Time-Efficient Code Completion Model for the R Programming Language

Artem Popov[1], **Dmitrii Orekhov**[13], **Denis Litvinov**[1],
Nikolay Korolev[12], **Gleb Morgachev**[14]

[1]JetBrains s.r.o., [2] Lomonosov Moscow State University,
[3] ITMO University, [4] Moscow Institute of Physics and Technology
{artem.popov, dmitrii.orekhov, denis.litvinov,
nikolai.korolev, gleb.morgachev} + @jetbrains.com

Abstract

In this paper, we present a deep learning code completion model for the R programming language. We introduce several techniques to utilize language modeling based architecture in the code completion task. With these techniques, the model requires low resources, but still achieves high quality.

We also present an evaluation dataset for the R programming language completion task. Our dataset contains multiple autocompletion usage contexts and that provides robust validation results. The dataset is publicly available.

1 Introduction

Code completion feature (for simplicity we will refer to it as autocompletion) is used in an integrated development environment (IDE) to suggest the next pieces of code during typing. Code completion engines can accelerate software development and help to reduce errors by eliminating typos.

In recent years quality improvements in the code completion task have been achieved with the transformer language models. Models with a huge amount of parameters usually demonstrate better performance (Brown et al., 2020), but in practice code completion is executed on a user laptop with limited computational resources. At the same time code completion should run as fast as possible to be considered as a convenient development tool.

In this paper, we show that the autocompletion task can be solved with a fairly good quality even with a small transformer-based model. We propose several techniques to adapt the model which was originally designed for NLP tasks to our task.

It is hard to build a good autocompletion system for dynamically typed languages without machine learning methods (Shelley, 2014). Let us consider an autocompletion of a function argument scenario. In static languages, an argument type is determined in the function definition. We can collect variables of this type from the scope in which the function is called. These variables may be used as an autocompletion output. However, in dynamic languages the argument type information is omitted. Since all dynamic languages are interpreted, variable types can not be obtained without running a program or special tools usage.

We choose a dynamic R programming language for our experiments. To the best of our knowledge, there are no papers about code completion based on deep learning for the R programming language specifically.

We also propose an evaluation dataset for the R programming language collected from the open-source GitHub projects[1]. Our dataset is divided into several groups specific for different code usage contexts. For example, there is a separate group containing package imports and another one containing function calls.

2 Related Work

There are many ways to design code completion models. One of the methods is a frequency-based system. The statistical language model is used to rank a set of possible completions extracted by the rule-based methods (Tu et al., 2014). Bruch et al. (2009) proposed proposed a ranking machine learning model, which additionally takes a feature vector describing completion context as an input.

Lately, deep learning approaches have gained popularity. Completions are generated by autoregressive models such as LSTM or transformer-based language models (Li et al., 2017) trained on a large source unlabeled code corpora. Some large models such as GPT3 (Brown et al., 2020) can even perform a full-line autocompletion with

[1]https://github.com/arti32lehtonen/rcompletion_evaluation_dataset

promising quality. Alon et al. (2019) suggest to predict the next node of the abstract syntax tree (AST) of the program to get completions. Liu et al. (2020) propose to predict the token and its type jointly to improve completion performance for identifiers.

3 Model

3.1 Baseline model

We use conventional GPT-2 (Radford et al., 2019) architecture with Byte Pair Encoding (BPE) tokenization (Sennrich et al., 2015), but with fewer layers and heads and a lower hidden size. We train it on a standard language modeling task, predicting the next BPE token x_t from the previous ones:

$$\mathcal{L}_{lm} = \sum_t \log p(x_t|x_{<t}) \rightarrow \max \tag{1}$$

However, we use special preprocessing to make this task easier. In particular, we apply R lexer to a source code to get so-called program tokens. We use that information to replace numerical and string literals by type-specific placeholders, delete comments and remove vector content.

At inference time we exploit beams search and softmax with temperature. To prevent generation of the repeating elements we use penalized sampling (Keskar et al., 2019).

3.2 Variable Name Substitution

As we know, transformers suffer from $O(n^2)$ complexity with n as input length. It limits their ability to exploit large contexts and therefore limits code completion quality. If we take only the last tokens as input, it can dramatically reduce a model quality in a code completion task. For example, it is very complicated to get a variable with a rare name in the model output if it is declared at the start of the program and never used after that.

While BPE tokenization allows us to represent rare words with fixed-size vocabulary, they can still have damaging effect on the training and the inference stages. We observe that rare variable names in a source code unnecessarily extend input sequence length, thus reducing effective context length. We tried to use some transformer modifications such as Reformer (Kitaev et al., 2020) to reduce inference time but the quality drop was very high.

Here we propose a simple idea of replacing a rare variable name with a placeholder (`varK`, where K is the variable index number) if its frequency is less than a certain threshold. Also, we should note, that by such replacement the language modeling task becomes a bit easier. Since there is no need to remember complex variable names and the model can concentrate on predicting more useful token sequences.

Proposed substitution increases quality and speed not only because of the context size. It is impossible to get a long name variable from the model output because there is a limit on the number of generation iterations. While such transformation allows us to generate a variable of any length.

3.3 Prefix Generation

We observe some discrepancy between training objective and model usage cases. Usually, the user can improve completion results by typing in several first characters of the desired token. The problem is that during inference user may invoke an auto-completion service, while the pointer is still in the middle of the BPE token of the desired output. So, at the training stage the tokenization is determined, while during inference it can take an arbitrary form. When the word is typed in by the user, its prefixes may be decomposed into BPE tokens in several different ways. For example, if a user wants to get the variable `maxDf` consisting of `max` and `df` BPE tokens, then the typed `m` can lead only to an unlikely sequence `m`, `ax`, `df`.

We try to work around this issue by utilizing token prefixes. To incorporate signal from the prefix, we propose to roll the pointer back to the start of the program token and to utilize only those BPE tokens that match our prefix during beam search. Searching tokens with the right prefix is computationally expensive ($O(D)$ for each call, D is a dictionary size). To overcome the computational cost we use the trie data structure to store all the BPE tokens ($O(m)$ for each call, m is the maximum length of the BPE token in the dictionary).

3.4 Beam Search with Early Stopping

We investigated full-line code completion setting, where we try to predict a sequence of program tokens till the end of the line. We selected the average number of correct program tokens predicted as our quality metric. We found out, that if we restrict model size and use regular language modeling objective, the model starts to hallucinate after 1-2 program tokens. So we decided to restrict our inference only to 1 program token, introducing early stopping into the beam search routine.

It is easy to understand if a generated sequence is exhausted in a single token completion task. The lexer is applied to extract program tokens after each beam search iteration. If at some point the lexer output contains more than one program token, the generation process is stopped for the current sequence. We also stop the beam search if we have already obtained k complete tokens, where k is a hyperparameter. It helps to accelerate the inference and has nearly no negative effect on model quality.

3.5 Distillation

Distillation is a model compression procedure in which a student model is trained to match the outputs of a large pre-trained teacher model. Some works (Bucila et al., 2006; Hinton et al., 2015) show that distilled model can perform even better than a trained from scratch model with the same architecture and the same amount of parameters.

For distillation, we use the cross entropy loss along with the KL divergence between the student and teacher outputs (Equation 3, where p_s is a student model, p_t is a teacher model, and α is hyperparameter to balance losses).

$$q(x,t) = -(1-\alpha)\log p_s(x_t|x_{<t}) + \\ + \alpha\, KL\left(p_s(x_t|x_{<t}) \,||\, p_t(x_t|x_{<t})\right) \quad (2)$$

$$\mathcal{L}_{dist} = \sum_t q(x,t) \to \min \quad (3)$$

4 Dataset

The dataset used for the model training consists of 500k R Markdown files (Rmd). Non-code information is erased from each file and the rest of the text is transformed into a script. Additionally, in one of the experiments we use a larger dataset that contains more than 4kk with both R and Rmd files.

The evaluation dataset was collected from the Github open-source projects and consists of 35k examples from the 9k R files. There is an issue with the using of the open-source project codes for the evaluation. It is very likely for the training and the test sets to intersect. A lot of repositories have forks with minimal differences and it is very hard to distinguish them from the source one. That is why we evaluate most of our models on R files only while training on Rmd files to avoid encountering the training samples in the test set.

Some papers investigate autocompletion behaviour on real-world autocompletion logs. Aye et al. (2020) showed that autocompletion models trained as language models on an unlabelled corpus perform much worse on the real-world logs than models trained on a logs dataset initially. Hellendoorn et al. (2019) showed a difference in the distributions of the completed tokens between the real completion events and the synthetic evaluation datasets.

after_operator_$	2158
after_operator_%>%	1493
after_operator_->	43
after_operator_::	1024
after_operator_<-	3748
after_operator_=	4488
c_key_argument	598
c_positional_argument	1776
f_key_argument	8920
f_positional_argument	6730
library	748
new_line_variable	1774
new_line_function	1113
with prefix	15470
without prefix	19143

Table 1: Dataset group sizes

Not having the real logs available, we decided to divide our synthetic evaluation dataset into several groups. It is useful to validate a model behaviour on different autocompletion contexts. This way, the model can be fine-tuned to improve quality in concrete autocompletion situations, such as a package import or a function call completion. Firstly, we divide the dataset into prefix and non-prefix groups. The last program token is always incomplete in the prefix group. Also, we divide our examples into groups by the usage context. For example, there is a group with the filling of the function arguments and a group with new variables declaration.

The first type of dataset groups corresponds to completion events following the concrete operators (`$`, `%>%`, `->`, `::` `<-`, `=`). Another type covers autocompletion events during the positional or keyword arguments completion in vectors or functions. The next one consists of packages import usage contexts. The last one corresponds to the completion of a variable or a function name at the start of the new line.

5 Experiments

The code completion task may be considered a ranking problem. We use mean reciprocal rank score (MRR) and mean Recall@5 score for evaluation in our experiments. There is only one relevant element a in the autocompletion task and with search results denoted as s the formulas can be written as follows.

$$RR(a, s) = \begin{cases} i^{-1}, & \text{if } s_i = a \\ 0, & \text{if } a \notin s \end{cases}$$

$$Recall@k(a, s) = \sum_{i=1}^{k} \mathbb{I}[a = s_i]$$

5.1 Implementation Details

Our aim is to build a model light enough to run smoothly on an average laptop. We evaluate our models on a laptop equipped with Intel Core i7 with 6 cores and 16 GB RAM. The average time for the single autocompletion event should be close to 100ms and RAM consumption should not exceed 400MB.

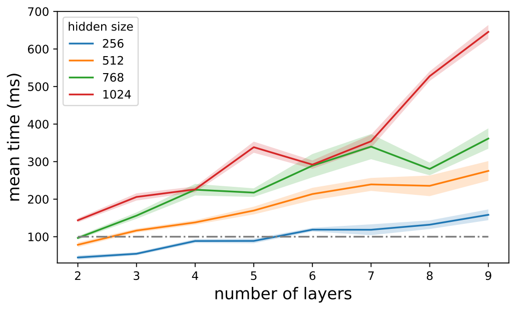

Figure 1: Mean inference time over 50k objects for different model parameters

Figure 1 presents average inference times for our model with all the proposed modifications. We keep the number of heads $= 4$ and vary hidden size and number of layers. It can be seen that the model with the hidden size $= 256$ and number of layers $= 4$ is the most complicated model that still satisfies the performance requirements.

5.2 Quality and Inference Speed

In this experiment, we evaluate each of our proposed modifications from the section 3. We apply modifications one by one and measure metrics and mean inference time for each of them. We use a transformer model with parameters from the previous experiment (hidden size = 256, heads amount = 4, number of layers = 4) as the baseline. For all experiments, we use Adam (Kingma and Ba, 2017) optimizer with the default parameters, cosine annealing learning rate scheduler (Smith and Topin, 2018) with upper learning rate boundary 5e-3 and gradient norm clipping by 10.

The results show that without the prefix generation modification the model is unable to take advantage of the given prefixes. It should be noted that almost 45% of the examples from the evaluation dataset contain unfinished tokens with a given prefix. Additional manipulations with the prefix slow down the model but it is compensated by the following two modifications. Variable name substitution during the prepossessing leads to both quality improvement and inference speed up. Generation early stopping procedure accelerates the inference without any ranking drawback.

	MRR	Recall@5	time
baseline	0.319	0.364	150 ms
+ prefix generation	0.64	0.709	195 ms
+ variable replacing	0.673	0.748	183 ms
+ BS early stopping	0.676	0.751	98 ms

Table 2: Model modifications performance

5.3 Big Dataset Effect

One of the standard methods to improve model performance in data science is to collect more data. As we mentioned before, we can not guarantee total fairness of the evaluation process in this setup, but we try to make sure that all the training examples are removed from the test set by eliminating possible duplicates.

	MRR	Recall@5
l4 s256	0.676	0.751
l6 s1024	0.683	0.751
+ more data	0.761	0.815
+ distillation	0.701	0.767

Table 3: Increasing dataset size and distillation effects

We consider multiple types of models in this experiment. The first one is the best model from experiment 5.2. The second experiment is similar to the first one but consists of six layers instead of four and has hidden size of 1024 instead of 256. The third experiment has the same architecture as the second one and is trained on a larger training set.

	w/o prefix		with prefix		both	
	MRR	Recall@5	MRR	Recall@5	MRR	Recall@5
after_operator_$	0.596	0.669	0.727	0.779	0.656	0.719
after_operator_%>%	0.566	0.714	0.840	0.895	0.702	0.804
after_operator_->	0.218	0.276	0.214	0.286	0.217	0.279
after_operator_::	0.614	0.716	0.798	0.858	0.709	0.789
after_operator_<-	0.563	0.659	0.787	0.849	0.666	0.746
after_operator_=	0.624	0.707	0.765	0.793	0.682	0.743
c_key_argument	0.659	0.706	0.796	0.824	0.719	0.758
c_positional_argument	0.752	0.820	0.815	0.846	0.778	0.831
f_key_argument	0.713	0.784	0.840	0.876	0.771	0.826
f_positional_argument	0.719	0.812	0.803	0.852	0.755	0.829
library	0.183	0.299	0.775	0.870	0.463	0.570
new_line_function	0.586	0.704	0.676	0.771	0.630	0.737
new_line_variable	0.274	0.316	0.329	0.377	0.299	0.344

Table 4: Distilled model performance on separate groups. Rows correspond to autocompletion contexts. Results for no prefix subset, prefix subset, and entire dataset are split into columns.

We apply Adaptive Softmax (Grave et al., 2017) during the first training iterations to speed up the training process. The fourth experiment is a result of distillation of the third one into the model with the architecture from the first experiment.

As we see from the results (Table 3) both increasing training set size and distillation have positive effect on the metrics. The distilled model outperforms all the models trained on a small dataset, even the more complicated ones.

5.4 Error Interpretation

Table 4 shows the distilled model performance on different parts of the evaluation dataset. In general, the additional prefix information allows achieving a higher score. Groups related to function arguments and vector content have the highest MRR score. It is an interesting observation since the vector content is eliminated during the preprocessing step. It seems that vector argument filling is very close to function argument filling semantically and the model is able to perform well in this situation without any relevant training samples.

The additional prefix information is very important for a `library` group. Library calls are usually located at the start of the program. If there is no last token prefix then the only reasonable model behaviour is to predict the most common completion.

Autocompletion usage after the <- operator means that we want to get a variable computation statement based on a variable name. In opposite, usage after the -> means that we want to get a variable name based on given computations. Corresponding groups at the table show that we are much better at the first one completion group. It makes sense as the user has no limits in the variable name design. Another reason for the low quality for the `after_operator_->` is a low amount of examples for this operator in the training data. That is why the quality for the `new_line_variable group` is better even though the task is harder.

6 Conclusions

In this work, we present a model for the R programming language completion. We introduced simple but effective techniques, which can improve a code completion quality, while not affecting the model architecture or the training objective. Thus, these techniques can be easily combined with other works in the field and any dynamic programming language. We also present an evaluation dataset for the R programming language containing different autocompletion contexts. The diversity of our dataset provides a robust estimation.

References

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2019. Structural language models for any-code generation. *CoRR*, abs/1910.00577.

Gareth Ari Aye, Seohyun Kim, and Hongyu Li. 2020. Learning autocompletion from real-world datasets.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie

Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. pages 213–222.

Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *KDD*, pages 535–541. ACM.

Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. 2017. Efficient softmax approximation for gpus.

Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When code completion fails: a case study on real-world completions. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 960–970. IEEE / ACM.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network.

Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. CTRL: A conditional transformer language model for controllable generation. *CoRR*, abs/1909.05858.

Diederik P. Kingma and Jimmy Ba. 2017. Adam: A method for stochastic optimization.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer.

Jian Li, Yue Wang, Irwin King, and Michael R. Lyu. 2017. Code completion with neural attention and pointer networks. *CoRR*, abs/1711.09573.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Nicholas McKay Shelley. 2014. Autocompletion without static typing.

Leslie N. Smith and Nicholay Topin. 2018. Super-convergence: Very fast training of neural networks using large learning rates.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 269–280, New York, NY, USA. Association for Computing Machinery.