Title

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

Abstract

Code completion is one of the essential features of any IDE and it significantly improves the developer experience and productivity. Thus, it is important to find a way to make it as effective as possible.

In this work, I propose to improve code completion by sorting completions in such a way that most frequently used tokens appear first. To that end, I implement a sorting plugin based on the n-gram language model.

I use quantitative and qualitative evaluation techniques to compare the performance of the n-gram (unigram and bigram) and alphabetic sorters. As a result of the evaluation, the unigram sorter was shown to have the best performance.

ACM Reference Format:
Myroslava Romaniuk. 2020. Title. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Context

Whether it is being used for improving speed and accuracy of typing or as an API guide, helping developers find their way around libraries, code completion is one of the first things a developer notices as soon as they start coding. The speed with which the results are suggested, as well as their accuracy, is essential, and it is something that can certainly "make or break" the workflow of the developer. Therefore, researchers and software engineers are always trying to find new approaches that can improve code completion quality and make it as effective, as accurate, and as fast[1] as possible.

Throughout this project, I focused on the code completion in Pharo[2]. Pharo is a dynamically typed programming language inspired by Smalltalk, and the Pharo IDE is an interactive IDE intended for developing in Pharo.

The current code completion approach in the Pharo IDE is a semantic one: it is based on analysing the abstract syntax tree (AST) of source code. The parser finds the best suitable node for each part of code, and the completion engine suggests contextually relevant completions: class names for a global node, method calls for a method node, et cetera. The AST-based approach replaced the completion engine that had been part of the IDE since Pharo was first released in 2008; it used a less sophisticated parsing implementation and tried to infer the type by looking at kinds of messages sent to variables, and so on. Recently, a separate code completion engine that uses generators[3] was ported to Pharo as well.

1.2 Problem

As mentioned above, the current implementation of code completion in the Pharo IDE is based on the AST, which allows us to determine the structure of the code, get the semantic role of tokens, and infer their type where possible. As a result, only those completions that are correct in the given context are suggested.

Currently there is no way to effectively sort the completion candidates. The order of suggestions is significant because with a wrong sorting strategy, desired completions can appear at the bottom of the list. This is unproductive as it makes developers scroll down or type more symbols. Thus, in this work, the main goal is to come up with an effective sorting strategy that would show relevant results first.

1.3 Proposed Approach in a Nutshell

In this work, I study how code completion can be improved using statistical language models for sorting the completion candidates, on top of an existing semantic completion. In particular, I propose to use n-gram language models, as they have been documented to be used for such a task [8], and I believe this strategy has the potential to improve the relevance of sorting in the Pharo IDE.

To find the most relevant completion suggestions based on their probability of occurring in source code, I implement

[1] By "fast" we mean that the completion is quick to give suggestions, by "accurate" that the completion proposes contextually correct results, and by "effective" we mean that the combination of both the aforementioned qualities makes the tool successful in achieving the result, which is to make finding the most suitable completion as easy as possible.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[2] http://pharo.org/

[3] Its main idea is to propose code completion suggestions from pre-generated local contexts using heuristics for optimisation. More details can be found at: https://github.com/guillep/complishon

two sorting strategies based on the unigram and bigram models. These models are trained on a large corpus of source code collected from 50 projects written in Pharo [19].

To see whether the proposed implementation does indeed enhance the quality of code completion in the Pharo IDE, I evaluate the effectiveness of this approach by comparing alphabetic, unigram, and bigram sorting (with the alphabetic sorting acting as a random baseline).

1.4 Research Questions

As part of this work, I intend to answer the following research questions:

RQ1: Can we improve the accuracy of code completion in the Pharo IDE by sorting candidate completions with n-gram language models?

RQ2: How can we effectively evaluate the results of code completion enhanced by different sorting strategies?

2 Related Works

2.1 Earlier Code Completion Systems

Standard code completions in IDEs used to only rely on language-specific pattern matching, i.e. sorting completions alphabetically based on the symbols already typed in. Robbes et al. [16], however, showed that code completion could be improved by using program history (program modifications over time). They managed to get good results by prioritising suggestions from recently modified method bodies, and even better results by using per-session vocabulary (changes of the last hour) and merging it with type-based completion.

Up until 10 years ago, code completion systems were mainly based on type information, with no contextual analysis. Bruch et al. [3] countered that by implementing intelligent code completions that learned from examples and had a significantly better performance in terms of the relevance of suggestions than other then-common implementations. Their solutions included the frequency-based code completion (frequency of use of code), an association rule-based completion, and the Best Matching Neighbours code completion (method calls of the closest source snippet found, using a modified k-nearest neighbours algorithm). The BMN based implementation was integrated into Eclipse and demonstrated promising results.

2.2 Software Naturalness

Hindle et al. [8] compared source code to natural languages. They claim that code is even more repetitive, predictable and full of patterns than human languages. In the paper, they also argue that code can be modelled by statistical language models, which can be used to support software engineers. Their approach was based on capturing high-level 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. This work served as a catalyst for the following research using natural language processing (NLP) for source code.

For instance, Tu et al. [18] also learnt that code "has a high degree of localness, where identifiers (e.g. variable names) are repeated often within close distance" [1]. Thus, they applied a cache mechanism that assigns higher probability to tokens that have been observed most recently, and improved the results even further. Nguyen et al. [12] enhanced the state-of-the-art n-gram approach by incorporating semantic information into code tokens, rather than treating them as text – i.e. annotating each token with its data type and semantic role if available, which allowed them to increase predictability even further.

2.3 Deep Learning for Code Completion

In the more recent years, researchers started applying deep learning models such as deep recurrent neural networks (RNN). For instance, Hellendoorn et al. [7] recorded the results of a case study of 15,000 completions (completion events) for VisualStudio. In comparison to n-gram models, they stated that the deep learner is better at core method invocations but loses on third-party library calls, whereas the n-gram model naturally outperforms it on internal API calls but loses out on the other categories. Proksch et al. [14] worked on an extensible inference engine for intelligent code completion systems, called PatternBased Bayesian Network (PBN). Eclipse Code Recommenders project adapted the PBN approach for their intelligent call completion.

Hellendoorn et al. [6] introduced a dynamically updatable n-gram model which outperformed both the traditional n-gram models and the deep learning RNN and long short-term memory (LSTM) models. Li et al. [11] developed an attention mechanism which exploits the parent-children information of the AST of source code. As correctly predicting out-of-vocabulary (OOV) values in code completion is mainly unsuccessful, the authors implemented a pointer mixture network which either generates a new value through an RNN component or copies an OOV value from local context through a pointer component. Raychev et al. [15] implemented a tool called SLANG, which first extracts abstract histories from the data. Then, these histories are fed to a language model such as an n-gram model or recurrent neural network model, which treats the histories as sentences in a natural language and learns probabilities for each sentence, without taking into consideration the AST of the code.

2.4 Existing Code Completion Systems in IDEs

The code completion system (Pythia) [17] is part of the Intellicode extension used in the Visual Studio Code IDE to complete Python code. Pythia uses LSTM networks trained on long-range code contexts extracted from abstract syntax trees, which allows it to capture semantics carried by distant

nodes and helps rank the method and API recommendations for developers more successfully. Python is a dynamically typed language, so to use type information in Pythia they infer types at runtime based on static analysis of user patterns and add this information into the training sequence.

Asaduzzaman et al. [2] developed a tool called CSCC (context-sensitive code completion), which is an example-based completion tool, which leverages contextual information to better support method call completion. CSCC uses tokenisation instead of deep parsing to collect method call usage patterns and requires type information of the receiver object. In terms of method call recommendation accuracy, it outperformed state-of-the-art approaches, including BMN [3]. The CSCC tool is available as an Eclipse plugin for the Java Editor.

CACHECA is a cache language model-based code completion tool for Eclipse's Java editor [4]; the tool is based on the cache language model [18]. According to the authors, CACHECA greatly enhances Eclipse's built-in engine by incorporating both the corpus and locality statistics, especially when no type information is available.

2.5 Conclusion

It is worth noting that the majority of the research experiments mentioned in this chapter have been tested on or applied for only statistically typed languages, predominantly Java and C# ([8] also trained on C).

The only related works in this chapter where the experiments included dynamically typed languages were: [16] (Smalltalk, in addition to Java), [18] (Python, in addition to Java), [11] (Javascript and Python), and [17] (Python).

3 Completion in Pharo

3.1 Code Completion Background

The Pharo programming language is an object-oriented dynamically typed programming language which is inspired by Smalltalk. The Pharo IDE is a programming environment meant specifically for developing in Pharo. It consists of a virtual machine (VM), on top of which an image, serving as the current IDE workspace, can be run.

The Pharo IDE itself is written in Pharo and can be extended from within. This means that when implementing code completion, one can test it live in the very environment where one is developing it, which, if not done carefully, can lead to breaking the system.

Code completion in the Pharo IDE is called at every keystroke, as soon as two and more alphabetic characters are typed in. A completion context gets created for the text typed in until then, regardless of the type of the code editor. And there are several code editor tools within the Pharo IDE, such as the Playground, the System Browser, and the Debugger.

3.2 Typing for Completion in Dynamic Languages

As Pharo is a dynamically typed language, the precise type information is only available at runtime. Not knowing the type when writing code can make the completion suggestions less precise and push relevant options to the bottom of the list of completions, which then requires scrolling or typing more characters.

There are a couple of approaches one could take to solve this. The first is type[4] inference, or reconstruction, which can be done by extracting type information by looking at the messages sent to a variable, and merging these results with types found by heuristics applied to the right-hand side of assignment expressions [13]. Type guessing by means of name analysis can also be done, but it is more likely to be misleading. The third approach involves the AST-based completion in the Pharo IDE. It performs a semantic analysis of source code, which provides us with more accurate type information for certain kinds of nodes.

3.3 AST-Based Completion

The completion context parses the text and transforms it into the AST representation, such as a sequence of AST nodes. In the process, all the information needed for further actions is extracted: e.g. the current position of the cursor (where we want to get the completion) and the class which we are currently modifying (or information that the completion happens in the Playground, for which there is no such data). By performing the semantic analysis, we get the most suitable type of node for each part of code, and then visit each node to get the correct completion behaviour (i.e. contextually appropriate suggestions). For instance, this means that for a Global node, we want to suggest all the globals, such as class names, for a Message node we only want to get message sends to a variable, and so on.

Combining the available prefix of length at least 2, as well as relevant semantic information, we give a list of suitable completion suggestions that are then passed to the sorter. The list itself is displayed in a completion menu that pops up once the completion is called and then is updated with every new keystroke, unless the developer cancels it by pressing *Esc* or clicking outside of the text area. The completion window can also disappear once there are no valid suggestions to give anymore.

3.4 Sorter Plugin

The sorter plugin is an important part of the code completion tool in the Pharo IDE. It is responsible for the final order in which the completions are presented. Within the sorter, we treat the completion implementation as a black box. The only information it receives is the list of completions to be

[4] *Type* meaning a kind of variable, instead of an act of writing something by pressing the keys. Throughout this text the word is often used in either one of the meanings, depending on context.

sorted, as well as the completion context (meaning any other completion-related information, such as the AST node, the cursor position, the class where the completion happens, and so on). Ultimately, this means that the way we get the completion results is the same every time (i.e. we get contextually suitable completions as a result of analysing the AST). However, the sorting strategy vastly influences the end result (i.e. the list of suggestions displayed in the pop-up completion menu) the users (developers) see. Specifically, a good sorter can prioritise the more relevant completions that would otherwise be positioned far down the list (for example, if the type of the token being completed is not known or there is no local context, as happens in the Playground).

4 N-gram Background

4.1 N-gram Language Models

Predicting the next word in a sequence is a central problem for many areas of Natural Language Processing (NLP), including speech recognition, spelling correction, spam filtering, machine translation, and others.

Models that assign probabilities to sentences or sequences of words, and can be used to find the most likely continuation of a sequence, are called language models ([9]). Among them, is the n-gram language model, which assigns probabilities to sequences out of n words, called the n-grams. A one-word sequence is a unigram, pairs of words are referred to as 2-grams or bigrams, 3-grams or trigrams are sequences of three words, and so on. Examples of bigrams might be "he ate", "ate the", "the whole" and "whole pizza", whereas trigrams would look like "he ate the", "ate the whole", and "the whole pizza".

To describe the conditional probability of a word w based on history h, we use the following notation:

$$P(w|h) \tag{1}$$

Here is a more concrete example: if we have a sentence "he ate the whole pizza", and want to compute the probability of the word "pizza" given that the previous words are "he ate the whole", we can express it as a conditional probability:

$$P(\text{pizza}|\text{he ate the whole}) \tag{2}$$

To estimate this probability, we can use relative frequency: we need to compute the number of occurrences of "he ate the whole", as well the number of occurrences of the sentence "he ate the whole pizza", and divide the latter by the former:

$$P(\text{pizza}|\text{he ate the whole}) = \frac{C(\text{he ate the whole pizza})}{C(\text{he ate the whole})} \tag{3}$$

To compute the probability of a whole sequence, not just one word, we can use the chain rule of probability:

$$P(w_1 w_2 ... w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)...P(w_n|w_1 w_2 w_3 ... w_{n-1}) \tag{4}$$

which means that the probability of the first word occurring is just its own probability, then the probability of the second word is its probability of occurring given that the first word occurred, et cetera; then the joint probability of the whole sequence is the product of each words' probability.

It is worth noting that n can be quite large, and so in a sizeable data corpora it can be a challenge to estimate the probabilities of large sequences. However, there is an assumption that to compute the probability of a word inside the sequence, the whole history is not needed, and we can approximate the probability only relying on the few history words (*n-1*) that are the closest to the word whose probability we are estimating. This is called a Markov assumption, and it is believed to hold true for n-grams. In other words, if we consider a bigram model, this means that we can reduce our conditional probabilities as follows:

$$P(\text{pizza}|\text{he ate the whole}) \sim P(\text{pizza}|\text{whole}) \tag{5}$$

where, in the case of a bigram model, we only consider one previous word, in the case of a trigram, two previous words, and so on.

From this, the joint probability of a sequence (see Equation 4) can be approximated the following way:

$$P(w_1^n) \approx \prod_{x=1}^{n} P(w_x|w_{x-1}) \tag{6}$$

4.2 Unknown Words and Smoothing

A notable problem for using n-gram models is data sparsity. It can happen that the words that we encounter in the test data were not present in the training data, and so are unknown to our model. Thus, the probability of them occurring would be zero. In that case, the joint probability of the whole sequence containing such a word would also be zero, which can lead to the model discarding perfectly valid and commonly encountered sequences, and might hurt the model performance.

One way of dealing with this is to replace, for example, the bottom n% of the training data with the unknown word token <UNK>, and then replace every unknown word in the test data with <UNK> as well, turning it into a known word with a probability above zero.

An alternative solution to the problem of unknown words is smoothing, which involves redistributing the general probabilities of all the words in the dataset, i.e. giving a bit of probability of more frequently encountered words to the unknown ones.

There are several smoothing algorithms, the simplest of which is Laplace smoothing. In this algorithm, we add one to all the counts (so all the counts are increased by one, and there are no zero counts), and then add $|V|$ (the size of vocabulary, i.e. number of unique words in it) to the denominator, to take into account the extra observations. So the smoothed

unigram probability of a word w looks the following:

$$P(w) = \frac{c+1}{N+V} \tag{7}$$

where c is the count of the word w and N is the total number of words.

5 Proposed Solution

5.1 Solution Overview

The objective of this implementation is the following: suggest the most likely tokens at every step of the completion process. The main idea is to leave the existing, AST-based code completion engine in place, and enhance the sorting strategy. For this, I implemented two separate sorters based on n-gram models. That means that after the list of completion candidates is proposed, it gets sorted based on each n-gram's probability so that the most relevant completions are shown first.

The n-gram models implemented were the unigram and the bigram. N-gram models of a higher order, such as 3- and 4-gram models, were not considered due to their computational complexity: the frequency table increases exponentially with every new order of n, which makes the calculation of probabilities too slow for the kind of task where results must be updated at every keystroke (i.e. code completion).

The first implementation, unigram-based sorting, is based on the 1-gram analysis, which means that we only take into consideration the actual token being completed. The implementation of the unigram model itself comes down to calculating individual token frequencies, i.e. the number of occurrences of each token in the source code. Then the completion candidates are sorted according to each one's frequency.

The second, more advanced implementation, is the bigram sorting strategy. As an n-gram model where n=2, it relies on both the token currently being completed and the history – in this case, one token before the current one. To get the history word, I take the source code that is currently being edited (which is known to the context) and find the token preceding the one we are currently completing. After that, once the history word and each completion candidate are assembled into respective pairs of tokens, I calculate the probability of each of the n-gram sequences. Then the joint sequence probabilities are used for sorting each completion candidate (second word of the sequence).

5.2 Implementation Details

Before implementing the sorting strategies, I needed to train the n-gram models. The first step was to get the dataset. For this, I retrieved the Pharo source code, which was collected[5] by [19] for their research on characterising Pharo code. The data came from 50 projects, which consisted of 824 packages, 13,935 classes, and 151,717 methods. In particular, I used the dataset where the source code was already split into tokens and respective token types for each method. Among the regular alphabetic tokens, the delimiters and non-alphabetic tokens were included as well. For example, a comment such as '"Here is a comment for this method"' or a delimiter '.' would be considered separate tokens, and their respective token types would be 'COM' and 'DOT'.

Due to a large number of tokens in the dataset, I needed to be mindful of potential time constraints, as the lookup of n-gram probabilities used for sorting had to be fast enough to not make the developer pause and wait for it. Throughout the experiment, I came up with various ways to additionally reduce the dataset. Each of those attempts is described in Section **??**.

The bigram model trained on source code data was implemented with the help of the NgramModel library[6] available in Pharo. To calculate the probabilities, I created a bigram model, trained it on the source code tokens, cut off all the sequences with counts less than 50, and returned the result. Afterwards, the model was written to file (and then retrieved from file when I used it for sorting the results) – this was done for speed purposes.

5.3 Engineering Details

Before training the models, I additionally cleaned the data by deleting some rows with token and token type mismatch, eliminating double tabs and replacing them with placeholders, and splitting token and token types into separate columns.

After that, when recording token frequencies for the unigram model, I recorded the results to file, for faster lookup for the sorting. However, even with that approach, there was a problem: as the initial number of tokens was around 150k, the lookup was slow enough to be noticeable and break the developer's flow when typing (results would be read from file when the completion is called, and it takes 3 and a half minutes to call the *readFile* a hundred times).

The approach to improve the time efficiency was to set a certain threshold for the number of occurrences and to cut off all the tokens with frequencies below it. I put the threshold equal to 10, meaning if the token occurred less than 10 times throughout the whole history, it was irrelevant enough to be discarded. This way, I was able to cut off most of the miscellaneous, rarely encountered tokens, and shortened the dataset from 150k to just 16k entries. This made the frequencies lookup during completion sorting very fast and, as a result, it was now possible to type and use completion information without any delays.

The one-time tokens, such as custom strings and comments, did not make a big difference during the implementation of the unigram sorting strategy. However, when I moved on to implementing the bigram sorter, the model became

[5] https://gitlab.inria.fr/rmod-public/2019-sourcecodedata

[6] https://github.com/pharo-ai/NgramModel

"too heavy" due to many combinations of such tokens, and needed to be additionally reduced.

The solution was to go back to the data processing step and replace those "one-time" tokens with placeholders. In particular, I added placeholders for strings, comments, symbols, characters, arrays and numbers (such as all strings being written as <str> and all numbers as <num>). This significantly reduced both the number of total tokens and the subsequent n-gram sequences, which helped both the bigram and the unigram model, as well as the lookup speed (for example, for the unigram model even without the threshold cut-off, with the placeholder replacement the number of tokens was reduced from 150k to just 85k, and for the bigram model the vocabulary size also got reduced in half).

5.4 Conclusion

Both the n-gram sorters can actually be used in the Pharo IDE by loading the CompletionSorting library[7], which contains the implementation of this experiment. The unigram-based sorter is available as the FrequencyCompletionSorter, and the bigram-based as the BigramCompletionSorter – they can be used by enabling the respective option in the Settings.

The unigram sorter is quite fast and accurate. The bigram sorter is still too slow for comfortable usage, and improving the implementation is left for future work.

6 Evaluation

6.1 Evaluation Overview

Evaluation is an essential part of any research process. When it comes to scientific research, there are two main types of evaluation one can perform: quantitative and qualitative.

Quantitative, as can be guessed from its name, refers to the type of an evaluation experiment that is based on quantifiable data and uses objective metrics. This is the predominant evaluation approach in the natural sciences and computer science, i.e. measuring physical parameters in the case of the former, or testing performance scores and accuracy in the latter. Qualitative methods, more native to fields such as sociology or psychology, are now being increasingly used in the evaluation of computer systems and software tooling [5]. The main objective is to get the "feel" for the system and deeper explore the human interaction experience, with the data gathered primarily from observations and interviews and subsequently analysed.

Conventionally, a proper qualitative evaluation [10] requires an extensive set-up, involving multiple people, recording their coding process, conducting feedback interviews, and so on. However, due to lack of time, the qualitative evaluation I conducted for this project is more minimal. Mainly, it involves manually selecting a number of common use cases and interacting with them as a user normally would, while recording the results and their meaning.

[7] https://github.com/myroslavarm/CompletionSorting

6.2 Challenges in Evaluating Completion

Coming up with a way to most accurately evaluate code completion is not trivial. One needs to have an idea of how to best reproduce the process and what the metrics should be, as well as have access to the resources needed. Mainly, here are the main challenges that need to be considered when conducting an evaluation:

1. Resources limitation. For benchmarking, there needs to be a large enough dataset to be able to evaluate the performance of the completion tool properly. It requires time to gather the data if it is not already available. If it already is and it is too big, then monetary and physical resources come into play: one might then need better and faster hardware, more machines, et cetera. Finally, one needs time and people to develop a suitable evaluation tool and set everything up well. For a qualitative evaluation, involving multiple people and managing time and money right is a concern, too.
2. Finding appropriate metrics. Ultimately, the goal of the evaluation is to be able to say whether the completion performs well or badly, and just *how* well or *how* badly. Choosing metrics which will adequately represent the quality of the performance takes a lot of careful analysis and research. Getting this part right considerably influences the validity of the end result of the evaluation.
3. Reproducibility. Less a problem for a qualitative approach, where the idea is to let people use the tool as is and report the results, for the quantitative evaluation it is important to come up with a way to make code completion testable, or reproducible. This brings us back to the resources, as a suitable dataset is needed.

Finally, this is all a matter of correctly managing the pros and cons of the evaluation. There are multiple ways to perform it, and multiple reasons to do something one way or another. For this project both the quantitative and the qualitative approaches were used, as the first provides concise numeric results of the completion sorting performance, and the second allows to analyse common cases and demonstrates the actual human-computer interaction as it would happen with regular users (developers using the Pharo IDE). The next two sections detail the experiments and their results.

6.3 Quantitative Evaluation

The quantitative evaluation approach performed for this project is based on the paper by Robbes et al. [16], wherein they used the change history data to evaluate the performance of their completion.

The idea with this approach is to recreate the coding process as it would happen naturally, i.e. a gradual appearance of new code as if it is being entered at the moment, and test the completion during that. Just as if a developer is typing and invoking a code completion engine at every step, with

this approach the completion gets triggered for every token with the cursor between the 2nd and the 8th position (the range is also from [16]). We can then compare whether the result suggested at that step matches the one that followed in real history.

For evaluation, several Pharo classes were chosen at random, each containing multiple methods. For each of them, I repeat this process of calling the completion at each step and perform the following:

- check whether the correct completion is present in the list of suggestions
- if it is, the results are only considered successful if the correct completion is also present in the top-10 suggestions
- for those cases I record the current sequence (i.e. what has been "typed in" up to that point), prefix length, the actual correct match, and its position in the list of suggestions
- I also calculate the accuracy (average success rate) for all the methods evaluated (i.e. out of all the attempts to complete the history, how many of them had the expected match in the first 10 results)

The three models I am comparing are the unigram and bigram sorters that were implemented as part of this work, and the alphabetic sorter, which is the existing default sorter of the Pharo IDE completion. The alphabetic sorter, as can be guessed from its name, simply sorts the completion suggestions alphabetically. Here we use it as a comparison baseline, as the idea is to find out whether indeed the n-gram based sorting performs better and improves the relevancy of code completion suggestions, in comparison to the alphabetic sorter (and the unigram and the bigram strategies tested against each other).

Table 1 presents the accuracy (average success rate) for each of the classes and sorting strategies tested.

Class	Alphabetic	Unigram	Bigram
OrderedCollection	0.32	0.38	0.30
CompletionEvaluation	0.24	0.25	0.22
FrequencyCompletionSorter	0.48	0.58	0.47
NgramModel	0.32	0.36	0.29

Table 1. Quantitative evaluation: the average success ratio (accuracy) for each class and sorting strategy

From the results, we can see that the unigram sorter performs better than the alphabetic sorter, as expected. However, the bigram sorter performs both worse than the unigram sorter and even slightly worse than the alphabetic one.

6.4 Qualitative Evaluation

For this evaluation approach, 10 examples were manually preselected. The main idea for each of the examples was to test a completion case which would be commonly encountered by a developer while coding in the Pharo IDE. These examples include completion of message sends, method names, variable names, expressions inside a block, expressions in parentheses, multiple message sends, and so on. To emphasise: each of those examples is of interest because they are indicative of a common coding process in Pharo, and that is the sole reason they were chosen: none of the examples were chosen with any bias towards the sorting strategies.

All the results are only recorded after two symbols were typed in, and having correct (most relevant) suggestions with such a short prefix appear among the top 10 results can be considered best-case scenario (as the correct suggestions would be more likely to be positioned higher with a longer prefix). Ergo, when the desired suggestion is missing from the top 10, it does not mean a complete failure of the completion engine or a sorting strategy used but rather hints that there is more work required to get to the correct suggestions, such as scrolling or typing more characters.

Note: if the *completion suggestion* and *position* columns contain '-' for any sorting strategy, it means that the completion suggestions expected for that code example were not in any of the top 10 positions of the suggestions list.

Example 1:

```
col := OrderedCollection new.
col ad
```

This was typed into the Playground and the results were recorded with the cursor directly after *ad*. For this context, the completion suggestions a developer would expect would be *add:* and *addAll:*. Here are the actual results:

Sorter Type	Completion Suggestion	Position
Alphabetic	-	-
Unigram	add:	1
	addAll:	2
Bigram	-	-

Table 2. Qualitative evaluation: the results of completing a message send in the Playground

As we can see in the Table 2, both the alphabetic and the bigram sorters did not manage to provide any relevant results in the top 10 suggestions, whereas the result from the unigram turned out exactly as expected.

Example 2:

```
something := 1.
(something = 1) if
```

Another example coded in the Playground, with the cursor at *if*. The expected results would be *ifTrue:*, *ifTrue:ifFalse:*, and *ifFalse:*.

Sorter Type	Completion Suggestion	Position
Alphabetic	-	-
Unigram	ifTrue:	1
	ifTrue:ifFalse:	2
	ifFalse:	3
Bigram	ifTrue:	5

Table 3. Qualitative evaluation: completing a message send to an expression in parentheses

In the Table 3 we can see that the alphabetic sorter once again did not suggest anything of interest, the unigram demonstrated an exemplary performance, and the bigram, while less than ideal, still suggested one of the correct options in the top 10 position (albeit a bit lower, which would require either typing more symbols or pressing the down arrow key to get to).

To briefly sum up the results of the manual qualitative evaluation, the unigram sorter still seems to be the best one out of the three, the same as in the quantitative evaluation. However, as seen in these examples, the bigram sorter is not very far behind. Thus, as part of the future work, it would be useful to investigate what exactly are the cases decreasing the bigram sorter's average accuracy, or, in other words, to find out what is the issue and what can be done to fix it.

7 Conclusion

In this work, I proposed a machine learning-based technique for improving the completion engine in the Pharo IDE. The exact implementation uses the n-gram language models to sort the completion candidates that are suggested to the user as they type. Specifically, I implemented two sorting strategies: one based on the unigram model and another on the bigram model.

Based on the evaluations conducted and the actual usage of the implemented sorters in the Pharo IDE, one of the implementations, the unigram sorter, has been shown to be: (1) fast, which means that it can be comfortably used by Pharo developers when coding, and (2) effective, which means it gives significantly better, or more relevant, results than the default Pharo IDE sorter.

Therefore, we can conclude that the main goal, improving the Pharo code completion with the help of an n-gram based sorting implementation, has been achieved.

7.1 Discoveries

Through completing this work, I am now able to answer the research questions that were stated at the beginning:

RQ1: **Can we improve the accuracy of code completion in the Pharo IDE by sorting candidate completions with n-gram language models?** As a result of the evaluation performed, the unigram based sorter has been shown to have a significantly better result than the default sorter in the Pharo IDE. The implementation is also fast enough for comfortable developer usage and is available by loading the CompletionSorting library available at https://github.com/myroslavarm/CompletionSorting.

RQ2: **How can we effectively evaluate the results of code completion enhanced by different sorting strategies?** For the quantitative evaluation, inspired by [16], I simulated the completion process as it would happen naturally, by generating source code sequences at various stages of typing, and comparing the results proposed to the ones in the codebase.
The qualitative approach allowed me to experimentally test the suggestions each sorting strategy proposes by using the tool as it would be normally used by a developer, and compare the results first-hand.

7.2 Directions of Future Work

As can be seen in the previous section, the bigram sorter did not perform as well as expected. Contrary to the intuition that the higher order of n-gram should work better, the bigram performed much worse than the unigram. Additionally, it also seemed to give less relevant suggestions than the alphabetic sorter. Thus, it would be useful to analyse the issues of the existing bigram implementation and try to fix them and improve the performance of the bigram-based sorter.

For the quantitative evaluation, I want to evaluate the results more precisely: instead of an average accuracy, perhaps a more sophisticated formula could be used. For instance, the idea by Robbes et al. [16] involves making a note of the exact positions and prefix lengths and ranking completions, which give more relevant results for shorter prefixes, higher. For the qualitative approach, a more extensive case study with multiple participants testing the actual completion in the IDE and reporting their feedback would be a good next step towards a more thorough evaluation of the results.

References

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. 2014. Context-sensitive code completion tool for better api usability. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 621–624.

[3] 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.

[4] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 705–708.

[5] Orit Hazzan, Yael Dubinsky, Larisa Eidelman, Victoria Sakhnini, and Mariana Teif. 2006. Qualitative research in computer science education. *Acm Sigcse Bulletin* 38, 1 (2006), 408–412.

[6] 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.

[7] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 960–970.

[8] 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.

[9] Daniel Jurafsky and James H. Martin. 2009. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc.

[10] Bonnie Kaplan and Joseph A Maxwell. 2005. Qualitative research methods for evaluating computer information systems. In *Evaluating the organizational impact of healthcare information systems*. Springer, 30–55.

[11] 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).

[12] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 532–542.

[13] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. 2009. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/1640134.1640145

[14] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent Code Completion with Bayesian Networks. *Transactions on Software Engineering and Methodology (TOSEM)* 1, 25 (2015). https://doi.org/10.1145/2744200

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

[16] 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.

[17] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735. https://doi.org/10.1145/3292500.3330699

[18] 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.

[19] Oleksandr Zaitsev, Stéphane Ducasse, and Nicolas Anquetil. 2020. *Characterizing Pharo Code: A Technical Report*. Technical Report. Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla. https://hal.inria.fr/hal-02440055