Improving Code Completion in Pharo Using N-gram Language Models

Myroslava Romaniuk
Ukrainian Catholic University
Lviv, Ukraine
romaniuk@ucu.edu.ua

ABSTRACT

In this paper, I present applying statistical language models to improve code completion in Pharo. In particular, the goal is to use n-gram models for sorting the completion candidates and, in such a way, increase the relevancy of the suggested completions.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Pharo, code completion, IDE, statistical models, dynamically typed languages, Smalltalk

ACM Reference Format:
Myroslava Romaniuk. 2020. Improving Code Completion in Pharo Using N-gram Language Models. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3397537.3398483

1 INTRODUCTION

Code completion is one of the essential features in any IDE; it is one of the first things a developer notices, and it is something that can "make or break" the flow of productivity when coding. The accuracy and the speed with which the completions are suggested are paramount in helping make the completion as effective as possible. That is why researchers and software engineers are continuously trying to find ways to enhance completion engines in IDEs.

When it comes to code completion in the Pharo IDE[1], the situation is no different: over the past couple of years, a lot of work has already been done to improve the completion engine there. However, I believe that it can only benefit from additionally improving the sorting strategies by using machine learning techniques.

In this research, I intend to study how code completion in Pharo can be improved using statistical language models. In particular, the idea is to use the n-gram model, as it has been documented to be used for such a task [3], and, if successfully adapted to Pharo, I believe it can help enhance the quality of code completion.

2 RELATED WORKS

There are many approaches to handling the process of completing code. The classic strategies used to rely on language-specific pattern matching, i.e. completing based on the prefix the developer has typed in. According to Bruch et al. [1], code completion engines up until ten years ago would also mainly rely on type information, with no contextual analysis. In the paper, the authors countered this approach by implementing intelligent code completion systems, which learned from source code examples and gave much more contextually relevant results. In the latest years, statistical and machine learning models have also gained popularity. In machine learning approaches, source code is often treated as regular text, and the contextual analysis is disregarded. IntelliSense[2] is an example of a code completion tool that uses semantic analysis, whereas TabNine[3] is an example of a tool using deep learning.

This work has been inspired by the paper "On The Naturalness of Software" [3]. There, as well as in [2, 4, 5, 7], the structural information is not taken into consideration and the problem of improving code completion is reduced to a natural language processing problem, where predicting the next token is treated as predicting the next word in a sentence.

The n-gram language model approach proposed by Hindle et al. [3] is based on capturing statistical regularity at the n-gram level by taking n-1 previous tokens that are already entered into the text buffer, and attempting to guess the next token. Using this model, it is possible to estimate the most probable sequences of tokens and suggest the most relevant code completions to developers.

3 BACKGROUND

3.1 Code Completion in the Pharo IDE

The current implementation of code completion in the Pharo IDE is based on analysing the abstract syntax tree (AST) of source code, where we are able to determine the structure of the code and infer the type information where possible. We can then give contextually suitable completions, such as suggesting a class name for a global node, a method name for a method node, and so on. However, once we get the list of suggestions, there is no efficient way to sort them, and so the desired completions can appear at the bottom of the list. This requires the developer to scroll down or type in more symbols and can be very unproductive.

[1] Pharo is an object-oriented dynamically typed programming language inspired by Smalltalk, and the Pharo IDE is an interactive IDE intended for developing in Pharo. More at: https://pharo.org/

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3398483

[2] https://code.visualstudio.com/docs/editor/intellisense
[3] https://www.tabnine.com/

As Pharo is a dynamically typed language, type information is not always available. Not knowing the type when writing code is tricky for code completion, as it gives less precise results and more relevant options can be pushed out of view of the developer.

However, the sorting strategy chosen can significantly influence the end result. The sorter plugin in the Pharo IDE has been designed to effortlessly support adding any sorting option a developer might want to have, which is why I propose to create sorters based on unigram (n=1 in n-gram) and bigram (n=2) models.

3.2 N-gram Language Models

N-gram language models are models that assign probabilities to sequences of n words, which are called n-grams.

To compute the probability of a whole sequence, the chain rule of probability is used, which means that the probability of a sequence is a product of probabilities of individual words in that sequence, and the probability of an individual word depends on the probability of all the words before it occurring, like this:

$$P(w_1 w_2 ... w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)...P(w_n|w_1 w_2 ... w_{n-1}) \quad (1)$$

Hence, in a unigram the probability of a word occurring is just its own probability, whereas in the bigram it is the probability of the word we are interested in and the word before occurring together, and so on. This is why the decision to only implement unigram- and bigram-based sorters was made, as the frequency (the number of occurrences) table increases exponentially with every new order of n and the calculations would be too slow for such a time-sensitive task as code completion.

4 PROPOSED APPROACH

4.1 Implementation

The goal of this approach is to suggest the most relevant results via implementing sorting strategies where tokens are suggested according to their probability of appearing in the already typed sequence. This should allow us to propose suggestions more effectively even without knowing the exact type of the token.

The unigram and bigram models can be trained on source code data. In case of the unigram, the sorter can then sort the results based on individual token probabilities, which is essentially how likely is a particular token to appear based on its number of occurrences in the existing codebase. In the bigram, the individual tokens can be sorted based on the probabilities of bigram sequences occurring, wherein the sequence itself is the token being completed following the word before. Ultimately, these sorting approaches should prioritise more common tokens and sequences, which should also represent the more desired suggestions in general.

4.2 Evaluation

For evaluation, the idea is to combine quantitative and qualitative approaches. In short, a quantitative approach is based on quantifiable data and uses objective metrics, whereas a qualitative evaluation is based on setting up "live" experiments, conducting interviews, recording feedback, et cetera.

For this project, using the quantitative evaluation would mean finding a good way to reproduce the code completion process, such as, perhaps, the idea of recreating code history (to simulate the code being gradually written) to compare how accurate the results the completion gives are to what actually followed in the code history (as described by Robbes et al. [6]). The qualitative evaluation could involve several developers testing the tool as they use the Pharo IDE and recording their results. The combination of these two approaches, I believe, would quite accurately demonstrate the effectiveness of the proposed solution.

5 CONTRIBUTION AND FUTURE WORK

The estimated contributions of this work are the following:

- improve code completion in the Pharo IDE by sorting candidate completions with the help of n-gram language models
- find a way how to best tokenise code and deal with delimiters
- build a tool based on a trained n-gram model that would propose completions fast enough to be used in an IDE
- come up with an efficient methodology to numerically evaluate the results of code completion produced by different completion strategies

6 CONCLUSION

In this paper, I propose applying n-gram language models when sorting completion candidates in Pharo. Having an AST-based completion engine, we are not always able to infer type information from the nodes. I believe implementing a fast n-gram-based sorting functionality to be used in the Pharo IDE would greatly improve the relevance of suggestions.

For evaluating the results by comparing the performance of the sorting strategies implemented, I propose combining quantitative and qualitative evaluation techniques to get the most comprehensive results. That way, both the accuracy and the effectiveness of the tool in day-to-day usage can be adequately measured.

REFERENCES

[1] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. 213–222.

[2] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[3] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[4] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[5] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[6] Romain Robbes and Michele Lanza. 2008. How Program History Can Improve Code Completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*. 317–326.

[7] 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*. 269–280.