

UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

---

# Improving Code Completion in Pharo Using N-gram Language Models

---

*Author:*  
Myroslava ROMANIUK

*Supervisors:*  
Oleksandr ZAITSEV  
Dr. Marcus DENKER

*The thesis submitted in fulfillment of the requirements  
for the degree of Bachelor of Science*

*in the*

Department of Computer Sciences  
Faculty of Applied Sciences



APPLIED  
SCIENCES  
FACULTY ●

Lviv 2020

## Declaration of Authorship

I, Myroslava ROMANIUK, declare that this thesis titled, “Improving Code Completion in Pharo Using N-gram Language Models” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

---

Date:

---

“Please do not be cynical. I hate cynicism. For the record, it’s my least favorite quality. It doesn’t lead anywhere. No one in life gets exactly what they thought they were going to get. But if you work really hard and you’re kind, amazing things will happen.”

Conan O’Brien

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

**Improving Code Completion in Pharo Using N-gram Language Models**

by Myroslava ROMANIUK

## *Abstract*

Code completion is one of the essential features of any IDE and it significantly improves the developer experience and productivity. Good code completion can both speed up the development process, as well as aid the developer in API exploration. On the other hand, having a slow or inaccurate completion can be very cumbersome. Thus, it is important to find a way to make it as effective as possible.

The current implementation of code completion in the Pharo IDE is based on the abstract syntax tree (AST) of source code. The AST allows us to learn the semantic role and the kind of tokens (e.g. class name, method name, literal, et cetera) and to suggest contextually relevant completions. However, the current implementation has no efficient way to sort the completion candidates, which means that sometimes the user must scroll through a long list of proposed completions to find the one that they need.

In this work, I study how code completion can be improved 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.

The sorting strategies implemented are part of a Pharo IDE plugin. It is open source and is available at <https://github.com/myroslavarm/CompletionSorting>.

## *Acknowledgements*

I want to express my endless gratitude to Oleksandr Zaitsev and Dr. Marcus Denker from the RMoD team at INRIA Lille for supervising this work, providing me with valuable feedback, and patiently answering my questions. I also want to thank Dr. Stéphane Ducasse for helping me arrange my internship at INRIA and my stay in Lille, as well as introducing me to Pharo during my first year of university and motivating me to participate in open source development. I am also grateful to INRIA Lille and the European Smalltalk User Group (ESUG) for financing this internship.

I would also like to thank Dr. Yuriy Tymchuk for helping me make my first steps in Pharo during Google Summer of Code (GSoC) 2017, Julien Delplanque and Dr. Guillermo Polito from the RMoD team at INRIA Lille for giving me advice and assisting me throughout various stages of my work on code completion, and Dr. Oles Hodych for providing me with recommendations for writing a thesis and with various  $\text{\LaTeX}$  tips. I am also grateful to my mum and dad for supporting me throughout my Bachelor journey, to my best friends Martin and Ryan for motivating me and keeping me grounded, and to Conan O'Brien and Tina Fey for providing endless comedic relief when it was most needed – such as writing this thesis in social isolation during the COVID-19 quarantine in a student room in Lille, France.

Finally, I want to thank the Ukrainian Catholic University and the Faculty of Applied Sciences for their continued support throughout my studies and this internship, particularly Oles Doboševych and Dr. Yaroslav Prytula for their patience, kindness, and belief in me. It has really been an honour to be a part of this program.

# Contents

<b>Declaration of Authorship</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	1
1.2 Problem . . . . .	2
1.3 Proposed Approach in a Nutshell . . . . .	2
1.3.1 Research Questions . . . . .	2
1.4 Contributions . . . . .	2
1.5 Structure of the Thesis . . . . .	3
<b>2 Related Works</b>	<b>4</b>
2.1 Earlier Code Completion Systems . . . . .	4
2.2 Software Naturalness . . . . .	4
2.3 Deep Learning for Code Completion . . . . .	5
2.4 Existing Code Completion Systems in IDEs . . . . .	5
2.5 Summary . . . . .	6
<b>3 Completion in Pharo</b>	<b>7</b>
3.1 Code Completion Background . . . . .	7
3.2 Typing for Completion in Dynamic Languages . . . . .	7
3.3 AST-Based Completion . . . . .	9
3.4 Sorter Plugin . . . . .	9
3.5 Summary . . . . .	9
<b>4 N-gram Background</b>	<b>11</b>
4.1 N-gram Language Models . . . . .	11
4.2 Unknown Words and Smoothing . . . . .	12
4.3 Summary . . . . .	13
<b>5 Proposed Solution</b>	<b>14</b>
5.1 Solution Overview . . . . .	14
5.1.1 Unigram Sorting . . . . .	14
5.1.2 Bigram Sorting . . . . .	15
5.2 Implementation Details . . . . .	15
5.2.1 Data Preparation . . . . .	15
5.2.2 N-gram Library Extension . . . . .	16
5.3 Engineering Details . . . . .	16
5.4 Using the N-gram Sorters . . . . .	17
5.5 Summary . . . . .	18

<b>6</b>	<b>Evaluation</b>	<b>19</b>
6.1	Evaluation Overview	19
6.2	Challenges in Evaluating Completion	20
6.3	Quantitative Evaluation	20
6.3.1	Experiment	20
6.3.2	Results	21
6.4	Qualitative Evaluation	22
6.4.1	Experiment	22
6.4.2	Results	22
6.5	Summary	27
<b>7</b>	<b>Conclusion</b>	<b>28</b>
7.1	Discoveries	28
7.2	Directions of Future Work	29
7.2.1	Enhancing the Bigram Sorter	29
7.2.2	Conducting a More Extensive Evaluation	29
	<b>Bibliography</b>	<b>30</b>

# List of Abbreviations

<b>API</b>	<b>Application Programming Interface</b>
<b>AST</b>	<b>Abstract Syntax Tree</b>
<b>IDE</b>	<b>Integrated Development Environment</b>
<b>LSTM</b>	<b>Long Short-Term Memory</b>
<b>ML</b>	<b>Machine Learning</b>
<b>NLP</b>	<b>Natural Language Processing</b>
<b>RNN</b>	<b>Recurrent Neural Network</b>
<b>RQ</b>	<b>Research Question</b>



## Chapter 1

# Introduction

### 1.1 Context

Code completion is one of the most used features in any IDE. 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<sup>1</sup> as possible.

The main distinction among the many approaches to code completion would be whether the approach uses semantic context or not. Namely, semantic models are models that use code structure analysis to tailor completions to type and/or semantic role, or, in other words, to give contextually correct and relevant results. Machine learning models can also employ semantic analysis but often treat source code as plain text, with the context not taken into consideration.

Throughout this project, I focused on the code completion in Pharo<sup>2</sup>. Pharo is a dynamically typed programming language inspired by Smalltalk, and the Pharo IDE is an interactive IDE intended for developing in Pharo<sup>3</sup>.

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<sup>4</sup> was ported to Pharo as well.

---

<sup>1</sup>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.

<sup>2</sup><http://pharo.org/>

<sup>3</sup>In the rest of this thesis, I use the word "Pharo" to refer to the programming language, and "Pharo IDE" to refer to the development environment.

<sup>4</sup>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>

## 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.

However, 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 requires the developer to scroll down or type more symbols.

In this project, 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 (Hindle et al., 2012), 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 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 (Zaitsev, Ducasse, and Anquetil, 2020).

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).

### 1.3.1 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?

## 1.4 Contributions

Throughout this project I made the following contributions:

- Enhanced the NgramModel<sup>5</sup> library, which is an open-source library that implements n-gram language models in Pharo (4 accepted pull requests).
- Trained several n-gram language models on the tokenised source code of the Pharo programming language.

---

<sup>5</sup><https://github.com/pharo-ai/NgramModel>

- Used those trained models to build a sorting tool that enhances code completion in the Pharo IDE by sorting the proposed suggestions according to their relevance in the given context.
- Made one of the two implemented sorters both effective and fast enough to actually be suitable for developer use. Both implementations can be used in the Pharo IDE after being loaded from the CompletionSorting GitHub repository (<https://github.com/myroslavarm/CompletionSorting>).
- Proposed a combination of two approaches for evaluating the results of code completion: (1) a quantitative evaluation technique inspired by Robbes and Lanza, 2008 and (2) a qualitative evaluation where I perform a case study of several completion scenarios and analyse the results.
- Part of this work has been accepted for the Student Research Competition at <Programming> 2020. The conference was supposed to take place in Porto, Portugal in March and was cancelled due to COVID-19. Despite that, the abstract is still up for publication in the ACM Digital Library.
- To my knowledge, this is the first implementation of a code completion engine in Smalltalk family of IDEs that uses machine learning for sorting completion candidates.

## 1.5 Structure of the Thesis

### Chapter 2. Related Works

Here I give a detailed overview of the research done around code completion in the last decade and a half. It includes both the solely software engineering approaches predating the use of machine learning for this task, as well as exclusively machine learning experiments of the recent years.

### Chapter 3. Completion in Pharo

In this chapter, I go over the implementation details behind the current completion engine in the Pharo IDE, challenges of code completion for dynamically typed languages, as well as describe the idea behind the sorter plugin.

### Chapter 4. N-gram Background

Here I provide a theoretical background needed to understand this work, which includes the introduction to n-gram language models and challenges of their implementation.

### Chapter 5. Proposed Solution

This chapter contains a detailed description of the final solution: data preparation, approaches taken to implement the n-gram sorters, and various engineering details.

### Chapter 6. Evaluation

In this chapter, I go into depth about evaluation techniques and challenges, as well as detail the evaluation of the approaches taken.

### Chapter 7. Conclusion

This chapter contains a summary of the results of this work and the shape it can take in the future.

## Chapter 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 and Lanza, 2008, 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.

According to Bruch, Monperrus, and Mezini, 2009, up until 10 years ago, code completion systems were mainly based on type information, with no contextual analysis. The authors 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), which was the main contribution of their paper. The BMN based implementation was integrated into Eclipse and demonstrated promising results. It was later extended by Proksch, Lerch, and Mezini, 2015 (see 2.3).

### 2.2 Software Naturalness

In the paper "On The Naturalness of Software", Hindle et al., 2012 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, Su, and Devanbu, 2014 also learnt that code "has a high degree of localness, where identifiers (e.g. variable names) are repeated often within close distance" (Allamanis et al., 2018). 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., 2013 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., 2019 recorded the results of a case study of 15,000 completions (completion events) for VisualStudio. One of the conclusions they reached is that even though RNNs often outperform n-gram models in typical natural language settings, n-gram models are sometimes a better choice for modelling source code. For example, they state 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, Lerch, and Mezini, 2015 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. They also tested (evaluating quality, speed and model size) Best Matching Neighbour algorithm, using additional context information for more precise recommendations, and applying clustering techniques to improve model sizes. They conclude that showing the developer hundreds of recommendations may be as ineffective as showing none, and intelligent code completions better target the needs of developers that are unfamiliar with an API.

Hellendoorn and Devanbu, 2017 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., 2017 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, Vechev, and Yahav, 2014 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) described by Svyatkovskiy et al., 2019 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., 2014 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. The CSCC tool is available as an Eclipse plugin for the Java Editor. Execution time-wise the tool showed the results roughly equal to the state-of-the-art. However, in terms of method call recommendation accuracy, it outperformed those approaches, including BMN (the Best Matching Neighbours implementation from Bruch, Monperrus, and Mezini, 2009).

CACHECA is a cache language model-based code completion tool for Eclipse's Java editor which was presented by Franks et al., 2015; the tool is based on the cache language model described by Tu, Su, and Devanbu, 2014 that was mentioned earlier in this chapter. According to Franks et al., 2015, 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 Summary

Here is a brief overview of the progress made in the area of code completion research and development in recent years:

- Code completion tools used to only rely on language-specific pattern matching (i.e. by prefix), but around 10 years ago, the idea of "intelligent" code completion that can learn from examples was popularised.
- The software engineering approaches following this idea relied on the frequency of use of code, contextual association, and hierarchical proximity.
- From another perspective, source code was likened to natural languages because of its repetitiveness and patterns; the idea that source code can be modelled by statistical language models attracted more machine learning approaches for the task of improving code completion.
- Following that, many code completion implementations using machine learning ignored such information as type and semantic meaning, and treated it as regular text, trying to infer relevant predictions from source code history alone.
- These days, code completion systems that are actually being used by developers (such as plugins for IDEs) mostly combine both the software engineering approach (taking advantage of semantic and contextual information) and the machine learning one (taking advantage of source code patterns).

Another thing worth noting is 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# (Hindle et al., 2012 also trained on C).

The only related works in this chapter where the experiments included dynamically typed languages were: Robbes and Lanza, 2008 (Smalltalk, in addition to Java), Tu, Su, and Devanbu, 2014 (Python, in addition to Java), Li et al., 2017 (Javascript and Python), and Svyatkovskiy et al., 2019 (Python).

## Chapter 3

# Completion in Pharo

In this chapter, I give a detailed overview of the current completion engine in the Pharo IDE, as well as describe the challenges of code completion for a dynamically typed language, and the idea behind the sorter plugin.

### 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, which is a tool for generating small scripts and sketching out some code
- the System Browser, a tool which allows one to browse classes and methods and has a dedicated code area for writing and editing code
- the Debugger, a specialised code editor for editing code during debugging

(Figures 3.1 and 3.2 demonstrate the way the completion menu looks in the Playground and the System Browser).

### 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.

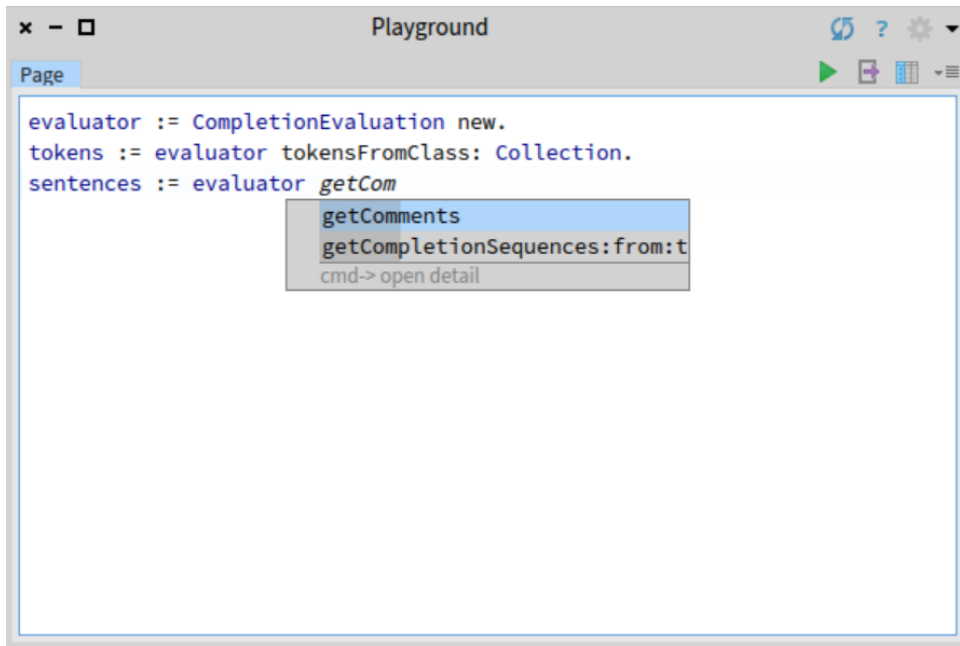


FIGURE 3.1: Completion in the Playground

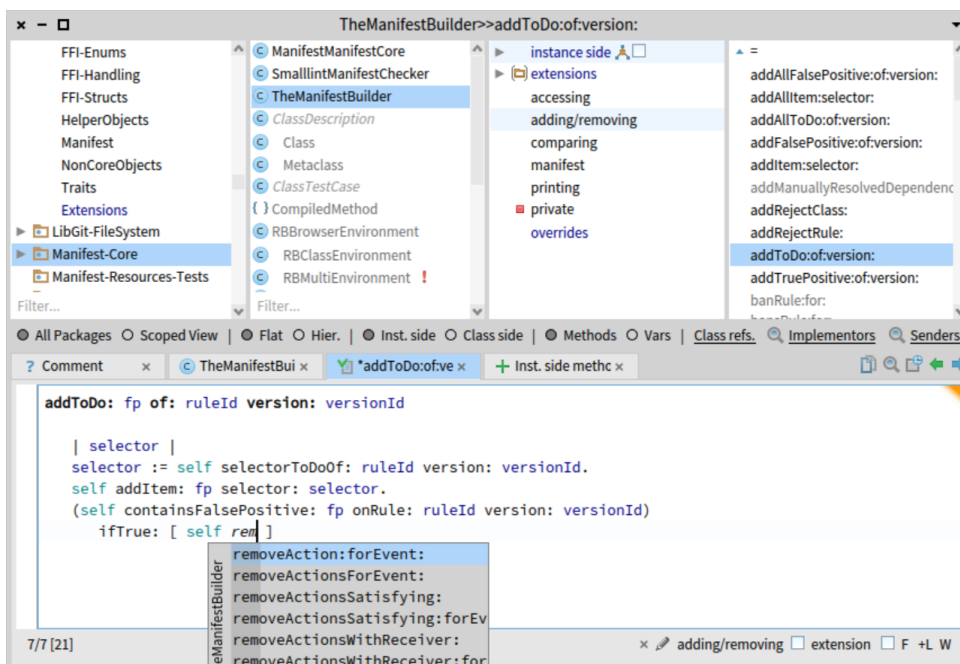


FIGURE 3.2: Completion in the System Browser

There are a couple of approaches one could take to solve this. The first is type<sup>1</sup> inference (or rather type 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 (Pluquet, Marot, and Wuyts, 2009). 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 (more details in the next section). It

<sup>1</sup>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.



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<sup>2</sup> 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 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). This was done to make the sorter extendable and open for modification, so that anyone would be able to implement and plug in a sorting strategy they would like to have.

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).

### 3.5 Summary

- Pharo is an object-oriented dynamically typed language inspired by Smalltalk, and the Pharo IDE is an interactive programming environment intended for developing in the Pharo language.

---

<sup>2</sup>Developer in the Pharo IDE, who is a user of code completion.

- 
- Code completion in the Pharo IDE is used in the code editors, the list of which includes the Playground (a scripting tool), the System Browser (a tool for browsing, writing and editing class functionality), the Debugger (a tool for editing code while in debug mode), and so on.
  - The current code completion implementation in the Pharo IDE is based on analysing the AST of source code; it is called after the developer types at least two alphabetic characters. The completion candidates are then suggested based on the most relevant semantic context.
  - Other than the completion engine, which is responsible for the process of *completing* code, another important part of the code completion tool is the sorter plugin that supports implementing various sorting strategies. It is responsible for the order in which the results are displayed to the user.

## Chapter 4

# N-gram Background

In this chapter, we go over the theoretical background for the n-gram models and the challenges in their implementation.

### 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 (Jurafsky and Martin, 2009). 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:

$$P(w|h) \tag{4.1}$$

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:

$$P(\text{pizza}|\text{he ate the whole}) \tag{4.2}$$

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:

$$P(\text{pizza}|\text{he ate the whole}) = \frac{C(\text{he ate the whole pizza})}{C(\text{he ate the whole})} \tag{4.3}$$

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

$$P(w_1w_2\dots w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1w_2)\dots P(w_n|w_1w_2w_3\dots w_{n-1}) \tag{4.4}$$

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:

$$P(\text{pizza}|\text{he ate the whole}) \sim P(\text{pizza}|\text{whole}) \quad (4.5)$$

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 4.4) can be approximated the following way:

$$P(w_1^n) \approx \prod_{x=1}^n P(w_x|w_{x-1}) \quad (4.6)$$

## 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  $\langle \text{UNK} \rangle$ , and then replace every unknown word in the test data with  $\langle \text{UNK} \rangle$  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:

$$P(w) = \frac{c + 1}{N + V} \quad (4.7)$$

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

### 4.3 Summary

- Language models assign probabilities to sentences and sequences of words, and can be used to predict the next word in a sequence.
- N-gram language models calculate conditional probabilities of words in a corpus given a limited history of  $n-1$  previous words.
- The joint probability of a sequence is a product of each words' probability.
- N-gram models belong to the family of Markov models – they make a restrictive assumption that every word in a sequence depends only on  $n-1$  previous words, and any words beyond that window have little effect on the probability and can be ignored.
- Occasionally, there are words that are not in the vocabulary, which means their probability of occurrence will be zero, and will bring the joint probability of a sequence to zero also, ultimately hurting the performance of the model that can end up discarding perfectly valid sequences.
- To solve this problem, the process of smoothing is used. That way probability is redistributed among all the words, with a little bit taken off more frequent words and given to the unknown ones.

## Chapter 5

# Proposed Solution

This chapter contains a detailed description of the final solution: data preparation, approaches taken to implement the n-gram sorters, and various engineering details.

### 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).

#### 5.1.1 Unigram Sorting

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. In Listings 5.1, 5.2 and 5.3 you can see the frequencies data being passed to the sorter, and then the completions being sorted based on that information:

LISTING 5.1: The unigram sorter implementation: passing the frequencies to the sorter

```
FrequencyCompletionSorter >> initialize
  sorter := FrequencySorter new.
  sorter frequencies: Unigram uniqueInstance frequencies
```

LISTING 5.2: The sorter implementation: sorting the candidates according to their frequencies

```
FrequencySorter >> sort: anOrderedCollection
  ↑ anOrderedCollection sort: [ :a :b |
    (frequencies at: a contents ifAbsent: 0) >
    (frequencies at: b contents ifAbsent: 0) ]
```

LISTING 5.3: The unigram sorter implementation: returning the results sorted in the code above

```
FrequencyCompletionSorter >> sortCompletionList: anOrderedCollection
  ↑ sorter sort: anOrderedCollection
```

## 5.1.2 Bigram Sorting

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). Below you can see how the sorting method looks in the end:

LISTING 5.4: The bigram sorter implementation: receives the list of completions, gets the word before from context, trains the model, and sorts the completions based on sequence probabilities

```
BigramCompletionSorter >> sortCompletionList: anOrderedCollection
  | probabilities |
  probabilities := bigram probabilitiesFromModel: model
  history: self getWordBefore
  andCollection: anOrderedCollection.
  sorter probabilities: probabilities.
  ↑ sorter sort: anOrderedCollection
```

## 5.2 Implementation Details

### 5.2.1 Data Preparation

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<sup>1</sup> by Zaitsev, Ducasse, and Anquetil, 2020 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 5.3.

<sup>1</sup><https://gitlab.inria.fr/rmod-public/2019-sourcecodedata>

## 5.2.2 N-gram Library Extension

The bigram model trained on source code data was implemented with the help of the NgramModel library<sup>2</sup> 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 (see Listing 5.5). 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, described in detail in section 5.3).

LISTING 5.5: Here is how the bigram model for sorting is created and trained

```
BigramTableCreator >> model
| model |
model := NgramModel order: 2.
model trainOn: self processedTokens.
model removeNgramsWithCountsLessThan: 50.
↑ model
```

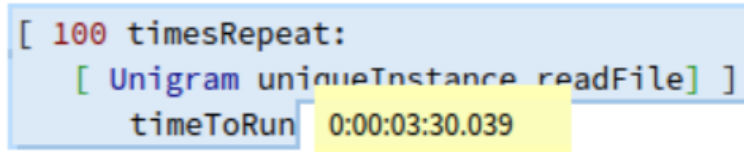
When I first started using the NgramModel library, the functionality to reduce the number of n-gram sequences based on frequencies, or the functionality to read and write the model to file was not present. In the process of using this library for creating the bigram sorter, I extended it with the methods needed, as it might be useful for others to have this in place, too. Thus, through 4 pull requests I:

- added the functionality to filter the table of n-grams (i.e. to cut off n-gram sequences with counts below a certain threshold)
- added writing to and reading n-gram models from file
- improved test coverage by writing tests for some additional examples

## 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 (see Figure 5.1: 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).



```
[ 100 timesRepeat:
 [ Unigram uniqueInstance readFile ] ]
timeToRun 0:00:03:30.039
```

FIGURE 5.1: It takes more than 3 minutes to read the results from file on repeat 100 times

<sup>2</sup><https://github.com/pharo-ai/NgramModel>



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 (see the improvement in Figure 5.2).

```
[ 100 timesRepeat:  
  [ Unigram uniqueInstance readFile] ]  
timeToRun 0:00:00:05.153
```

FIGURE 5.2: After the cut off, it takes only 5 seconds to run it 100 times

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 "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 Using the N-gram Sorters

It is worth noting that both the n-gram sorters can actually be used in the Pharo IDE by loading the CompletionSorting library<sup>3</sup>, 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 sorting option in the Settings (Figure 5.3, below).

The unigram sorter is quite fast and accurate (we go over the quality of the performance in Chapter 6). However, the bigram sorter is too slow for comfortable usage, and re-engineering the implementation to be faster is left for future work.

<sup>3</sup><https://github.com/myroslavarm/CompletionSorting>

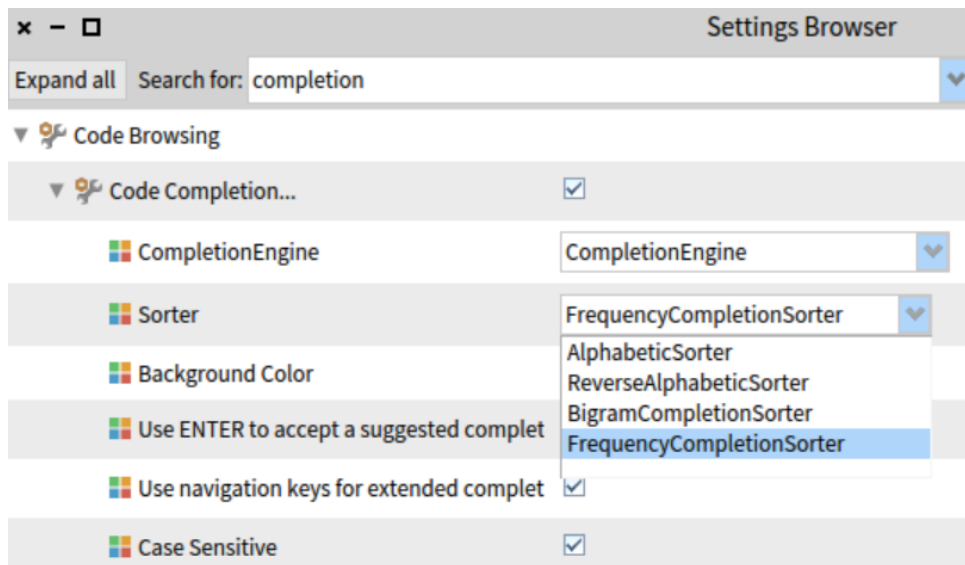


FIGURE 5.3: The Settings window in the Pharo IDE

## 5.5 Summary

- The data used to train the models came from 50 projects implemented in Pharo; in the dataset, the source code was already split into tokens and token types. In the data, delimiters and non-alphabetic tokens, as well as comments, were all considered to be separate tokens.
- For unigram sorting, individual token frequencies were extracted, and the sorting of completions was based on those frequencies.
- For bigram sorting, completion candidates were added to the previous token to form sequences of bigrams, whose probability was later calculated and recorded to file. The sorting was based on the sequence probability, but it was only applied to individually displayed completion candidates.
- Different approaches were taken to reduce the sizes of models and to speed up lookup and sorting time; among them: cleaning up tokens with occurrences below a certain threshold, and replacing "one-time" tokens (such as specific strings, numbers, et cetera) with general placeholders (such as <str>, <num>).
- In the process of working on the bigram sorting strategy, the NgramModel Pharo library that was used for this approach was additionally extended with new functionality, such as reading/writing to file and reducing the number of sequences with counts below a certain threshold.
- Finally, both sorting strategies can be used in the Pharo IDE. However, the bigram sorter remains too slow for comfortable day-to-day usage.

## Chapter 6

# Evaluation

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong."

---

— Richard Feynman

In this chapter, we go over different approaches and challenges for evaluating code completion, as well as analyse the results from the evaluation experiments for this project.

### 6.1 Evaluation Overview

Evaluation is an essential part of any research process. In the Richard Feynman quote above, the evaluation *is* the experiment he is referring to. To paraphrase and make it more relevant to software engineering, it does not matter what kind of tool you implemented and how well do you think it works, there need to be clear metrics the implementation is measured on and a clear definition of what is considered to be a successful evaluation.

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<sup>1</sup>. 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<sup>2</sup> 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.

---

<sup>1</sup>Hazzan et al., 2006 in their paper discuss the qualitative approach and its relationship to the quantitative one.

<sup>2</sup>Kaplan and Maxwell, 2005 describe an extensive qualitative evaluation example and talk about the key benefits and considerations of this approach.

## 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. The knowledge, interests, perceptions and skills of the researches undoubtedly play a role in the evaluation, and the subjective nature of the process is an undeniable part of the experiment and needs to be carefully addressed. In the end, the goal is to represent the process as closely as possible to how it really happens and report the results that are suitably meaningful, while exercising a healthy amount of unbiased professional detachment.

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

### 6.3.1 Experiment

The quantitative evaluation approach performed for this project is based on the paper by Robbes and Lanza, 2008, 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 taken from the Robbes and Lanza, 2008 paper). 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)

### 6.3.2 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 6.1 presents the accuracy (average success rate) for each of the classes and sorting strategies tested.

Class	# of Methods	Alphabetic	Unigram	Bigram
OrderedCollection	64	0.32	0.38	0.30
CompletionEvaluation	9	0.24	0.25	0.22
FrequencyCompletionSorter	2	0.48	0.58	0.47
NgramModel	21	0.32	0.36	0.29

TABLE 6.1: 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, surprisingly, the bigram sorter performs both worse than the unigram sorter and even slightly worse than the alphabetic one.

## 6.4 Qualitative Evaluation

### 6.4.1 Experiment

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.

### 6.4.2 Results

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.

It is worth noting that 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. However, as making the results as effective as possible is the goal of improving code completion in the first place, consistent placement of relevant results in the top 10 even after two symbols is a considerable advantage and does demonstrate which sorting strategy performs better.

Below you can see the 10 examples tested, the results, and their explanations.

#### 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 6.2: Qualitative evaluation: the results of completing a message send in the Playground

As we can see in the Table 6.2, 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	-	-
	ifTrue:	1
Unigram	ifTrue:ifFalse:	2
	ifFalse:	3
Bigram	ifTrue:	5

TABLE 6.3: Qualitative evaluation: completing a message send to an expression in parentheses

In the Table 6.3 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).

**Example 3:**

```
TokenProcessing >> returnProcessedData
| tokensDataFrame |
tokensDataFrame := self re
```

This was typed in the Editor, in the method *returnProcessedData* of class *TokenProcessing*. This is done to try out some more elaborate examples where the local context can play a role.

Here, as *rejectInvalidTokens*, *replaceWithPlaceholders* and *readFile* are also methods on the instance side of the *TokenProcessing* class, we as users would expect to get their names as suggestions after typing *self*.

Sorter Type	Completion Suggestion	Position
Alphabetic	readFile	3
Unigram	-	-
	rejectInvalidTokens	7
Bigram	replaceWithPlaceholders	8
	readFile	9

TABLE 6.4: Qualitative evaluation: completing a message send after *self*, with awareness of local context

Table 6.4: it seems that the unigram approach fell short, whereas the bigram approach, trained on prioritising local method names after *self*, has performed quite well. In the alphabetic approach, the other two method suggestions seem to be buried below the first 10, whereas *readFile* gets a better position.

**Example 4:**

```
TokenProcessing >> replaceWithPlaceholders
  | data tokens tokenTypes arr |
  data := self returnProcessedData.
  to
```

By typing *to* we are expecting to get *tokens* and *tokenTypes*, as those are temporary variables that we declared and now might want to initialise with some values. Here are the results:

Sorter Type	Completion Suggestion	Position
Alphabetic	tokenTypes	1
	tokens	2
Unigram	tokens	1
	tokenTypes	2
Bigram	tokens	1
	tokenTypes	2

TABLE 6.5: Qualitative evaluation: completing temporary variable names, with awareness of local context

Here (Table 6.5) the order in the unigram and bigram cases is slightly better, as having a shorter word suggested first is preferable, but in general all three approaches worked quite well for this example.

#### Example 5:

```
UnigramTableCreator ne
```

Typing this in the Playground, we expect *new* and maybe even *new:*, depending on the context.

Sorter Type	Completion Suggestion	Position
Alphabetic	new	2
	new:	3
Unigram	new	1
	new:	2
Bigram	new:	4
	new	8

TABLE 6.6: Qualitative evaluation: completing message send to a global variable (in this case, trying to create a new instance of a class)

In the Table 6.6 we can see that the best positions are for the unigram sorter, but in general all the sorters suggested the desired options in the top 10.

#### Example 6:

```
in
```

In the Pharo IDE it is possible to get completion suggestions even when creating a new method, by typing the desired method name in the System Browser code editor on the instance side. If it is a common method that users often create in their classes, having a completion suggestion is useful. As this is the



case with the method *initialize*, we would like to get the name suggested to us after typing in *in*, as illustrated above.

Sorter Type	Completion Suggestion	Position
Alphabetic	-	-
Unigram	initialize	2
Bigram	-	-

TABLE 6.7: Qualitative evaluation: completing method name during its creation

Table 6.7: the unigram sorter is the only that was able to suggest *initialize* from only two characters typed in, and at the 2nd position!

#### Example 7:

```
ExperimentalNgramTest >> testMostLikelyContinuations
| expected actual word |
word := 'lorem'.
actual := model mostLikelyContinuations: word asNgram top: 3.
expected := #('ipsum' 'dolor' 'lorem').
self as
```

In test methods having an *assert:equals:* call is quite common, and as there are a lot of tests being written, we would be interested in having a completion suggestion that is as accurate as possible after *as*.

Sorter Type	Completion Suggestion	Position
Alphabetic	-	-
Unigram	assert:	1
	assert:equals:	2
Bigram	assert:equals:	1

TABLE 6.8: Qualitative evaluation: completing a message send to self, when self is a test class

Table 6.8: both the unigram and the bigram sorters showed good results, but not the alphabetic one.

#### Example 8:

```
CompletionEvaluation >> averageSuccessRatio
| sumS sumA |
sumS := 0.
sumA := 0.
countSuccess do: [ :each | su ]
```

Here we are interested to see how the sorting performs when we are inside a block, so the cursor is after *su* before the bracket *]*. Since inside the block we want to do an assignment for the temporary variables, we would be expecting *sumA* and *sumS* as suggestions.

Sorter Type	Completion Suggestion	Position
Alphabetic	sumA	1
	sumS	2
	super	3
Unigram	super	1
	sumA	2
	sumS	3
Bigram	super	1
	sumA	2
	sumS	3

TABLE 6.9: Qualitative evaluation: completing inside a block

The results in the Table 6.9 (above) are quite straightforward: all the sorters seem to perform well, although since one is less likely to want to call *super* directly inside a block, the alphabetic one seems to be most relevant (purely coincidentally, as *p* follows *m*).

**Example 9:**

```
RBScanner sc
```

Typing the above sequence in the Playground, we want to get method names on the class side of *RBScanner* that begin with *sc*. There are two: *scanTokens:* and *scanTokenObjects:*, so they are the ones we should be suggested.

Sorter Type	Completion Suggestion	Position
Alphabetic	scanTokenObjects:	1
	scanTokens:	2
Unigram	scanTokenObjects:	1
	scanTokens:	2
Bigram	scanTokens:	1
	scanTokenObjects:	2

TABLE 6.10: Qualitative evaluation: completing a message send in the Playground

As can be seen in the Table 6.10, all sorters showed essentially the same result. However, *scanTokens:* is more likely to be used next to *RBScanner* and it is a shorter token, so the bigram result is slightly superior.

**Example 10:**

```
UnigramTableCreator >> writeToFile
| data stream |
data := self frequencies.
stream := TokenProcessing frequenciesFile writeStream.
(NeoCSVWriter on: stream)
  nextPut: #(key value);
ne
```

Now, for an even more creative example, in Pharo there can be multiple messages sent to a receiver, as long as they are all separated by a semicolon ;.

So having a cursor after *ne*, we know we are sending a message to *NeoCSVWriter* on: *stream* and want to get suitable suggestions, which are *nextPut:* and *nextPutAll:*.

In the Table 6.11 below we see that the unigram sorter got the best result.

Sorter Type	Completion Suggestion	Position
Alphabetic	-	-
Unigram	nextPutAll:	2
	nextPut:	3
Bigram	-	-

TABLE 6.11: Qualitative evaluation: completing one of multiple message sends, with awareness of local context

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, also more often than not prioritising the desired completions, which disagrees with the quantitative evaluation. So, while based on raw data the bigram sorter is the worst of the three *on average*, from the manual evaluation it seems that it gives perfectly adequate results for examples that would be commonly encountered in Pharo. 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.

## 6.5 Summary

- Quantitative evaluation involves benchmarking the model and calculating performance scores. In contrast, the qualitative approach involves conducting the evaluation via letting people test the tool, observing the human-computer interaction, recording feedback interviews, and so on.
- There are certain challenges in evaluating code completion, which can be divided into three main categories: resources limitation, finding appropriate metrics, and reproducibility.
- Based on the evaluation tool that I created and used to evaluate average success rate (i.e. out of all the attempts to complete the history, how many of them had the expected match in the first 10 results) for each sorting strategy, the unigram sorter performed better than the alphabetic sorter but, surprisingly, the bigram one performed worse.
- For the qualitative evaluation, 10 completion cases were preselected for their common occurrence when coding in the Pharo IDE, with no bias towards any sorting strategy. The examples were tested in the real environment, and the positions of the desired completion suggestions were recorded for each of them. Using this evaluation, the unigram approach was once again the best, but on the cases tested, the bigram approach did not seem to perform so badly.
- Based on this, as part of future work, I plan to investigate the potential issues with the bigram implementation and try to enhance its performance.

## Chapter 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 initially stated in Section 1.3:

**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 Robbes and Lanza, 2008, 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

### 7.2.1 Enhancing the Bigram Sorter

As can be seen in Chapter 6, 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.

Hence, for future work, it would be useful to see what exactly went wrong with the implementation. It could be that there is a mismatch between how source code is parsed and tokenised in the training and test data. Or it could be a matter of managing the delimiters incorrectly, as punctuation in source code has a significant influence on the contextual meaning of any part of code. It could also be an issue with the NgramModel library, which was only used for training the bigram model, whereas the unigram model was implemented using a different approach.

In any case, this would be a good idea for the continuation of this project: analyse the issues of the existing bigram implementation and try to fix them and improve the performance of the bigram-based sorter.

### 7.2.2 Conducting a More Extensive Evaluation

For the quantitative evaluation, it would be useful to evaluate the results more precisely. Meaning that instead of an average accuracy, perhaps a more sophisticated formula could be used. For instance, the idea by Robbes and Lanza, 2008 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.

# Bibliography

- Allamanis, Miltiadis et al. (2018). “A survey of machine learning for big code and naturalness”. In: *ACM Computing Surveys (CSUR)* 51.4, p. 81.
- Asaduzzaman, Muhammad et al. (2014). “Context-sensitive code completion tool for better api usability”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, pp. 621–624.
- Bruch, Marcel, 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*, pp. 213–222.
- Franks, Christine et al. (2015). “Cacheca: A cache language model based code suggestion tool”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE, pp. 705–708.
- Hazzan, Orit et al. (2006). “Qualitative research in computer science education”. In: *Acm Sigcse Bulletin* 38.1, pp. 408–412.
- Hellendoorn, Vincent J 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*, pp. 763–773.
- Hellendoorn, Vincent J et al. (2019). “When code completion fails: A case study on real-world completions”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, pp. 960–970.
- Hindle, Abram et al. (2012). “On the naturalness of software”. In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, pp. 837–847.
- Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc. ISBN: 0131873210.
- Kaplan, Bonnie and Joseph A Maxwell (2005). “Qualitative research methods for evaluating computer information systems”. In: *Evaluating the organizational impact of healthcare information systems*. Springer, pp. 30–55.
- Li, Jian et al. (2017). “Code completion with neural attention and pointer networks”. In: *arXiv preprint arXiv:1711.09573*.
- Nguyen, Tung Thanh et al. (2013). “A statistical semantic language model for source code”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 532–542.
- Pluquet, Frédéric, 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*. Orlando, Florida, USA: ACM, pp. 69–78. ISBN: 978-1-60558-769-1. DOI: [10.1145/1640134.1640145](https://doi.org/10.1145/1640134.1640145).
- Proksch, Sebastian, Johannes Lerch, and Mira Mezini (2015). “Intelligent Code Completion with Bayesian Networks”. In: *Transactions on Software Engineering and Methodology (TOSEM)* 1.25. DOI: [10.1145/2744200](https://doi.org/10.1145/2744200).
- Raychev, Veselin, Martin Vechev, and Eran Yahav (2014). “Code completion with statistical language models”. In: *Acm Sigplan Notices*. Vol. 49. ACM, pp. 419–428.

- Robbes, Romain and Michele Lanza (2008). "How Program History Can Improve Code Completion". In: *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pp. 317–326.
- Svyatkovskiy, Alexey et al. (July 2019). "Pythia: AI-assisted Code Completion System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2727–2735. ISBN: 978-1-4503-6201-6. DOI: [10.1145/3292500.3330699](https://doi.org/10.1145/3292500.3330699).
- Tu, Zhaopeng, 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*, pp. 269–280.
- Zaitsev, Oleksandr, Stéphane Ducasse, and Nicolas Anquetil (2020). *Characterizing Pharo Code: A Technical Report*. Technical Report. Inria Lille Nord Europe - Laboratoire CRISTAL - Université de Lille ; Arolla. URL: <https://hal.inria.fr/hal-02440055>.