A Novel Code Completion Strategy

Hayatou Oumarou
University of Maroua, Maroua, Cameroun
LaRI Team, Maroua, Cameroun

Ousmanou Dahirou
University of Maroua, Maroua, Cameroun

***Abstract*—Programmers rely on a multitude of techniques to speed up the development process. Among these techniques is code completion, a productivity improvement technique widely used by developers to explore APIs and automatically complete a word being typed by providing a progressively refined list of candidate words (or recommendations). Still called auto-completion, it reduces incorrect calls to APIs. Several techniques have been developed to obtain the list of candidates. Some methods use the history of the code, others neural networks or artificial intelligence; some exploit the program's structure through AST. Often the recommendation list is long, and finding suitable candidates comes at a cost. In this work, we propose a strategy that improves the accuracy of recommendation list offered by code completion. We present a sorting approach based on the popularity and importance of the elements (suggestions) of the list by analyzing the usage data of classes, methods, and variables of projects in the same development environment. We implemented our sorting strategy in Pharo (IDE and language), an immersive modern programming environment to show its applicability. The empirical evaluation results of this strategy show that our approach improves the quality of the suggestions.**

***Keywords*—Integrated development environment; code completion; API; code completion tool; pharo**

I. Introduction

Integrated Development Environments (IDEs) have become a critical paradigm for software engineers to speed up the coding process and reduce typos and other common errors. An IDE brings together tools for developing software such as mobile applications, computer or game console applications, web applications, etc. There are more IDEs than programming languages. However, most IDEs are specific to a given language [1]. A modern IDE has various tools distributed together, which are among others: the text editor, the graphical interface, the debugger, the compiler, testing and versioning tools, ... helping the programmer to write code efficiently and accurately by providing it with a set of valuable services such as automatic indentation of code blocks, highlighting of language keywords by color or bold characters, code completion, etc. Code completion is the mechanism allowing from part of a word entered by the user to offer him a progressively refined list of candidates (complements) which could suit the remaining string of characters of the word. This functionality can be found in several applications : text editors (for entering source code, for word processing, etc.), web browsers, as well as specific intuitive input systems installed on smartphones. This work will focus on code completion in IDE editors, highlight its shortcomings, and propose an improvement.

A. Context

Murphy [3] published an empirical study on how 41 Java developers used the Eclipse IDE. One of their findings was that every developer in the study used the code completion feature. Among the top commands executed by the 41 developers, code completion came in sixth with 6.7% of the number of commands executed, sharing the top spots with basic editing commands such as copy, paste, save and delete. Not surprisingly, this has been little discussed: Code completion has become second praxis to implementation activity. Nowadays, every major IDE offers a language-specific code completion system; according to [4] any text editor must provide at least word completion to be considered usable for programming. In the same vein, we did an online survey in June 2020 with local developers. The survey focused on quality practices and measures in software development companies [5].

B. Motivations

Now-a-days, software development has become highly complex and very difficult to master due to the increase in the scale of the projects, the increasingly short development time, the requirements and the quality of the software product. To successfully manage the development of software, it is necessary to consider many parameters, including material resource constraints, the programming languages used, and the human factor. The latter greatly influences the progress of software development. The software production cost includes the hardware cost, the training cost, and the effort required for the development and maintenance. However, the most significant cost is that of the construction effort (software development) because it represents more than 80% of the software production time [6]. Cost overruns and delayed product delivery deadlines are often encountered during software development. To provide an element of the solution to these problems, engineers should strive to act to ease the development process by reducing costs (effort) and development time. Thus, we could increase the productivity of programmers by automating part of the activities (reducing keyboard input by code completion) and by simplifying operations (rapid debugging, less or easily consults the documentation) to achieve the qualitative objectives (quality, cost and time). The code source is the essential element of the software. It is done during implementation and is reviewed during maintenance, representing more than 80% of the software cost. The code completion mechanism takes center stage. Jin [1] highlighted the hidden cost of auto-completion, which mainly impacts developers when code completion techniques produce long recommendations. They show how the length of the recommendations list affects other factors that can lead to inefficiencies in the process. The idea of improving and adapting the completion mechanism is relevant

to provide support to developers by providing considerable time savings and reducing the number of typing errors during development. So software developers will write code more effectively and efficiently.

C. Description of the Problem

Automatic code completion is considered the most used feature in integrated development environments [14]. The recommendations (suggestions) are presented to the programmer during the completion process in a pop-up window in a specific order. A study on the length of self-completion suggestion lists [1] found that around 17% of the lists were 250 items long. In the same study, they showed that the median position of the selected item is beyond the 100th spot. So given the multitude of suggestion possibilities offered by code completion and the way these suggestions are sorted, finding a candidate's place in the list can be tedious or slower than typing in the full name of the element to be completed. Generally, it takes too long to locate the word in the suggestions list. In that case, code completion loses all its importance because, instead of being a tool that increases the coder's productivity by reducing the entry time, it slows down and slow motion. This highlights weaknesses in the suggestion sorting strategies implemented in most code completion systems in development environments. These strategies do not rank the results relevance according to the programmer's context. This work will propose a sorting strategy that improves the precision of the list of suggestions.

D. Structure of the Document

Our main contributions are summarized as follows:

- We propose a new prioritization method.
- We propose a discriminator model on top of the IDE code completion engine that uses contextual scope information for precise code completion.
- We do an extensive experiments showing performance in terms of precision.

The rest of the paper is organized as follows: Section II presents the details of the RBSS strategy. Section III reports the experimental results. Section IV investigates related work. Section V discuss threats to validity of our approach. Section VI concludes the document.

II. The Strategy: Relevance Based-Sorting Strategy

In this section, we present the candidate list refinement strategy called Relevance Based- Sorting Strategy (RBSS). Fig. 1 show an overview of the approach. The idea is to measure the relevance score of each of the list elements (method name, variable name, class name, etc.) according to the number and weight of the links that it receives to refine its position in the suggestions list depending on the context. We need the static and structural information from the source code to do this. Thanks to of the Abstract Syntax Tree (AST) analysis, which returns the syntactic information of the element to be completed, we are able to sort the completion options (names of methods, variables, classes, definitions, etc.) according to their popularity ratings which are the essential

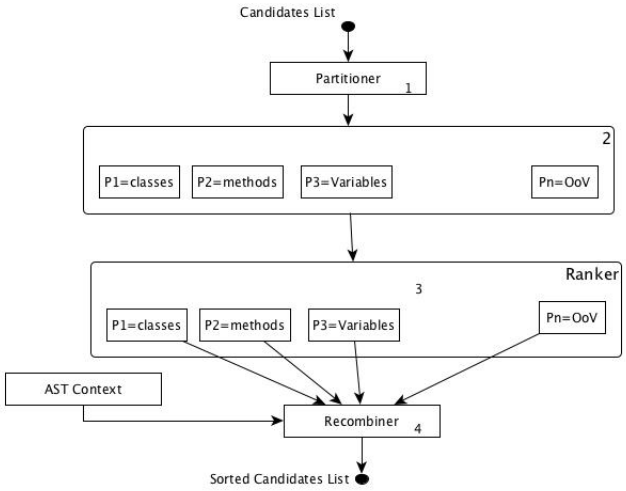

Fig. 1. Overview of RBSS Approach

criteria in referencement. To do this, we contextualize the idea of Google's web page indexing algorithm (PageRank). PageRank is an referencement technique used by the Google search engine to index web pages and provide results that are relevant. The latter, an index used by Google to know the popularity of its index, is noted between 0 and 10. A page is considered very popular if it has a maximum rating or index.

1) Partitioning of candidates The first step is to partition the candidates according to the types of elements with the RBSS Partition algorithm as in Fig. 2.

```
PARTITION
INPUT : candidates List (L)
OUTPUT : List of candidates lists by type (L')
BEGING
MethodsList, VariablesList, ClassesList,
        OutOfVocabList : List of candidates
   For each elt in L do
   case of type elt
        Method : Add elt in MethodsList
        Variable : Add elt in VariablesList
        Class : Add elt in ClassesList
        Otherwise : Add elt in OutOfVocabList
   end case
        End for
Return L'= [MethodsList, VariablesList,
        ClassesList, OutOfVocabList]
END.
```

Fig. 2. Algorithm RBSSPartitionner: Partitions a List According to Elements Type

In this step, each item in the list is classified according to:

- Methods For each method in suggestions list, it will be a question of looking at the number of their Senders and their Implementors. The

Senders represent all methods that may use or invoke a given method. The Implementors also work the same way. Instead of returning a list of Senders of a message (or method-envoyeuses), they resolve all classes that implement a method with the same selector. RBSS considers a method popular when many links point to it. Here, a link can represent either a call of the method in question by another method or an implementation of the method in the different classes of the system.

- classes Concerning the case of a class, we will be interested in its references as an attribute for evaluating its popularity. These references are : AllRefInside and AllRefOutside. AllRefInside represents all the references of the other classes of the system which reuse (inheritance, re-implementation,...) the properties of a given class. AllRefOutside does the opposite of AllRefOutside by displaying all the elements of the classes whose attributes a given class uses. Our strategy considers that a class is popular if it has more AllRefInside than AllRefOutside if the size of the AllRefInside list is greater than the size of AllRefOutside.
- variables Regarding the variables, we will consider either the number of methods that use it (outgoingInvocation), or the number of methods that store data in this variable (incommingInvocation). In both cases, the RBSS sorting strategy will consider the size of the list of methods as a criterion for evaluating the popularity of the variable.

2) Score attribution by category RBSS then rates the popularity of each item in the suggestion list based on the number of times other objects in the system use it. To assess the popularity score of items in a suggestion list partition, we assign a weight to each link that a given item can have. Thus, the total number of links of an element represents its popularity score, which is one factor that conditions the element's positioning (method, Class, Variable) in the suggestions list. for this, we use the RBSSSorter algorithm Fig. 3. This algorithm uses algorithms from Fig. 4 and Fig. 5.
The position of an element in the completion list considers the context of the word to be completed. So, we will contextualize the sorting of the list according to the type and the current situation. For example, in the case of a list having methods, variables, classes etc. as a completion proposal, the RBSS strategy after having sorted this list will first consider the context before displaying the result. If we are in a method context, RBSS will place the most popular methods at the top of the list and then complete the list with other popular elements (variable, class, ...) from this list and thus vice versa.
3) Recombination of the elements (see Fig. 6) of the partitions into a single list. For this we rely on the following RBSS Recombiner algorithm:

```
SORT
INPUT: L, the list of Candidates
RETURN: L', the sorted list of candidates
LOCAL: chg, the list of couple (l,c) where l in L and c
    the computed score
For each l in L
    Chg add (l, score(l))
End for
For i=1 to threshold do
    For each (l,c) in chg
    (l,c) ← (l, (c + SUM Score(l') l' In Neighbor(l) ) /
                    1+ (c + SUM Score(l') l' In Neighbor(l) ))
    End For
End for
Sort chg
RETURN first chg
 RETURN seq
END.
```

Fig. 3. Algorithm RBSSSorter: Sort the List of Candidate According to their Score

```
NEIGHBOURG
INPUT: l, a Candidate
RETURN: N, a list of neighbor of l
Case type of l :
    Method : N ← senders(l)
    Class : N ← AllRefInside (l)
    Variable : N ← OutGoingInvocation (l)
End Case
RETURN N
END.
```

Fig. 4. Algorithm Neighbourg: Find Neighbor of a Candidate

III. Experiments and Discussions

In a recent study [2] Hellendoorn and others present a case study on 15,000 code completions that were applied by 66 real developers. They find that many aspects of real-world completions are not represented in synthetic benchmarks and tested completion tools were far less accurate on real-world data. Worse, on the few completions that consumed most of the developers' time, prediction accuracy was less than 20% – an effect that is invisible in synthetic benchmarks. For these reasons we choose to test our strategy on real-world system.

```
SCORE
INPUT: l, a Candidate
RETURN: c, the score for l
    c ← #Neighbourg(l) / 1+ #Neighbourg(l)
RETURN c
END.
```

Fig. 5. Algorithm Score: Give a Score to a Candidate

```
RECOMBINER
INPUT : List of candidates list sorted by type (L')
OUTPUT : List of candidates(L)
BEGIN
    L':= empty list
    For i = 1 to lenght(L) do :
    if type-elt-context = type-elt (L[i]) then
    Add elements of L at the beginning of L'
    else
    Add elements of L at the end of L'
    EndIf
    EndFor
RETURN (L')
    END
```

Fig. 6. Algorithm RBSSRecombiner: Recombine Partioned Lists.

A. Setup

For the evaluation the data used is as follows:

- We tested a large body of 1000 tokens in the pharo 9.0 source code to clearly show the impact. Pharo is a pure object programming language in which everything is object. the criteria that guided this choice are : the length of the object's source code, the type of the object (instance side, class side, ...), for the evaluation. Finally, we compare the score of our model with some basic strategies. The main objective of this evaluation is to empirically answer the research question about RBSS:
- How accurate is the RBSS method? We use this dataset as the basis for our assessments. We use the context and scope of each source code element.

Table 1 shows the data statistics, where LOCs are the line of codes, Files are the number of files, and Total Projects are the number of included projects.

TABLE I. DATA STATISTICS

Packages	Object Types	LOCs	Tokens
Epicea, Collections-Strings, Colors	Instance side	1119	7514
Epicea, Collections-Strings, Colors	Class side	370	2860
Total		1489	10374

We have tried different scope granularities such as class scope, method scope, and block scope. To the test pattern, we used 10,374 Entries names range in length from 3 to 12, with an average of 5. Accuracy assessment measures the difference between the expected result and the result obtained. We are looking for the right word to be placed in top positions at best. Otherwise, it occupies the highest position in the list. For this purpose, we will look at the top2, top 3, top4 and top5. We stop at this level because [1] have shown that beyond position 5 the programmer continues to type. By the way the words are not usually long in Pharo. Experimental result We evaluate the accuracy of our completion models according to standard metrics. We mainly consider top-K accuracies implying that the correct completion was often near the top of the suggestion list. Meaning that, we evaluated the RBSS strategy according to the position of the expected word in the list of suggestions. We did not dwell on the speed of execution, which will be the subject of another study. For this, we intend to exploit optimization techniques such as indexing and dynamic programming. Table 2 shows the results obtained within the Pharo environments.

Through empirical evaluation of RBSS in both environments, we were capable to show its ability to improve the accuracy of the list of returned candidates. The tests results carried out according to the defined case show that with the RBSS strategy, If we take the first five candidates from the list, in 61,6% of cases, we have the right candidate in the list. From this point of view, the RBSS strategy is improved on auto-completion. In conclusion, we have shown that the sorting strategy based on the popularity of the elements improves the precision of the code completion system, that is, the positioning of the elements in the suggestion list. However, although RBSS performs better, we did not consider the execution speed aspect, which we intend to improve in future studies.

IV. THREATS TO VALIDITY

As any empirical evaluation, the results of our experiments are subject to threats to validity. We identified the following noteworthy threats:

- The studied system might not entirely represent a larger population of systems, either from another application domain or written in another programming language. This is always a complex threat to mitigate as there is little information on what property of a system is essential to ensure representativeness. Replication of the experiment for other systems must be realized. This said, we strongly believe our approach is independent of the programming language and the application domain.
- The way in which we setup our experimentation may introduce bias. We also believe Pharo and visual works are credible, real world, non-trivial, case study. It was medium to big system and it includes a significant number of completion tools and options. However, we firmly believe that our approach is language independent.
- We tried with different type of object (instance side, class side, traits). This was done to eliminate a possible problem with the obviously simple solution working for any kind of object.
- Internal threats to validity are related to the implementation of our approach. It is still possible that our approach implementation contains errors that can affect our results' exactitude. We manually studied a subset of the results to counter this threat and did not find any obvious errors. Bias concerning developer working habits might also occur in our selection of evaluation subjects. We selected various packages from the two studied systems to reduce this risk, all issued from different areas. Thus, we believe the objects represent a heterogeneous enough population of the source code element.

TABLE II. DATA COMPARISON

Token Type	Method			Class			Variable		
	Top3	Top4	Top5	Top3	Top4	Top5	Top3	Top4	Top5
RBSS	56,4%	59,7%	61,6%	50,4%	55,8%	63,4%	38,4%	39,9%	47,6%
Pharo	55,7%	59,6%	61,5%	50,7%	55,5%	63,4%	37,7%	39,8%	47,6%

V. SOME WORK ON CODE COMPLETION

Code completion. With the birth of IDEs, code completion research has received much attention in recent decades. In [8] authors present a largescale study of user interactions with autocompletion. They found that lowerranked auto-completion suggestions receive substantially lower engagement than those higherranked. They note that users are most likely to engage with auto-completion after typing about half of the query, and in particular at word boundaries. They also found that the likelihood of using auto-completion varies with the distance of query characters on the keyboard. In a first study, the traditional models which are based on the formal structure of the programs, that is to say on the syntactic information and the static properties of the code. These syntactic approaches have been the most explored [9], [4], [7]. In the paper [12] we can read: Software engineering and programming languages (SE / PL) should make the same transition as research on natural language processing, assisting traditional methods that only take into account the formal structure of the programs, that is to say the information on the statistical properties of the code, and also exploiting repetitive and predictable elements of the source code. Essentially, three completion techniques are recited, each using the information in the example database differently. It is about : A Frequency Based Code Completion System (FreqCCS) uses the frequency of method calls to decide their suitability and suggests the most frequently used method. An Association Rule Based Code Completion (ArCCS) which is a statistical learning technique for finding interesting associations between elements in data, ArCCS exploits the rules m $\rightarrow$ n which, if the method m is used, the method n is frequently called and suggested. The Best Matching Neighbors code completion (BMN) which is the modification of the K-Nearest-Neighbors machine learning algorithm, BMN adapts the KNN to suggest a variable v . The probabilistic models or statistical language models (Statistic Language Model). Recent work has started to examine linguistic models based on statistical learning [17], [18], [10] aiming to model the source code as statistical language learning models. These approaches offer an exciting new goal of the code completion problem that suggestions can capture the deeper meaning of terms' semantic and idiomatic meaning. These are, among others : N-gram models and Recurrent Neural Networks (RNN). The n-gram models which exploit probabilistic models and predict each token based on the probability of the preceding token. To deal with data scarcity, an N-gram data model estimates the probability of a sentence by modeling language as a Markov chain of order. The probability of the next word in the sentence (phrase) depends only on the previous words [11]. The Recurrent Neural Network outperforms the n-gram and predict each token (node to predict) sequentially. For example [16] proposes an approach that combines RNNs with networks of pointers to complete the code. In [13] authors explore the use of neural network techniques to automatically learn code completion from a large corpus of dynamically typed JavaScript code. Authors propose a neural network model and believe that neural network techniques can play a transformative role in helping software developers manage the growing complexity of software systems. Performance measurement of these approaches. The following metrics are the most used [15]: precision , recall and F-Measure. Whose formulas are:

$$precision(P) = \frac{Recommendations_{made \cap relevant}}{Recommendations_{made}}$$

$$recall(R) = \frac{Recommendations_{made \cap relevant}}{Recommendations_{relevant}}$$

$$F - measure = \frac{(2 * P * R)}{(P + R)}$$

The measurement F is called the harmonic mean of recall and precision. Usually, it is difficult to achieve optimal results simultaneously for recall and precision. For example, if all words are classified as irrelevant, the resulting recall score will be 100% where the accuracy score will be low. Therefore, measurement F is a compromise between recall and precision. The score range for measure F is 0 to 1 ; the higher score implies a better classification model.

VI. CONCLUSION

In this paper, we have proposed an approach for improving code completion. This approach if used increase the productivity of coders. In addition, this mechanism offers advantages in terms of reducing typing errors and time while increasing the efficiency and productivity of programmers. We have examined our approach with Pharo a dynamic object language. The experimental results have shown that our proposed method surpasses existing methods in terms of effectiveness and efficiency. It turns out that our sorting strategy significantly improves code completion. The accuracy of the list of suggestions (the right candidate is in the top 3 first candidates 59.6% of the time) compared to the Pharo 9.0 completion engine sorting.

Several improvements and perspectives are possible despite achieving an adequate precision sorting strategy. Indeed, the RBSS sorting strategy proposed in this work is limited and has shortcomings. The essential concerns its execution speed : RBSS is slower than the alphabetical sorting strategy, i.e. the strategy takes much longer to display list of suggestions. Thus, the main perspective considered concerns optimizing the RBSS sorting strategy to consume less resources. This optimization requires for example the creation and initialization of an index or of an array in which RBSS will read and that the array is updated automatically at a precise time or at a given

frequency to go fast. And this takes into account the size and the appropriate structure of this table. We also plan to compare the performance of our system with the latest systems proposed in the literature. Another perspective is to evaluate our approach on a large set or developers' community to collect their comments and quantify the pros and cons of our approach. We also wish to extend and test our approach on other languages and environments because currently, our results are valid than Pharo.

REFERENCES

[1] Jin, X., & Servant, F. (2018). The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-list Length on its Efficiency. Virginia Tech.

[2] V. J. Hellendoorn, S. Proksch, H. C. Gall and A. Bacchelli, "When Code Completion Fails: A Case Study on Real-World Completions," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 960-970, doi: 10.1109/ICSE.2019.00101.

[3] Murphy, M. K. (2006). How are java software developers using the eclipse IDE ? IEEE Software, 23(4), pp. 76–83.

[4] Robbes, R. a. (2008.). How program history can improve code completion. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.

[5] Hayatou, O., Kolyang, & Moulla, K. (2019). Sur l'utilisation des mesures de qualité lors des phases de développement dans l'industrie logicielle camerounaise. Maroua: Universite de Maroua.

[6] ZAKRANI, A. (janvier 2020). Estimation des coûts de développement de logiciels par un réseauneuronal RBF flou.

[7] D. Hou and D. M. Pletcher. (2011, Septembre). An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion. ICSM '11: proceedings of the 2011 27th IEEE International Conference on Software Maintenance, 3-6.

[8] Mitra, Bhaskar et al. "On user interactions with query auto-completion." Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval (2014).

[9] Han, S. D. (2009). Code completion from abbreviated input. Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on. IEEE.

[10] Bielik, P. V. (2016). PHOG: probabilistic model for code.

[11] Raychev, V. P. (2016). Probabilistic model for code with decision trees.

[12] Allamanis, M. e. (2018). A Survey of Machine Learning for Big Code and Naturalness.

[13] Chang Liu, X. W. (s.d.). NEURAL CODE COMPLETION. University of California, Berkeley.

[14] Amann, S. e. (2016). Une étude de l'utilisation des studios visuels dans la pratique. 23e Conférence internationale IEEE sur l'analyse, l'évolution et la réingénierie logicielles (SANER)., 1.

[15] Rahman, M. W. (2020). A Neural Network Based Intelligent Support Model for Program Code Completion. Scientific Programming.

[16] Jian Li, Y. W. (2016). Code Completion with Neural Attention and Pointer Networks.

[17] Hu, S. X. (2019). Scope-aware code completion with discriminative modeling. Journal of Information Processing, 27, 469-478. .

[18] Liu, F. L. (2020). A self-attentional neural architecture for code completion with multi-task learning. Proceedings of the 28th International Conference on Program Comprehension, 37-47.