction

We consider the problem of predicting a Python source code token given its context. Predictive software development tools can improve programmer productivity and help implement correct code. Although a given programming language has a known syntactic structure, developer tools can be difficult to implement, usually don't formally model higher-level programming idioms, and may fail in the presence of syntactically invalid programs. Our goal is ultimately to explore the effectiveness of autocompletion suggestions and error detection without relying on a formal model of the programming language. We use a transformer [6] based architecture trained with the masked language model (MLM) objective over a dataset of publicly available open-source Python programs and evaluate it on the token prediction problem.

2 Related Work

The project was initially inspired by TabNine [7], a proprietary project that purports to use the GPT-2 [11] model to facilitate autocompletion for many programming languages. It differs significantly from our work in that it also models the natural language content of strings and comments, is a much larger model trained on more data, and operates on many source languages. It is limited in that it makes predictions left-to-right, whereas we use bidirectional contextual information.

Our work is inspired by previous work in natural language modeling. [4] proposes the continuous bag-of-words model and uses it to obtain semantically meaningful word embeddings. In [9], the authors create a bidirectional transformer model BERT that can be pre-trained in an unsupervised manner on a large text corpus to produce weights that can be reused for various NLP tasks. Using the parameter reduction techniques from [10], we train a small model of this form from scratch on source code. Our work differs in that we use the trained model to directly solve the MLM problem, without any additional downstream task.

3 Dataset

The 150k Python dataset [1] consists of 150,000 unique Python source files obtained from GitHub. It contains both the textual versions of each program and parse trees in a json format obtained using the Python `ast` library.

For this project, we tokenize [2] the raw source files, discarding any with tokenization or encoding errors ($\sim$200 files). The Python tokenizer is the only language-specific information we use, and we use it only to segment the text. The model could be trained on another programming language corpus by substituting the appropriate tokenizer.

We split the dataset 60%/20%/20% for training/development/testing purposes. Across the roughly 113,000 files in the first two splits, we see roughly one million unique tokens; since each of our models represent inputs as one-hot vectors in a space of dimension equal to the vocabulary size, we must reduce this size. We make the following simpifications:

- We avoid modeling string literals; each occurrence is replaced by a special `STRING` token.
- Most variable/method names occur only a few times. At the same time, we expect many code blocks to have similar patterns of variable definition and reference, so we would like to model this structure. We include a set of "local name" tokens [8] in our vocabulary to stand in for infrequent tokens. For instance, assuming `foo` and `bar` are uncommon tokens, the line `foo = foo + bar()` would be tokenized as

  ```
  n1, '=', n1, '+', n2, '(', ')', '\n'
  ```

  where `n1` and `n2` represent elements of the finite set of local name tokens.
- The rest of the vocabulary consists of the most frequently occurring token strings (e.g., `\n`, `(`, `+`, `None`, `numpy`, `import`, etc.).

4 Methods

4.1 Continuous Bag-of-Words

The continuous bag-of-words (CBOW) architecture is used in [4] to train a low-dimensional vector space representation of words from some vocabulary V. The training objective assembles a context of words (X) around a target word (y), and optimizes the cross-entropy loss of the following network function:

$$y = \text{softmax}(U_2(U_1 x)),$$

where $x \in \mathbb{R}^{|V|}$ is an average of one-hot encoded words from X, $U_1 \in \mathbb{R}^{|V|\times k}$, $U_2 \in \mathbb{R}^{k\times|V|}$, and k is the hidden layer dimension.

4.2 Transformer Model

Here we briefly describe the transformer model; see [6] for full mathematical detail.

The transformer model defines a function on a sequence of vectors $[v_i]_{i=1\ldots n}$ of length n from the vocabulary space V. Each input token is annotated with its position in the sequence; then the (token, position) pair is embedded in a space of dimension d, the hidden dimension. Each inner transformer layer maps its input $I = [v_i]$ to another sequence of the same length. These layers contain a set of self-attention mechanisms (called "heads") and a feedforward network (FFN) with one hidden layer. Each attention head includes learned linear transformations

$$k : \mathbb{R}^d \to \mathbb{R}^{d_k}$$
$$q : \mathbb{R}^d \to \mathbb{R}^{d_k}$$
$$v : \mathbb{R}^d \to \mathbb{R}^{d_v},$$

referred to as key, query, and value maps, respectively; the key and query maps target the same space, and together determine a similarity measure. For each layer input vector v, the mechanism computes $q(v)$ and $k(v')$ for each other vector v' and then $w = \text{softmax}(\{q(v)^T k(v') | v' \in I\})$. The weight vector w is used to form a linear combination $\sum_{i=1}^{n} w_i v(v_i)$. A layer contains h heads, each with their own learned key/value/query maps. The head outputs of dimension d_v are concatenated to form the output of the attention mechanism for a given vector, so it is required that $h \cdot d_v = d$.

The attention output is passed through the FFN. Note that the elements of the vector sequence do not interact at this stage; the FFN outputs are computed independently for each input vector.

The self-attention mechanism and FFN are each wrapped in a residual connection [14] and layer normalization [13] with learned gain/offset parameters.

The final transformer layer is decoded by a linear transformation to the vocabulary space, and softmax is applied to obtain a final distribution for each input vector.

4.3 Masked Language Model

The masked language model (MLM) [9]is a loss function that represents the problem of token prediction. A training case is obtained by first sampling a window of tokens $[t_i]$ from an input file, then probabilistically selecting individual tokens as training examples. Each token position i is selected with probability 0.15. The original token at that position is used as the labeled target, and the input token t_i at that position is then either

- replaced by a special `MASK` token with probability 0.8,
- replaced by a random token with probability 0.1,
- or left alone otherwise.

Intuitively, this masking procedure forces the model to (1) produce good predictions for `MASK` tokens based on their context, (2) compute the identity function for ordinary unmasked, uncorrupted tokens, and (3) distinguish when a token was probably corrupted.

5 Results [15]

5.1 Model Parameters

For our CBOW baseline we use an embedding dimension of 64. For training, we use a context of four words before and after the target.

Given our limited computational resources, we train a small transformer. Our vocabulary size is 2048. We use a hidden layer of dimension 64, two attention heads per layer, embedding dimension 64, feedforward network size 256. The model has two layers, and shares attention and FFN weights across layers. The model contains 337216 parameters, of which 262144 come from the initial embedding and final decoding layers which map to and from the vocabulary space. For training, we use a context of 32 tokens. For comparison with the baseline, we also evaluate on contexts of eight tokens.

5.2 Training

We train the transformer (called TR below) for nine epochs across our ∼100 million token corpus.

The accuracy figures given below are taken from the testing dataset, but we noted that accuracy on the training set was roughly the same, suggesting the model has underfit. We note that the ALBERT model apparently does not converge either [12].

5.3 Model Comparison by Aggregate Accuracy

We give two accuracy measures:

- top-1: the test-case label is the token with highest predicted weight
- top-3: the test-case label token occurs amongst the three tokens with highest predicted weights

model	Top-1	Top-3
1-Gram	0.14	0.28
CBOW	0.42	0.67
TR [context=8]	0.56	0.73
TR [context=32]	0.68	0.81

We include the 1-gram model (individual token frequency) to note that the token distribution is very skewed. The transformer significantly outperforms the baseline; within the context of autocompletion, 81% accuracy would be quite helpful.

5.4 Accuracy by Token

Over a sampling of 250,000 test cases taken from the test files, we examine top-1 accuracy by token value across a sampling of the vocabulary and offer some interpretation:

token	top-1	#occurrences	token	top-1	#occurrences
local names:	0.80	47142	def	0.90	2251
NEWLINE	0.87	38837	if	0.71	1411
.	0.83	15430	import	0.76	1870
(	0.84	15401	0	0.20	744
,	0.75	13943	==	0.46	569
STRING	0.84	12529	True	0.21	512
=	0.75	9943	else	0.75	332
:	0.90	7710	**	0.73	276
self	0.85	5480	kwargs	0.61	274
INDENT	0.86	6141	bottom 1000:	0.02	5062

- The model has relatively high accuracy on the most frequently occurring tokens. Also, for tokens that tend to occur in pairs (e.g. `**kwargs`), the accuracy is good.
- For value expressions such as numeric or boolean literals, the accuracy is low; it appears that the model cannot effectively use context to guess the types of literals that should occur.
- The accuracy for local variable names is quite high (80%).
- For the bottom half of the vocabulary (tokens 1000 to 1918 when sorted by frequency) the accuracy is roughly zero. Some example tokens in this range: `reshape, 64, temp, ~, ArgumentParser`. It's possible that reducing the vocabulary size and relying more on locally named tokens would increase accuracy for these tokens somewhat, if they appear more than once per window.

6 Future Work

Published work on transformers generally use deeper (e.g. 12 layer) and wider (e.g. 1024 hidden dimension with 16 heads) models. With a larger dataset and more computational resources we would like to perform a larger experiment. We also expect larger window sizes to improve accuracy, up to a point.

We would like to test the model within an editor for autocompletion (suggesting likely tokens at the cursor, or during an edit within a code block) and error detection (detecting tokens t such that $p(t)$ is small).

Much useful contextual information within a program is not immediately available; allowing the transformer to attend to a function definition while writing at a call site might be very helpful. More generally, combining semantic information obtained from a language-specific tool with the learned information of the model might enable very powerful suggestions or refactorings.

References

[1] Raychev, V., Bielik, P., and Vechev, M. Probabilistic Model for Code with Decision Trees. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (2016), OOPSLA '16, ACM

[2] https://docs.python.org/3.6/library/token.html

[3] https://docs.python.org/2/library/ast.html

[4] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).

[5] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.

[6] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.

[7] https://tabnine.com/

[8] https://web.stanford.edu/ chshah/files/contextual-code-completion.pdf

[9] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

[10] Lan, Zhenzhong, et al. "Albert: A lite bert for self-supervised learning of language representations." arXiv preprint arXiv:1909.11942 (2019).

[11] Radford, Alec, et al. "Better language models and their implications." OpenAI blog (2019).

[12] https://openreview.net/forum?id=H1eA7AEtvS&noteId=SklNxFQ6KB

[13] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." arXiv preprint arXiv:1607.06450 (2016).

[14] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[15] implementation available at https://colab.research.google.com/drive/1l2dP_x_-nBVhuAfWlAsz1pxBWA-hBYGk