Sequential coding patterns: How to use them effectively in code recommendation

Luiz Laerte Nunes da Silva Junior[1]

UFF, Instituto de Computação, Niterói, RJ, Brazil.

Troy Costa Kohwalter[1,*]

UFF, Instituto de Computação, Niterói, RJ, Brazil.

Alexandre Plastino[1]

UFF, Instituto de Computação, Niterói, RJ, Brazil.

Leonardo Gresta Paulino Murta[1]

UFF, Instituto de Computação, Niterói, RJ, Brazil.

Abstract

Context: Some programming constructs frequently appear together in different parts of the code, representing sequential coding patterns throughout the project. These sequential coding patterns can be mined from the project repository and, whenever the code a developer is writing coincides with the beginning of a sequential pattern, the remainder of this pattern can be suggested to the developer. This is equivalent to the usual Code Completion, which suggests syntactic structures based on the line being programmed. However, instead of providing syntactic suggestions for completing the current line, such feature suggests code snippets containing multiple lines.

Objective: This paper contributes with an in-depth study on how code pattern recommendation can be used effectively.

Method: We answer three research questions through a quantitative study

[*]Corresponding author.
Email addresses: `luiznunes@id.uff.br` (Luiz Laerte Nunes da Silva Junior), `troy@ic.uff.br` (Troy Costa Kohwalter, `plastino@ic.uff.br` (Alexandre Plastino), `leomurta@ic.uff.br` (Leonardo Gresta Paulino Murta)
[1]Universidade Federal Fluminense.

using a robust experimental infrastructure with a corpus of five open-source projects: (1) “In a code recommendation, how many frequent coding patterns should be presented?”, (2) “What is the impact of filtering sequential patterns by their confidence?”, and (3) “Does the effectiveness of the sequential coding patterns degrade over time?”.

Results: Our study shows that it is possible to achieve correctness above 80% when using suggestions with the highest confidence values and that a threshold confidence of 30% generally provides better outcomes. Furthermore, it shows that frequent code pattern completion effectiveness tends to degrade 50 commits after the patterns have been mined.

Conclusion: We could observe that: (1) the top five ranked suggestions are the ones that deliver the best results; (2) the code recommendations that deliver the best results are the ones with the highest confidence values; and (3) the code recommendation performance degrades as the source code evolves because patterns become outdated.

Keywords: sequential code patterns, data mining, recommendation system, version control system

1. Introduction

The improvement of code quality and productivity during software development is one of the main concerns of Software Engineering [1]. For instance, *code completion*, which is available in almost every Integrated Development Environment (IDE) [2], is an example of a tool aimed at increasing both code quality and productivity. It statically analyzes the source code and suggests automatic completion of variable and method names. This potentially increases code quality by avoiding typos and increases productivity by reducing typing effort.

Several works aim at extending the idea of *code completion* to recommend code from previously written code [3, 4, 5, 6, 7, 8, 9, 10, 11], considering that programs are written mostly using repetitive code [12]. These works are based on the idea that what was previously developed can be used to foresee, in some degree, what will be done in the future. For instance, a call to commit or rollback always appears after a call to begin a transaction on a database API, or a call to close a file always appears after calls to open the file and read or write to it. In addition to those examples based on well-known APIs, such approaches are able to detect more complex frequent patterns, which belong to the business domain.

The main challenge of code recommendation approaches is determining what, among a great amount of data obtained from the past, must be suggested. One of the most prominent ways to address this challenge is via sequential coding patterns mining [13] over the previously developed source code. Then, when new code is being developed, if it matches with the beginning of some of the previously obtained patterns, a code completion suggestion is provided with the remainder of the pattern that has not already been coded.

However, there is little knowledge about how these patterns behave and how they could be effectively used in code completion. Thus, this work aims at conducting a detailed study over the recommendation of sequential coding patterns for code completion by answering the following research questions (RQs):

1. RQ1: In a code recommendation, how many frequent coding patterns should be presented?
   Existing approaches suggest sequential patterns to a developer in a ranking. The number of patterns present in the ranking directly affects the correctness and automation provided by the approach. On the one hand, suggesting all patterns would certainly benefit automation but degrade correctness. On the other hand, suggesting just the top-ranked patterns would improve correctness but compromise automation. This RQ investigates the impact of presenting only a subset of suggestions to the developer.
2. RQ2: What is the impact of filtering sequential patterns by their confidence?

Just restricting the number of suggestions, as discussed in RQ1, may still include suggestions with a low probability of being useful. The confidence metric could be incorporated to prune the suggestions to be presented to the developer. Thus, the developer can analyze those with confidence values above a specific threshold, which represent suggestions with a high probability of being useful, omitting the ones with low probabilities of being relevant. This RQ investigates the impact of presenting a variable number of suggestions based on their confidence instead of a fixed number of suggestions.

3. RQ3: Does the effectiveness of the sequential coding patterns degrade over time?
   Our third study is targeted at examining the possibility of pattern expiration. As aforementioned, suggestions are obtained through the discovery of patterns in a repository. However, after mining the repository, developers commit changes to the code base, and previously obtained patterns may no longer reflect the current source code reality. Thus, this RQ focuses on observing the influence of time passing on sequential patterns.

We used four metrics for answering the RQs. The first metric is the **Automation**, which maps the adoption rate of a sequential coding pattern. The second is the **Correctness**, which maps the relevance of the pattern in a given situation. The third metric is **F-Measure**, the harmonic mean between the first two metrics. Finally, the fourth metric is the **Applicability**, which is the percentage of situations where sequential coding patterns can be applied. We also developed a robust experimentation infrastructure and used this infrastructure over five open-source project repositories to answer those three research questions.

We could observe that the top five ranked suggestions are the ones that deliver the best results, and it is possible to achieve correctness above 80% when using suggestions with the highest confidence values. Furthermore, our analysis shows that a threshold confidence of 30% generally provides better outcomes and that frequent code pattern completion effectiveness tends to degrade 50 commits after the patterns have been mined, since the source code evolves and the patterns become outdated.

This paper is organized in five sections. Section 2 describes the materials and methods employed in this work. Section 3 presents the experimental results for each research question. Section 4 presents the related work. Finally, Section 5 concludes this work, presenting the contributions and future work.

2. Materials and Methods

The first step needed to answer the research questions is providing of means to evaluate a significant amount of frequent coding patterns. This way, we built an infrastructure that can evaluate sequential coding patterns automatically. The evaluation demands two distinct stages to be performed: (1) We need to extract codification patterns, and then (2) we evaluate the frequent code pattern recommendations.

Version control repositories hold the evolution history, organized in commits and file revisions. Each commit creates a new revision for one or more files and, by diffing two consecutive revisions, we can obtain every added line of code. We can also check out an entire project revision, obtaining an old version of the project source code. This structure allows us to extract the sequential code patterns from a specific moment in the past and, after that, navigate through the subsequent repository commits. In this navigation, each file revision is individually evaluated, checking whether frequent code pattern suggestions would have been useful if they were available when the developers were coding.

Therefore, in our evaluation method, for a given version control repository $R = \{c_1, c_2, ..., c_n\}$, where n is the amount of commits available in the repository, the first $n/2$ commits are used for pattern extraction. Thus, the specific project revision produced by these $n/2$ first commits is checked out, and the pattern mining stage is applied in the source code, resulting in a set of frequent code patterns that may be stored in a pattern tree. Figure 1 gives a graphical representation of this tree with the following five patterns stored in it: $\langle(A)(B)\rangle$, $\langle(C)(D)(E)\rangle$, $\langle(C)(D)\rangle$, $\langle(C)(E)\rangle$, and $\langle(D)(E)\rangle$.

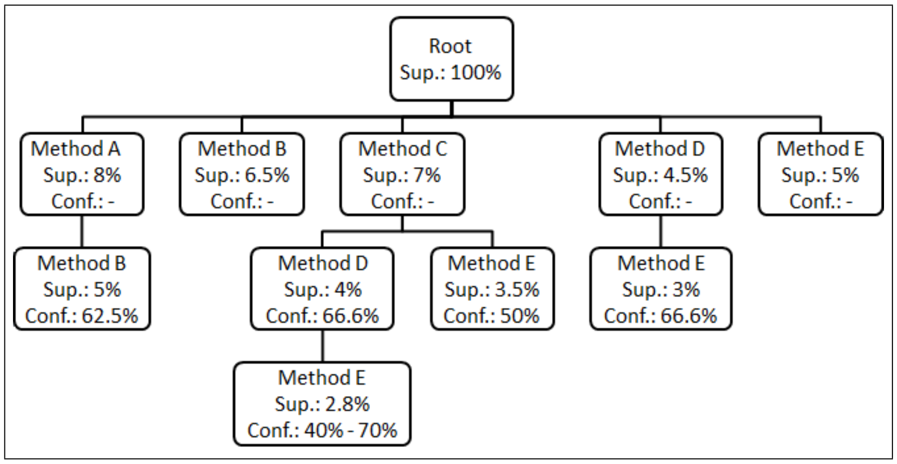


Figure 1: Sequential Pattern Tree with *support* and *confidence* annotation.

Considering every tree node as the end of a sequential pattern, each node contains the *support* of the pattern it represents. On the other hand, a pattern *confidence* cannot be seen as a single value, as it depends on the subsequence being queried. Given a pattern consisting of three methods, for example, the pattern could be suggested in two distinct situations: when the developer may have coded only the first method or may have already coded the first two methods of this pattern. In the former situation, the second and third methods of this pattern would be suggested, whilst in the latter, only the third. In these situations, what is more important is that both have different confidence values. Therefore, the length of the sequential pattern determines how many *confidence* values it should have.

In Figure 1, we are also able to see the aforementioned annotation of *support* and *confidence* in the tree. For instance, when observing the frequent sequence $\langle(C)(D)(E)\rangle$, it is possible to see the *confidences* 40% and 70% in the last node (deepest level), which represents method E. These *confidence* values represent the *confidence* of $\langle(C)(D)(E)\rangle$ related to $\langle(C)\rangle$ and the confidence of $\langle(C)(D)(E)\rangle$ related to $\langle(C)(D)\rangle$, respectively.

In the following, we exemplify the whole process based on the tree shown in Figure 1. Suppose a developer codes the method calls A, D, and F, in this order. Then, we would generate the following method combinations: $\langle(A)\rangle$, $\langle(D)\rangle$, $\langle(F)\rangle$, $\langle(A)(D)\rangle$, $\langle(A)(F)\rangle$, and $\langle(D)(F)\rangle$. The combination $\langle(A)(D)(F)\rangle$ would not even be generated because there is no pattern in the tree with more than three method calls. This way, the maximum combination size is automatically limited to two. These combinations would be queried in the sequential pattern tree, ordered according to their size.

First, the method call A would be queried, and method call B would be returned with *support* equal to 5% and *confidence* equal to 62.5%. After that, the method call D would be queried and the method call E would be returned with 3% of *support* and 66.6% of *confidence*. Next, after querying method call F, no pattern would be found and the following combinations would be discarded: $\langle(A)(F)\rangle$ and $\langle(D)(F)\rangle$. Then, combination $\langle(A)(D)\rangle$ would be queried and again no pattern would be returned. Last but not least, the identified patterns would be ranked according to their confidence values and suggested in the following order: $D \rightarrow E$, $A \rightarrow B$.

This strategy is based on the holdout method [14], where the data used into the evaluation is divided in two mutually exclusive subsets, the training subset, and the test subset. In our case, we have decided to split in half the project history. The first half is used to extract the patterns (training subset) and the second to evaluate the obtained patterns (test subset).

After that, the code in the last $n/2$ commits are used to assess the mined patterns. Each particular commit, c_i, is composed by file revisions, $c_i = \{f_1, f_2, ..., f_p\}$, where p is the number of files modified (added, deleted or updated) in the commit. Due to the need for static analysis over the source code, we restricted our assessment to Java file revisions, and also the ones that contains method bodies with method calls. Each Java file revision, f_j, is composed by method bodies, $f_j = \{m_1, m_2, ..., m_q\}$, where q is the number of method bodies present in the Java file. Finally, each method body, m_k, is composed by method calls, $m_k = \{mc_1, mc_2, ..., mc_r\}$, where r is the number of method calls present in the method body.

Therefore, the Java files must include at least one method body. This method body is eligible for evaluation if it contains at least two method calls, one for querying the set of frequent code patterns and the other for comparing with the query result. Furthermore, we decided not to evaluate modified method bodies, restricting the process to new method bodies. This choice was taken provided that we are trying to infer the code recommendation influence when a developer is coding an entirely new piece of code. Therefore, the method must be new and contain at least two method calls.

Our evaluation considers each method body as an opportunity to provide recommendations in the second stage. Thus, each method body is individually evaluated and, after that, the results are combined. For each method body, we check whether each method call mc_m, $2 < m \leq r$, could be foreseen through a frequent code pattern recommendation. To do so, we query for patterns using the ordered arrangements of $\{mc_1, ..., mc_{m-1}\}$ and the query result is compared against the remainder of the method calls, $\{mc_m, ..., mc_r\}$. The size of the arrangements was limited to five to run the experiments in a timely manner. Besides, the first method call, mc_1, is not evaluated, provided that there is no previous method call to be used in a query. The arithmetic mean of the results obtained from each method body produces the frequent code pattern recommendation performance assessment.

2.1. Project Corpus

The projects used to answer the research question were chosen according to some criteria. They should have Java as their primary programming language, as our analysis is language-dependent and Java is one of the most popular programming language[2]. Moreover, they should have at least 1,000 commits, aiming at avoiding toy projects and guaranteeing a significant amount of commits for evaluation.

[2]https://www.tiobe.com/tiobe-index/

Also, the projects should use Git as their version control system, as the data extraction phase of our analyses depends on the characteristics of the version control system. We have chosen Git, given its popularity. According to the Stack Overflow Developer Survey Results of 2018[3], which received answers from more than 100 thousand developers, Git is the most used VCS, summing up 87.2% of the answers.

We have selected five widely known open-source projects in the Java development community. All projects were active and under constant evolution (see table 1). The projects are:

- Commons IO[4]. A library of utilities aimed at helping developers coding input/output functionalities. It is developed by The Apache Software Foundation[5].
- Guava[6]. A group of core Java libraries involving collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. It is developed by Google and used in their Java-based projects.
- JUnit[7]. A framework to support writing automated tests in Java. It is the standard Java implementation of the xUnit architecture for unit test frameworks.
- RxJava[8]. A Java implementation of Reactive Extensions Library[9], an API developed to provide an easy way to deal with asynchronous programming through the Observer Design Pattern [15].
- Spring Security[10]. An authentication and authorization framework to

[3]https://insights.stackoverflow.com/survey/2018
[4]http://commons.apache.org/proper/commons-io/
[5]http://www.apache.org
[6]http://code.google.com/p/guava-libraries/
[7]http://junit.org/
[8]http://github.com/ReactiveX/RxJava
[9]http://reactivex.io/
[10]http://projects.spring.io/spring-security/

Table 1: Selected Projects.

Projects	**Number of Method Bodies**	**Number of Commits**	**Creation Date**
Commons IO	1,104	1,717	2002-01-25
Guava	6,055	3,024	2009-06-18
JUnit	1,626	1,951	2000-12-03
RxJava	1,494	3,744	2012-03-18
Spring Security	3,260	5,640	2004-03-16

secure Spring-based[11] web applications.

2.2. Independent and Dependent Variables

Data mining processes are characterized by discovering new and useful knowledge, in terms of rules and patterns, from large amounts of data. Sequential patterns consist of ordered sequences of events that appear with significant frequency in a dataset. A concept called *support* is used to evaluate the relevance of a sequential pattern. Given a dataset S, consisting of a set of sequences, the *support* of a sequence α, represented by $Sup(\alpha)$, is the number of sequences in S which are super sequences of α. Thus, support is an important metric, as it indicates if a sequence of method calls that repeatedly appears in the source code can be considered a pattern.

We also use another metric, called *confidence*. This metric is originated from the association rules field and can be adapted in the context of sequential patterns mining as follows. Considering α and β as two sequences, where α is a subsequence of β, the *confidence* of β in relation with α, $Conf\,(\alpha \rightarrow \beta)$, is the proportion of sequences in S that contain β among all sequences in S that contain α: $Conf\,(\alpha \rightarrow \beta) = Sup(\beta)/Sup(\alpha)$.

This concept can be exemplified as follows. Assuming a sequential pattern β consisting of $\langle$(“Star Wars”)(“The Empire Strikes Back”)(“Return of

[11]http://spring.io/

the Jedi")⟩, whose *support* is 28%, and another sequential pattern α consisting of ⟨("Star Wars") ("The Empire Strikes Back")⟩, whose *support* is 35%, then $Conf\,(\alpha \rightarrow \beta) = 80\%$. In this case, we can state that 'with 80% confidence, customers that rent "Star Wars" and "Empire Strikes Back", in this order, also rent "Return of the Jedi" afterwards.

In the context of code recommendation, given a sequential pattern of method calls $\langle(A)(B)(C)(D)\rangle$, whose *support* is 21% and another sequential pattern of method calls $\langle(A)(B)\rangle$, whose *support* is 28%, we could state that: "Developers that invoke methods A and B, in this order, also invoke, with 75% confidence, methods C and D." When the suggestions for method calls are provided to the developer, the confidence indicates which suggestions should be presented first. Thus, even if many patterns are returned from a query, the developer can analyze only the returned ones with the largest confidence values.

Finally, we only considered method bodies for which at least one suggestion was provided. These are classified as the *evaluated methods*, while the method bodies containing at least two method calls are the *valid methods*. This way, not all *valid methods* were actually evaluated, only the *evaluated methods* were considered in the results. We made this decision because in some situations where no suggestions were provided, the code recommendation was neither helping nor disturbing the developer with useless suggestions. Thus, it would be meaningless to penalize the Automation analysis with the accounting of these situations.

We defined four dependent variables that were used to answer our research questions, aiming at evaluating code recommendation performance in different scenarios: **Automation**, **Correctness**, **F-Measure**, and **Applicability**. The first two metrics respectively verify whether the recommendations could be foreseen and whether the suggestions are relevant. In addition, F-Measure represents a compromise between them, where high values of F-Measure are obtained only when both Automation and Correctness are also high. The F-Measure is defined in Formula 1. The fourth metric, Applicability, is the reason between the *evaluated methods* and the *valid methods*.

$$F - Measure = 2 \times \frac{AutomationPerc \times Correctness}{AutomationPerc + Correctness} \quad (1)$$

To exemplify the metrics evaluation process, consider the sample pattern tree in Figure 1. Suppose the evaluation of a single method body composed of five method calls: $\langle (A), (C), (D), (E), (F) \rangle$. The metric calculation is made as follows:

1. For each method call, a query is performed with all methods already coded: Considering the method calls being evaluated, the following requests are made, in this order:
   - Search_Patterns(root, $\langle (A) \rangle$);
   - Search_Patterns(root, $\langle (A), (C) \rangle$);
   - Search_Patterns(root, $\langle (A), (C), (D) \rangle$);
   - Search_Patterns(root, $\langle (A), (C), (D), (E) \rangle$);

   As presented before, the variable 'root' represents the pattern tree root node.
2. For each query response, the suggested method calls are evaluated: Taking the first query, Search_Patterns(root, $\langle (A) \rangle$), we receive $\langle (B) \rangle$ as response. Each method call not already coded is then evaluated, checking if they could be foreseen with the received suggestions.
   As these methods are $\{(C), (D), (E)\}$, the suggested method $\langle (B) \rangle$ is tagged as useless.
3. For each method call being evaluated, we check whether it would have been coded automatically through a suggestion:
   Provided that the first query was not useful, the evaluation process advances, and method call (C) is tagged as not automated.
   The second query, Search_Patterns(root, $\langle (A), (C) \rangle$), is performed and we receive $\{(D), (E), (D, E)\}$ as suggestions, sorted by their confidence values. In this case, the method calls $\langle (D) \rangle$ and $\langle (E) \rangle$ are suggested individually. Moreover, as the method calls (D) and (E) are commonly used together, it also suggests $\langle (D, E) \rangle$, which means (D) followed by (E).

It is important to notice that $\langle(B)\rangle$ would also be returned, provided that the tree has the pattern $\langle(A),(B)\rangle$. However, as we have already evaluated this suggestion, we have customized it to provide only new patterns in the evaluation process. This customization avoids the reevaluation of an already evaluated pattern.

Another important aspect is that we evaluate each suggested method call individually instead of the suggestions as a whole. For example, suppose a suggestion has three method calls, and two of them are useful. In that case, we mark these as useful and the third one as useless (not the whole suggestion as useful or useless), making the evaluation process as fair and precise as possible.

This way, as the coded method calls (D) and (E) were foreseen, they are tagged as automated. In the same way, the suggested methods $\langle(D)\rangle$ and $\langle(E)\rangle$ are tagged as useful.

4. The next executed query is Search_Patterns(root, $\langle(A),(C),(D)\rangle$). This query does not suggest any new pattern, as method (E) was already automated.
5. The last query is Search_Patterns(root, $\langle(A),(C),(D),(E)\rangle$). As method (E) is not followed by any additional method call in the pattern tree, there is no new pattern to be suggested. The method (F) is then tagged as not automated, and the evaluation process is concluded.

At this moment, it is possible to summarize the evaluation result and calculate the metrics values. The method calls evaluation is presented in Table 2, where the last line shows an Automation Percentage of 50%. The first method call, (A), is not considered, as it would necessarily be hand-coded to motivate to run the first query.

The suggestions evaluation is presented in Table 3, where the last line shows a Correctness of 66,67%. After that, the F-Measure, defined in Formula 1, is calculated, resulting in a 57,16% value.

We used those dependent variables and the independent variables described

Table 2: Automation Percentage calculation

Method Call	Automated
A	N/A
C	No
D	Yes
E	Yes
F	No
Automation Percentage	50%

Table 3: Correctness calculation

Suggested Method	Useful
B	No
D	Yes
E	Yes
Correctness	66.67%

in Table 4 to answer the proposed research questions, along with the treatments applied in the independent variables.

3. Results and Discussion

Sections 3.1, 3.2, and 3.3 present the experimental results for the aforementioned research questions. As the experiments[12] were executed over five open-source projects, we first present the results for each open-source project individually and then analyze the results as a whole.

[12]Experiment data available at https://gems-uff.github.io/vcc/.

Table 4: Independent Variables.

RQ	Independent Variables	Treatments
RQ1	Amount of patterns	Variation of the amount of patterns from 1 to 20.
RQ2	Confidence threshold	Variation of the confidence threshold from 0% to 100% with increments of 10%.
RQ3	Number of commits	Variation of the number of commits using a sliding window of 50 commits with increments of 5 commits.

3.1. RQ1: In a code recommendation, how many frequent coding patterns should be presented?

This first study analyzes if we can define a number of suggestions that should be presented to developers. For each project, the code recommendation performance was compared, varying the amount of provided suggestions from 1 to 20.

Only the first 728 valid methods of the revisions selected for evaluation were considered to equalize the amount of data evaluated in each open-source project. This number was defined considering that the JUnit project, the one with the lowest amount of valid methods available for evaluation, contains exactly 728 valid methods.

Even though methods are the experimental objects of this evaluation, commits are the elements sequentially processed from the repository. As a commit can have many valid methods, some methods created in the same commit will be evaluated while others will be ignored when the aforementioned methods limit is reached. Since there is no temporal relationship between the classes and methods modified in a single commit, it is not possible to properly select these methods following a temporal criterion. Aiming at guaranteeing the experiment reproducibility, we needed to define a deterministic criterion to select the meth-

Table 5: Evaluation Statistics

Project	**Commits**	**Valid Methods**	**Eval. Methods**	**Applicability**	**Method Calls**
Commons IO	170	728	425	58.4%	3,504
Guava	69	728	274	37.6%	1,450
JUnit	180	728	414	56.9%	2,083
RxJava	140	728	347	47.7%	2,173
Spring Security	101	728	403	55.4%	3,005

ods from the last evaluated commit, which led us to implement this selection through the lexicographic order of the fully qualified method names.

Some statistics about this study are presented in Table 5. The first column shows the project names. The second column presents the total number of valid commits evaluated for each project. The third column gives the number of valid methods, while the fourth, the number of evaluated methods, i.e., the number of method bodies for which at least one suggestion was provided. The fifth column shows the Applicability, the percentage of valid methods that were evaluated. Finally, the sixth column presents the total number of evaluated method calls (i.e., the method calls coded inside the evaluated methods) that were verified if the code recommendation could be automated.

Figure 2 also shows the length of the recommended sequence for each project. As we can see, sequences with length one are predominant. Meanwhile, Figure 3 shows the length of all the detected patterns in each project, which includes the query length plus the recommendation sequence length. We can see that the pattern length 2 is predominant and length 3 is closely behind in some cases.

The results for Commons IO are presented in Figure 4. The curve shows that code recommendations automate more than 25% of the method calls coded in the evaluated method bodies. We can also observe a steep rise in Automation (from 17% to 25%, approximately) when the suggestion threshold varies between 1 and 10. However, when more than 10 suggestions are analyzed, the curve stabilizes, indicating that considering more than 10 suggestions may be

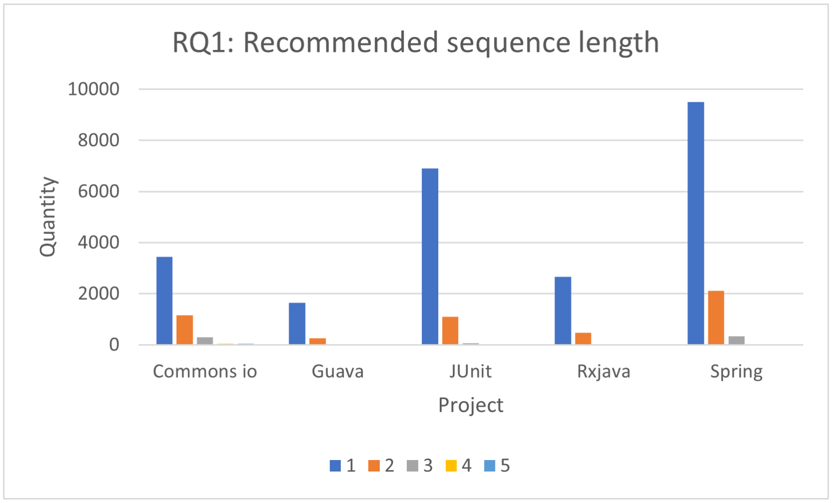


Figure 2: Recommended sequence length for RQ1 data

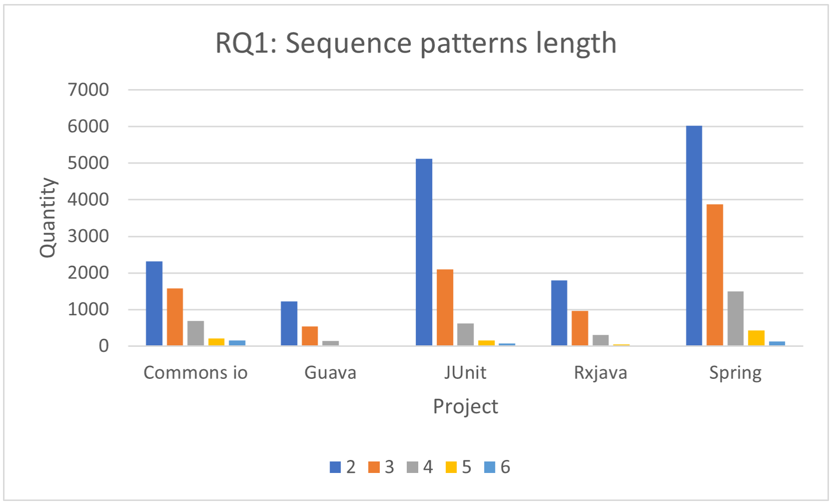


Figure 3: Patterns length for RQ1 data

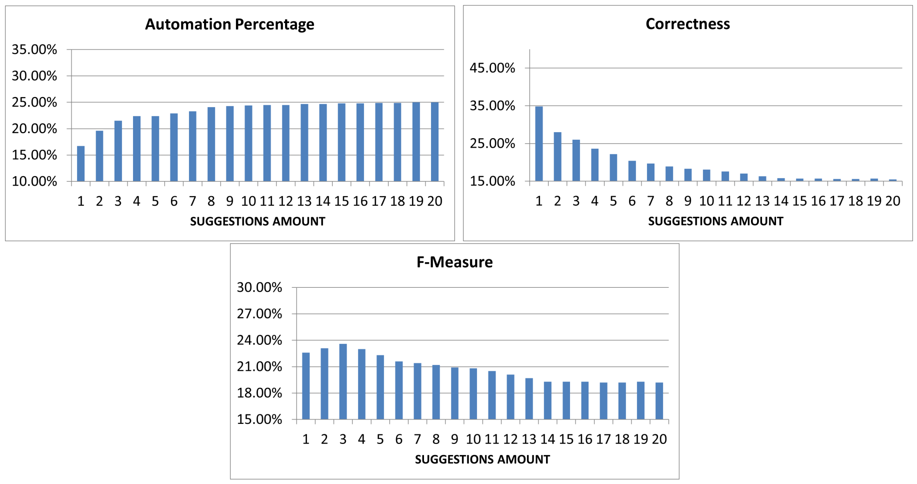


Figure 4: Automation, Correctness and F-Measure results for Commons IO

pointless, at least for this project. As we can also see, the correctness stabilizes after 8 suggestions and becomes steady after 14 suggestions. Finally, the best performance in terms of F-Measure is obtained when less than 5 suggestions are considered, with the best absolute result obtained when 3 suggestions are provided.

We obtained equivalent results for Guava, as illustrated in Figure 5, with the Automation values also reaching above 25%. The curve behavior is also similar, with stabilization after 8 suggestions. Regarding the Correctness curve, it is possible to notice that it presents a continuous decrease, as expected. Although, its worst result is still close to 20%, when up to 20 suggestions are analyzed. In the F-Measure curve, we can see that the maximum F-Measure values are obtained with 4 and 7 suggestions. Moreover, the results achieved between 3 and 9 suggestions are considerably better than the others.

The JUnit evaluation, illustrated in Figure 6, shows that the Automation and the Correctness curve behaviors are similar to the ones obtained in the previously presented projects. However, by analyzing the F-Measure curve, we observe that the results are more stable in this project, with only a smooth decrease when more than 15 suggestions are evaluated. Although, there are two

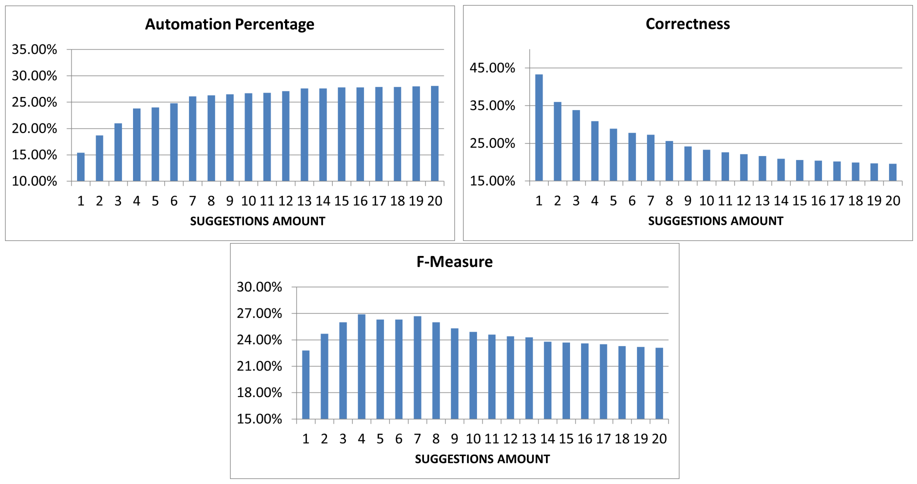


Figure 5: Automation, Correctness and F-Measure results for Guava

peaks when four and six results are provided. It is also worth mentioning that the Automation values were around 30%, with Correctness superior to 20% in almost all scenarios. This means that about one-third of the method calls would have been automated if 10 or more suggestions had been considered.

The results for the RxJava project are illustrated in Figure 7 and are very close to the ones presented by Commons IO and Guava projects. As expected, the F-Measure is also equivalent, presenting a continuous decrease. The best results are accomplished when two, three, or four suggestions are analyzed.

Finally, by analyzing the Spring Security results presented in Figure 8, we can observe an analogous performance when compared to the JUnit project. The Automation metric maintains a smooth rise, while the Correctness metric tends to stabilize. However, the F-Measure metric follows the declining tendency shown in other projects, however, in a softer way, with peaks in 3 and 5.

As expected, we observed a continuous increase in Automation in all projects when more suggestions are taken into consideration. At the same time, the Correctness decreases as the number of suggestions analyzed get bigger. The F-Measure curves are essential to analyze whether the Automation increase compensates for the Correctness decrease or not. Table 6 presents, for each

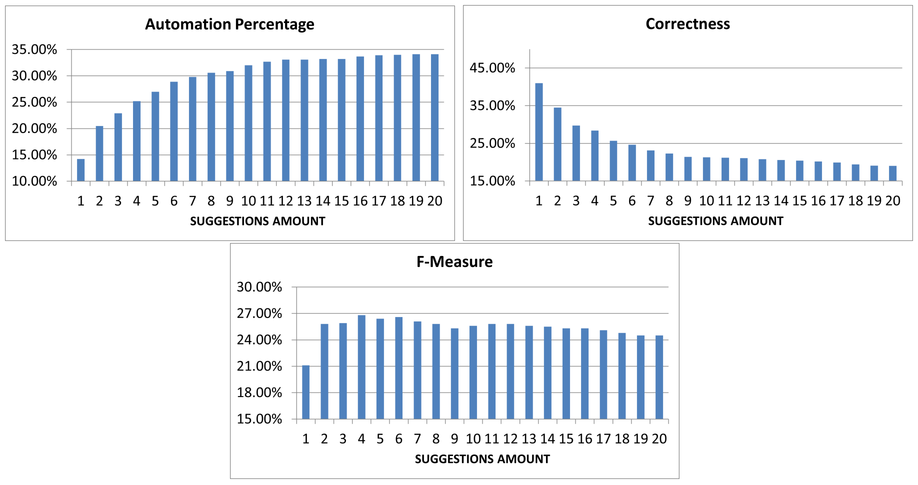


Figure 6: Automation, Correctness and F-Measure results for JUnit

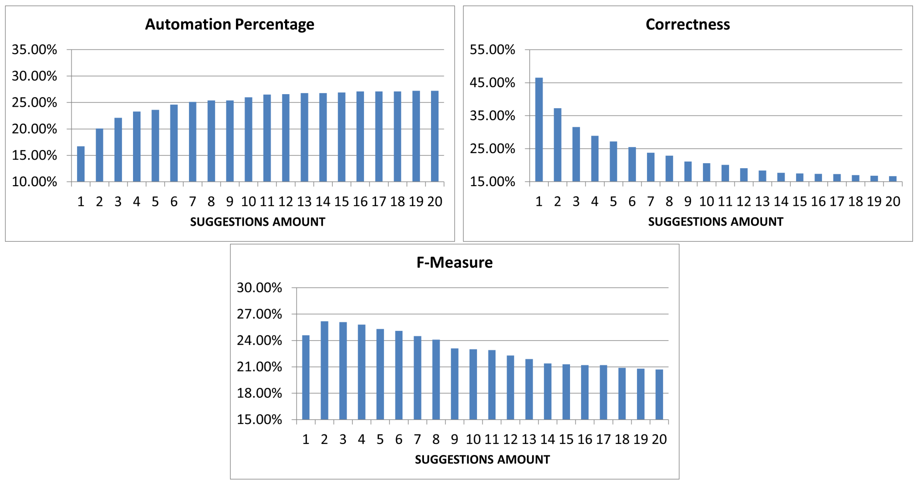


Figure 7: Automation, Correctness and F-Measure results for RxJava

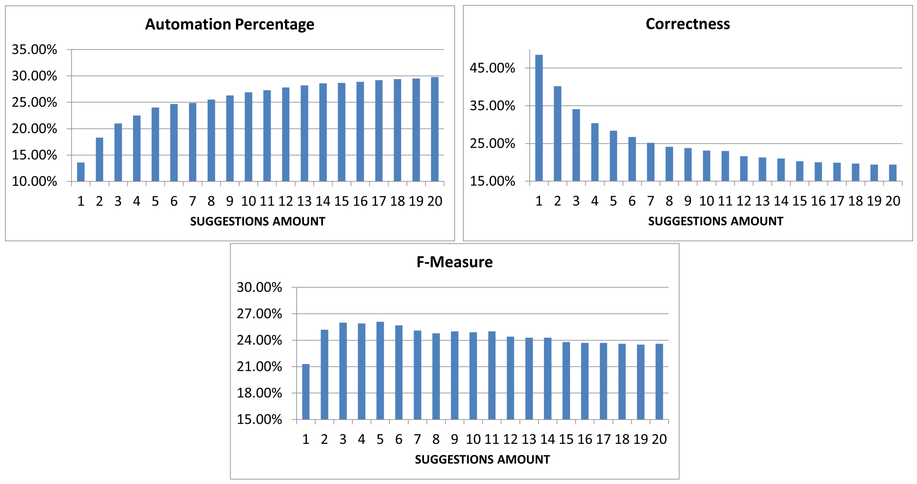


Figure 8: Automation, Correctness and F-Measure results for Spring Security

Table 6: F-Measure Results

Project	Best F-Measure	Suggested Amount
Commons IO	24.2%	3
Guava	26.9%	4
JUnit	26.8%	4
RxJava	26.2%	2
Spring Security	26.1%	5

project, the number of suggestions that delivers the highest F-Measure value. The first column shows the project names. The second column presents the highest F-Measure obtained for the project. Finally, the third column gives the number of suggestions evaluated that delivered this F-Measure result.

In Commons IO, Guava, and RxJava, it was possible to notice a significant decrease in F-Measure as more suggestions were analyzed. It indicates that a gain in Automation does not compensate for the Correctness decrease caused by presenting suggestions with lower confidence values. In JUnit and Spring Security, the F-Measure performance also decays when more suggestions are analyzed. However, its decreasing rate is much smoother, which indicates that the analysis of more suggestions may be helpful in some situations.

RQ1. In a code recommendation, how many frequent coding patterns should be presented?
Answer: The obtained results point out that the suggestions ranked in the *first five* positions are the ones that provide the best overall performance. Around 25% of method calls are automatically coded when these suggestions are considered.
Implications: Developers should seriously consider these suggestions since they present Correctness superior to 25% in four out of the five evaluated projects. Nonetheless, since the suggestions ranked after the fifth position can also help developers, we believe that the code recommendation suggestions should be paginated. This strategy would highlight the best suggestions while still allowing developers to navigate through the remaining suggestions on demand.

3.2. RQ2: What is the impact of filtering suggestions by their confidence instead of only ranking them?

This RQ aims at evaluating suggestions by confidence, using different threshold values, and analyzing the impact of these thresholds on the results. This way, we investigate if filtering suggestions by confidence improves the quality of code recommendation results. We have defined eleven confidence thresholds, from 0% to 100% confidence, with an interval of 10% between each analyzed threshold. It is important to notice that each threshold is a minimum confidence value. Thus, when the threshold is 0%, all suggestions are evaluated. When the threshold is 10%, only the suggestions with confidence greater or equal to 10% are evaluated, and so on. When the threshold is equal to 100%, only suggestions with exactly 100% confidence are evaluated.

Figure 9 also shows the length of the recommended sequence for each project, without filtering by thresholds. As we can see, sequences with length one are

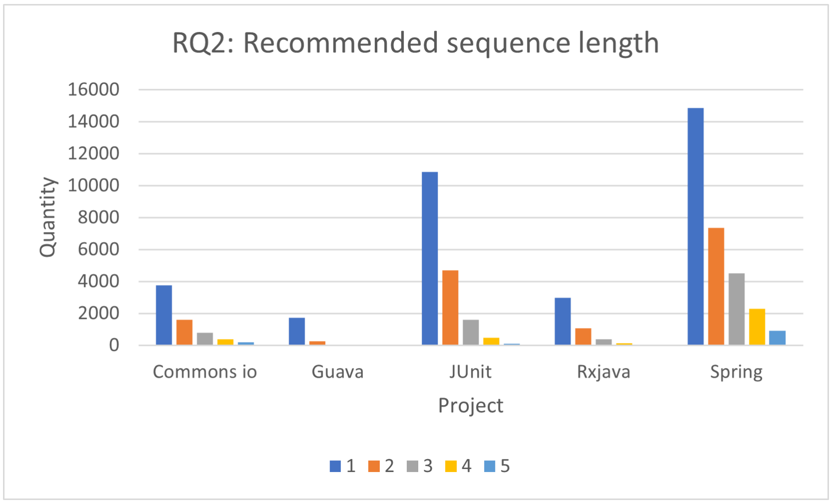


Figure 9: Recommended sequence length for RQ2 data

still predominant. Meanwhile, Figure 10 shows the length of all the detected patterns in each project, which includes the query length plus the recommendation sequence length. There is no longer a predominant length in this study and the length varies from project to project due to different coding patterns. Nevertheless, the recommendation sequence length of 1 is still predominant, as shown in Figure 9.

In addition, this study also analyzes the impact in the Applicability metric of filtering out the suggestion. When we filter out suggestions according to a threshold, we may have no suggestions, as there may not remain any suggestions with confidence that is superior to the threshold. Therefore, filtering suggestions may reduce the number of evaluated methods and impact the Applicability.

The results are presented using scatter plot charts in order to also show the Applicability metric. Each scatter plot crosses two dependent variables: the Applicability and one of the other metrics (Automation, Correctness, or F-Measure). Also, each chart presents 11 samples, and each one is an independent execution of the experiment for a different confidence threshold.

Following the same order from RQ1, we first evaluated Commons IO with the results presented by Figure 11. First of all, choosing a confidence threshold

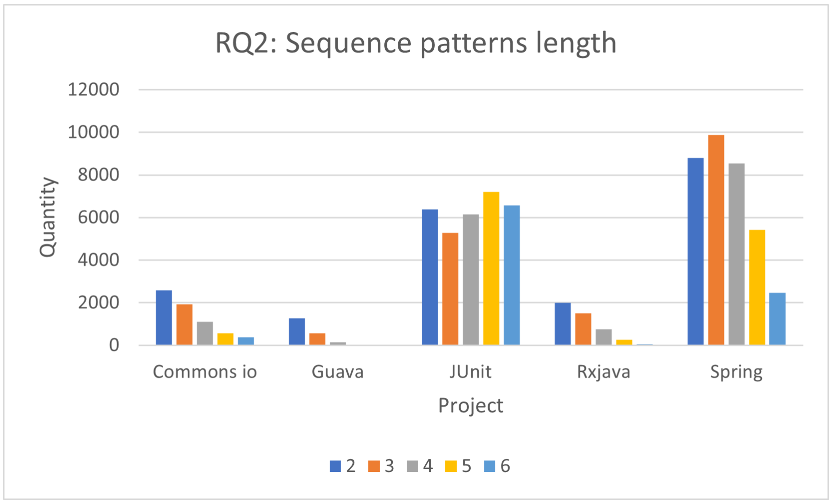


Figure 10: Patterns length for RQ2 data

superior to 60% reduces the Applicability to less than half of the total. On the other hand, choosing a confidence threshold inferior to 30% does not represent a notable increase in the number of evaluated methods. Nonetheless, the Automation increases when the threshold is inferior to 30%, while it remains almost stable for the other confidence values.

We can observe an outlier in this chart: the 70% confidence threshold. It is the only point where there is a noteworthy reduction in Automation when the confidence threshold is reduced. Note that when the threshold reduces, more method bodies are evaluated, increasing the Applicability. This kind of outlier may occur when the Automation obtained with the new evaluated method bodies is worse than the one obtained in the method bodies that were already being evaluated (with the threshold set to 80%), which may reduce the overall Automation. Moreover, there is a significant variation in the Correctness results, from a value slightly above 10% to almost 60%, when the confidence values varies between 0% and 100%. It is possible to observe that, as the confidence threshold becomes more restrictive, the returned Correctness suffers a substantial positive impact.

We obtain the F-Measure values by contrasting the steep Correctness in-

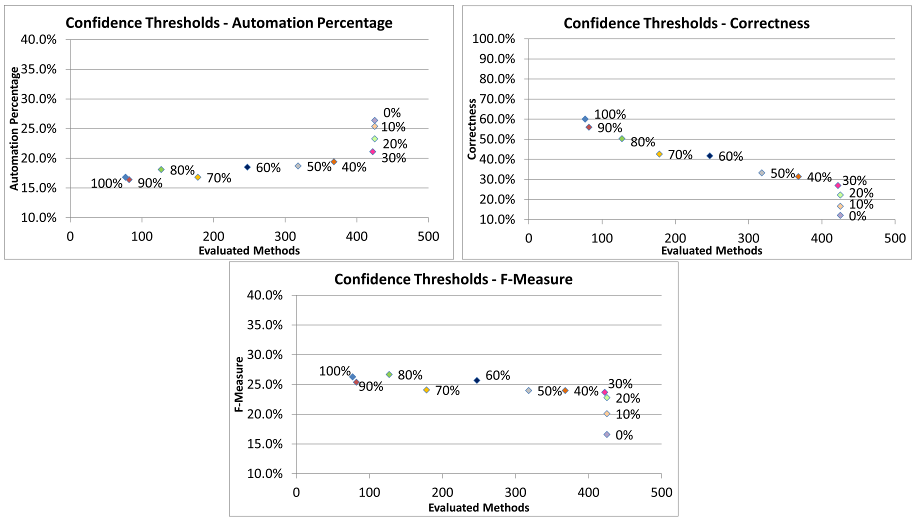


Figure 11: Automation, Correctness, and F-Measure results for Commons IO

creasing with the Automation. Apart from the result when the confidence threshold is 70%, caused by the aforementioned outlier, the other results between 60% and 100% confidence threshold are almost stable. While the highest F-Measure is obtained when the confidence threshold is 100%, many more methods are evaluated when the threshold is 60%. When looking at the results between 50% and 0% confidence, the threshold of 30% confidence represents an interesting value. Its F-Measure value is higher than those obtained with smaller confidence thresholds, while the amount of evaluated methods is almost the same. At the same time, many more methods are evaluated when the threshold is 30% than when it is 40% or 50%, whereas the F-Measure result remains stable. In other words, the threshold of 30% dominates almost all other thresholds in the range from 0% to 50%.

In the Guava project, as illustrated by Figure 12, the Automation behavior is considerably different when compared to the one observed in Commons IO. There is a continuous decrease in the Automation values as the confidence threshold gets more restrictive. However, the Automation decrease is reflected by a steep Correctness increase. Although, the most important information

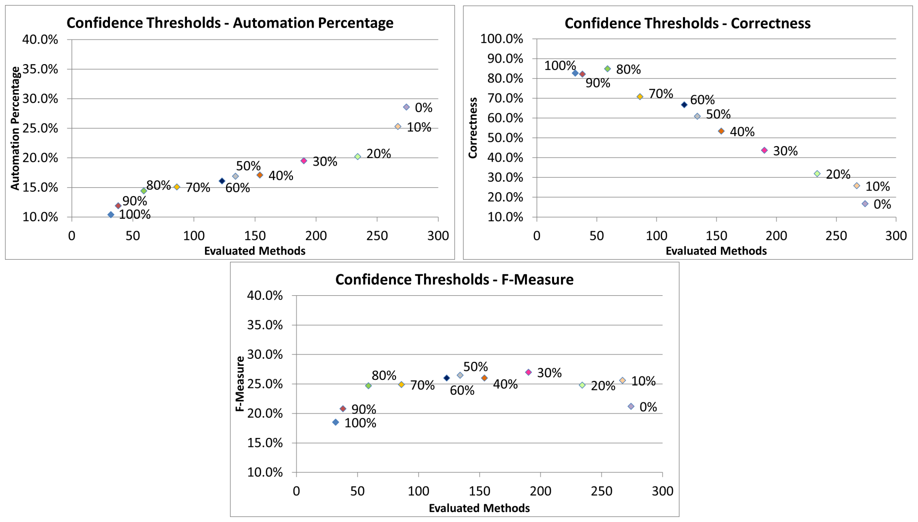


Figure 12: Automation, Correctness, and F-Measure results for Guava

in this chart is the obtained Correctness values, which reach more than 80%. This indicates that if a developer had only received suggestions with confidence values above 80%, approximately eight in ten suggested method calls would be correct. This behavior shows that filtering suggestions by confidence represent a powerful tool to customize the quality of the provided suggestions. Indeed, the increase in the Correctness is also opposed by an expected reduction in the number of evaluated methods.

We can see that the highest F-Measure value is obtained when the confidence threshold is 30%. This threshold also provides a high amount of evaluated methods. Using 30% as the threshold in confidence would decrease automation by 32%, decrease Applicability by 31%, but increase Correctness by 260% if compared to not filtering by confidence (0% as the threshold).

For JUnit, as illustrated by Figure 13, there is a smooth decrease in the Automation as the confidence threshold increases. This decrease is more intense when the evaluated thresholds vary between 0% and 20% confidence. Moreover, even with a threshold of 100%, more than 100 method bodies were evaluated in this project. It is also worth mentioning that almost 40% of Automation, the

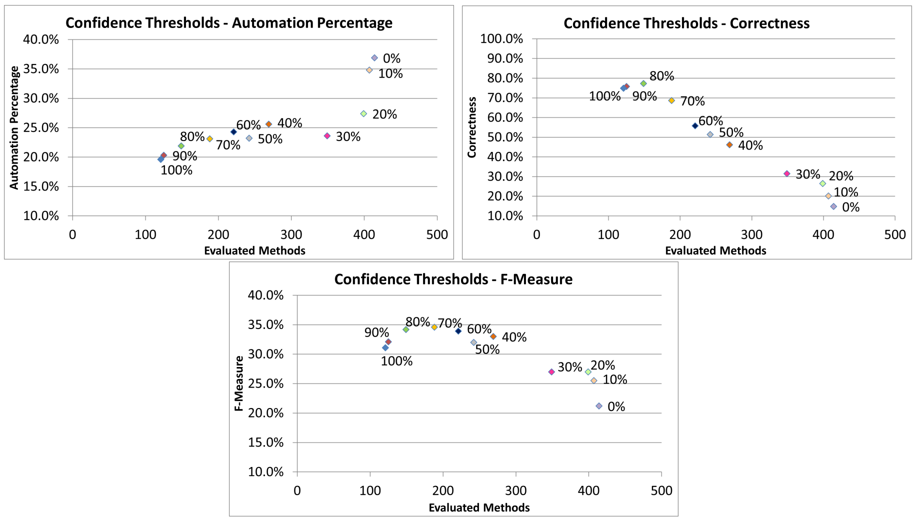


Figure 13: Automation, Correctness, and F-Measure results for JUnit

biggest value so far, was obtained when the threshold was set to 0%.

The middle graphic from Figure 13 shows that we have obtained Correctness values around 80% when the confidence threshold is set to 80% or more. When the threshold is set to values lower than 80%, there is a continuous decrease in the Correctness results. Meanwhile, the highest F-Measure values are found when the threshold is between 60% and 80%. Also, if the threshold is 40%, the F-Measure is slightly inferior, but the Applicability increases, indicating that this is also an interesting value. When the threshold is less than or equal to 30%, the F-Measure results are significantly lower. However, much more methods are evaluated in this condition.

In RxJava, as illustrated by Figure 14, the automation decreases when the confidence threshold increases, as expected. We can observe that the continuous Correctness increase when the confidence threshold becomes more restrictive. In this project, the Correctness also reaches values close to 80%, but only when the confidence threshold is 90% or 100%. The highest F-Measure value is achieved when the confidence threshold is set to 30%.

For Spring Security, Figure 15 shows that all confidence thresholds have more

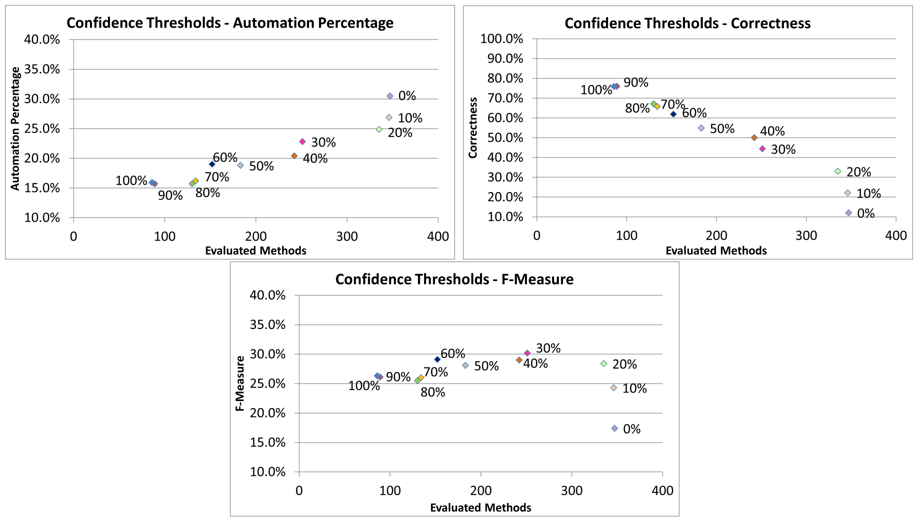


Figure 14: Automation, Correctness, and F-Measure results for RxJava

than 100 method bodies evaluated, except for 100%. On the other hand, we can observe a Correctness of 90% when the threshold is set to 100%, the highest value in the five evaluated projects. When the threshold is set to 90%, the Correctness still stays above 80%. Between the thresholds 0% and 80%, there is a continuous Correctness increase as the threshold also increases, as expected. The highest F-Measure values are obtained when the confidence threshold is set to 60%, where about 250 method bodies are evaluated. The threshold of 30% is also an interesting value, provided the F-Measure reduction is small, while there is a substantial Applicability increase.

Therefore, differently from our first research question, when the Applicability had a single value for each project, in this evaluation, it decreases as the confidence threshold increases. This way, the configuration of a confidence threshold must take this Applicability reduction into consideration.

Table 7 presents, for each project, the confidence threshold that delivers the highest F-Measure value and the obtained Applicability. The first column shows the project names. The second column presents the highest F-Measure obtained for each project. The third column shows the confidence threshold

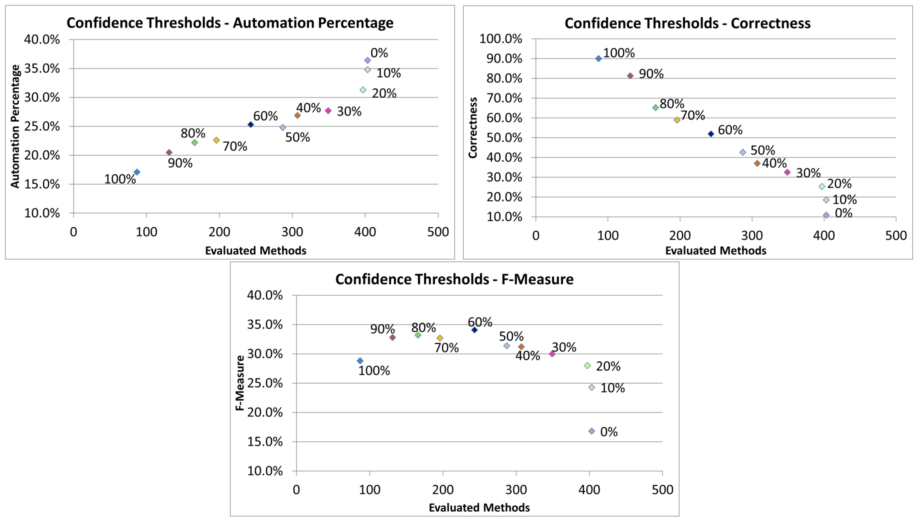


Figure 15: Automation, Correctness, and F-Measure results for Spring Security

that provided this highest F-Measure. Finally, the fourth column displays the Applicability obtained with the used confidence threshold.

The results presented in Table 7 represent an improvement in comparison to the ones obtained with the suggestion without filtering out by confidence. All the highest F-Measure values obtained when filtering by confidence are superior to the highest values obtained when filtering by suggestions amount, which can be observed when contrasting the results presented in Table 7 with Table 6.

Table 7: Highest F-Measure obtained and the respective Confidence and Applicability values

Project	Highest F-Measure	Confidence Threshold	Applicability
Commons IO	27.5%	100.0%	10.5%
Guava	27.0%	30.0%	26.1%
JUnit	34.6%	70.0%	25.8%
RxJava	30.2%	30.0%	34.5%
Spring Security	34.1%	60.0%	33.4%

However, there are two drawbacks in filtering suggestions by confidence: the Applicability reduction and the divergent results obtained for each project. While filtering only by suggestion amount provides all the best results between two and five suggestions, the best results were obtained with the confidence filter with 30%, 60%, 70%, and 100%. This divergence makes it hard to claim an ideal confidence threshold to be applied in other projects.

RQ2. What is the impact of filtering suggestions by their confidence instead of only ranking them?
Answer: Our initial conclusion is that it seems to be worth filtering out the suggestions by confidence, although the appropriate threshold varies according to the project where code recommendation is being used. However, the overall results indicate that applying a threshold confidence of 30% provides better outcomes than when no filter is applied, i.e., when the threshold is 0%, making 30% a conservative value that can be initially applied to every project and tuned subsequently.
Implications: Developers should consider filtering through confidence to select only the correct suggestions. The most precise results achieve Correctness values around 80% for all evaluated projects, except for Commons IO. This observation indicates that code recommendation tools could offer developers customization, where they would only receive suggestions with a high probability of being useful. However, if the developers want to receive a larger quantity of suggestions, they could reduce the confidence threshold and trust only in the pagination offered by the confidence ranking. Therefore, combining confidence filtering with the pagination idea mentioned in RQ1 provides a promising way to improve the user experience of developers that adopt code recommendation tools.

3.3. RQ3: Does the effectiveness of the sequential coding patterns degrade over time?

In the two previous studies, we assessed code recommendation performance in 728 method bodies. In this case, the source code was evolving while the patterns were not being updated. The goal of this study is the measurement of a possible performance loss caused by outdated patterns. This measurement allows us to investigate when a new code recommendation mining stage should be executed, updating the patterns.

In this study, we do not evaluate code recommendation over each of 728 methods as we did before. The commits were filtered, selecting only the valid commits, which have at least one new method body added and have at least two method calls. After that, the code recommendation was evaluated in commit windows, where each window was composed of 50 valid commits. The amount of windows varies in each project, as in this study. we evaluate the entire project history.

The commit windows are not mutually exclusive to provide smoother charts. The windows intersect, advancing five commits in each window. This way, our first commit window contains commits between the 1st and the 50th commit; the second commit window contains commits between the 6th and the 56th commit; the third commit window contains commits between the 11th and the 61st; and so on.

Moreover, this study is not intended to oppose Automation and Correctness, as in the previous two studies. We wanted to discover an ideal configuration value that would balance these metrics. Actually, Automation and Correctness tend to be influenced similarly by the source code evolution. Thus, this study only presents the Automation and the Correctness without the F-Measure calculation.

Table 8 presents the characterization of the projects in this study. The first column shows the project names. The second column presents the total number of valid commits evaluated for each project. The third column gives us the number of valid methods, while the fourth, the number of evaluated methods,

Table 8: Evaluation Statistics

Project	Commits	Valid Methods	Eval. Methods	Applicability
Commons IO	185	773	453	58.6%
Guava	257	5,905	1,334	22.6%
JUnit	181	728	414	56.9%
RxJava	167	5,174	2,897	56.0%
Spring Security	649	3,677	1,784	48.5%

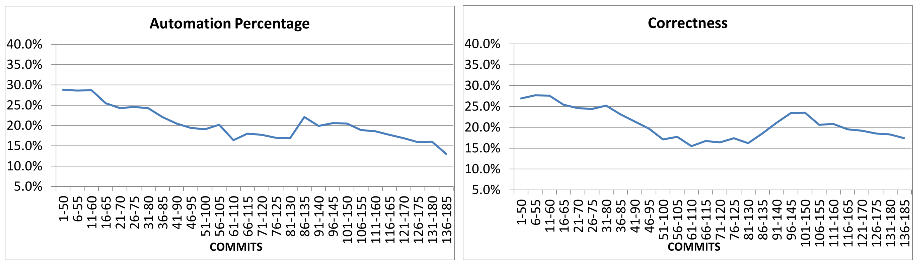


Figure 16: Automation and Correctness results for Commons IO

i.e., valid methods for which at least one code recommendation suggestion was provided. Finally, the fifth column shows the Applicability, i.e., the proportion between the evaluated methods and the valid methods.

In Commons IO, we evaluated 185 commits over 74 months. During this period, 1,197 method bodies were created, and 774 were modified. Our first evaluated metric is Automation, and the second is Correctness, presented in Figure 16. In this study, the horizontal axis represents the previously explained commit windows. The vertical axis represents the Automation for the first chart and the Correctness for the second chart.

The first chart shows an Automation performance loss as more commits are evaluated. This is the expected behavior since the source code is being changed in relation to the code used to extract the patterns. These changes could be the inclusion of new methods bodies, refactorings that changed method names, or even the simple inclusion or deletion of method calls in already existing

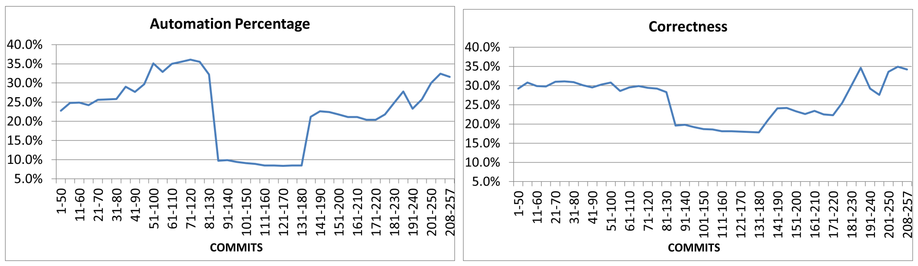


Figure 17: Automation and Correctness results for Guava

methods. These modifications could create new patterns, which are not being considered when the code recommendation pattern querying is being executed, impacting the Automation. The effective performance reduction started from the 61th commit, which would justify the re-execution of the code recommendation pattern mining stage at this moment, updating the patterns. The overall Correctness also reduces, but at a lower rate. This lower rate can be explained by the fact that, while the aforementioned code modifications may degrade some patterns, other patterns are still useful, provided that the code from where these patterns were extracted has not suffered changes that impacted the patterns.

In Guava, we evaluated 257 commits over 25 months. During this period, 13,031 method bodies were created, and 2,608 were modified. Figure 17 shows the Automation and Correctness values. These curves show very different performance than the obtained in the previously evaluated project, with some unexpected behaviors. In some situations, refactorings that include, change, remove, move or even copy a great amount of code in few commits may impact the evaluated metrics, positively or negatively. We looked individually at the commits evaluated in this study to explain these situations.

The first unexpected behavior is the Automation increases between the 1st and the 125th commits. In this case, a refactoring was made in the 47th commit, where several methods were renamed. These modifications impacted the performance of all the commit windows between 1-50 and 46-95. This happened because renamed methods are considered as new methods in Git, inserting a huge amount of method calls in a single commit. This is an artificial Automation result, provided the method calls already existed. However, we could not eliminate these situations from the evaluation. From the commit window 51-100 on, this 47th commit no longer impacted the results, and the Automation was raised.

The second unexpected behavior is the abrupt decrease in commit window 91-140. This result was caused by the inclusion of a great amount of unit test classes in a single commit. Apparently, this code was migrated from another repository, and, as it was not closely related to the code used in the code recommendation pattern mining stage, it caused a significant performance loss in the Automation. Finally, in the last analyzed commit windows, there were also some refactorings that renamed methods. However, in this situation, the effect was positive, increasing the Automation value. As previously mentioned, refactorings insert some noise in the evaluation, provided it is a sudden and artificial insertion of method calls. It is important to observe that this noise can be positive or negative, it all depends on whether the impacted method calls take part in codification patterns or not. If they take part, the effect is positive. Otherwise, the effect is negative. In the second chart from Figure 17, it is possible to observe a similar performance when compared to the Automation. The refactoring that affected the first commit windows did not reflect on the Correctness, but the other unexpected behaviors can be observed in the chart.

In JUnit, we evaluated 181 commits over 35 months. During this period, 1,403 method bodies were created, and 520 were modified. Figure 18 shows the Automation and Correctness values. In the first curve, despite having a period with a lower Automation between the windows 61-110 and 101-150, the Automation values are almost stable, varying between 25% and 30%. JUnit Correctness is also stable. There is only one single peak in commit window 46-95. This stability, presented in both metrics, indicates that, in some projects, the patterns can remain valid for a longer period of commits than in others. What can also be responsible for this behavior is that the evaluation period is not very long, comprising only 35 months, different from what we observed in Commons IO, where the evaluation comprised 74 months.

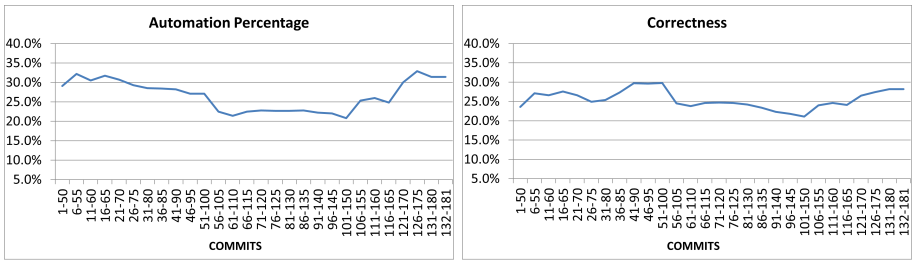


Figure 18: Automation and Correctness results for JUnit

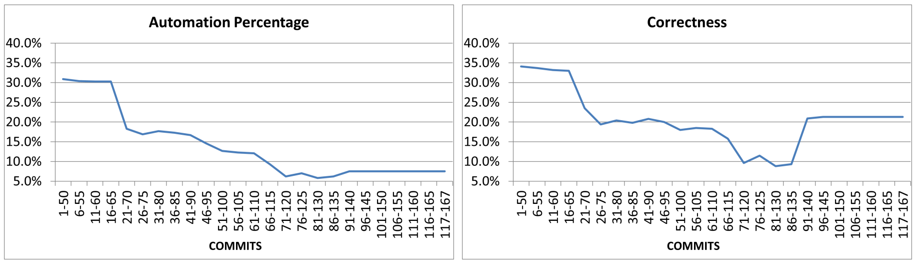
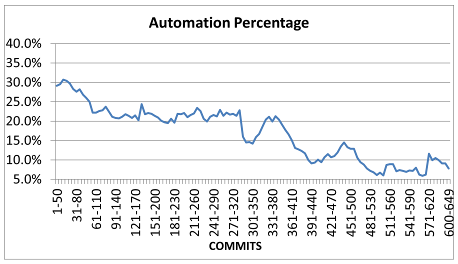


Figure 19: Automation and Correctness results for RxJava

In RxJava, we evaluated 167 commits over 10 months. During this period, 8,860 method bodies were created, and 1,197 were modified. Figure 19 shows the Automation and Correctness values. The first curve shows an intense decrease in the code recommendation performance as the patterns become outdated. We believe that this intense performance loss may be credited to the intense insertion of new code in this project. The evaluation period comprised just 10 months, although 8,860 new methods were added during this short period of time.

When we look at the Correctness values, the behavior is analogous. However, in the commit window 96-145, two significant refactorings were made, impacting more than 4,000 method bodies. The first refactoring renamed 30 classes, and the second renamed two packages, affecting almost 400 classes. These refactorings created a side effect, where these classes were considered as new classes, with new methods bodies. Thus, the methods were selected for evaluation, artificially impacting the Correctness. It is important to notice that the impact



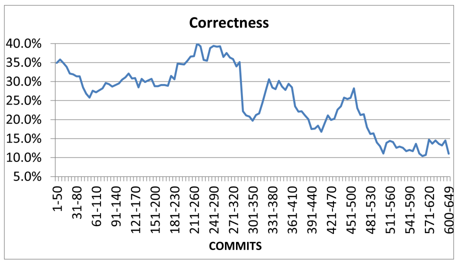


Figure 20: Automation and Correctness results for Spring Security

of this refactoring was so big that the curve became constant from the commit window 96-145 onward.

In Spring Security, we evaluated 649 commits over 76 months. During this period, 6,989 method bodies were created, and 3,026 were modified. The Spring Security project delivered the most relevant results of this study, as illustrated in Figure 20. Its repository provided more than 600 valid commits, where a great number of methods were added, allowing us to see the code recommendation performance variation in the long term. The chart shows a continuous trend of Automation decrease. The initial performance is around 30%, while at the end of the analysis, the performance varies around 5% to 10%. Another important observation to be made is that the curve is not monotonically decreasing. There are plenty of intermediary peaks where the performance temporarily increases. However, with a large number of commits being evaluated, it is possible to see that the decreasing trend is maintained.

This behavior endorses the analysis made for the previously presented projects. If we concentrate the analysis on a smaller amount of commits, we may perceive that the Automation remains stable or even increases. Although, this is likely to be only a local result, probably influenced by refactorings that included or modified a great amount of code in few commits.

As in the previously evaluated projects, we can see that the Spring Security Correctness tends to be more stable than the Automation. However, despite the intermediary peaks, there is also a clear declining trend in Correctness, especially after the commit window 281-330. Table 9 presents the initial and

Table 9: Automation Historical Variation

Project	**Initial Automation**	**Final Automation**
Commons IO	28.8%	13.0%
Guava	22.8%	31.6%
JUnit	29.2%	31.4%
RxJava	32.1%	7.5%
Spring Security	29.0%	7.8%

the final Automation value for each project. While the projects Commons IO, RxJava, and Spring Security presented a substantial Automation decrease, we observed an overall stabilization in projects Guava and JUnit. Guava, in fact, presented an increase. We also observed that the amount of added methods and the length of time might also influence the results beyond the number of commits. With the obtained results, we could observe the overall declining trend and that the intermediary abrupt ups and downs do not change this global declining trend.

RQ3. Does the effectiveness of the sequential coding patterns degrade over time?

Answer: The effective performance reduction starts after the 50th commit, justifying the re-execution of the code recommendation pattern mining stage at this moment, updating the patterns. The overall Correctness also tends to reduce after the 51th commit, but at a slower rate due to still having somewhat useful patterns.

Implications: We can state that an outdated pattern can have a major influence over code recommendation performance. It is important to define a policy to update the code recommendation patterns periodically, guaranteeing that the patterns reflect the code of the project being developed. Refactorings have a great influence over code recommendation results and should considered when a pattern update policy is being defined. Thus, it is important to re-execute the code recommendation pattern mining stage after a major refactoring is made.

3.4. Threats to Validity

Despite the effort made to provide a consistent evaluation, we have identified some threats to validity in the experiments. This section analyzes these threats from the perspective of the four types of validity: internal, construct, external, and conclusion.

First of all, code recommendation demands that developers specify a pattern support prior to the obtainment of the codification patterns. This configuration varies from project to project, and we could not find in the literature an appropriate formula to calculate it in advance, according to the project's characteristics. Hence, we needed to define the projects' support empirically, tuning it manually when necessary. This manual configuration impacts the internal validity, and to reduce this threat, we defined the closest possible values for all projects. Moreover, the impact on code recommendation results caused by the refactorings made in the source code of the evaluated open-source projects also affected the internal validity. These refactorings could not be removed from the evaluated commits, as Version Control Systems do not atomically identify renamings, considering these actions as deletions followed by additions.

We identified two threats regarding construction validity. The first is the possibility of a mistake with the decision of mapping productivity and quality with the metrics Automation and Correctness, respectively. The second is the chance of the proposed experimental methodology not being an adequate representation of a real usage scenario of code recommendation. Although we recognize the possibility of such threats, we have no concrete evidence that they have actually occurred in our case.

Regarding the open-source projects we have evaluated in this work, we needed a fully compilable source code to extract the codification patterns.

This is a trivial task for a real usage scenario, although we needed to checkout old open-source project revisions, which made this task very time-consuming. Nonetheless, a fair and correct analysis required access to all of the projects' historical dependencies, including old versions of external libraries. We could not obtain these versions for many projects, which prevented us from assessing more than five projects. This limited amount of projects impacted the external and conclusion validity of the experimental evaluation. In the former, we could not guarantee that these projects are a representative set of all the other open-source Java projects. In the latter, because we could not run statistical tests over the results.

4. Related Work

In this section, we discuss works [6][12][16][17][18][19][20][21] that present strategies for collecting and recommending source code patterns.

Hindle et al. [12] use natural language processing to model programming languages to discover appropriate code completion suggestions. The authors claim that programming languages, in theory, are complex, flexible, and powerful, but the programs that people actually write are mostly simple and rather repetitive, having predictable statistical properties that statistical language models can capture. Using this idea, source code repositories are lexically analyzed after comments removal, and every textual element is mapped to tokens. These tokens are organized using an n-gram model, which statistically estimates how likely tokens are to follow other tokens through conditional probabilities, generating a language model. Finally, when a new code is under development, the previous two tokens coded are used in an attempt to guess the next token. The obtained tokens are ranked using their probabilities and presented to the developer.

Laerte et al. [4] proposed Vertical Code Completion (VCC), a code completion approach that goes beyond the existing syntax-based approaches. Their approach aims at providing more sophisticated suggestions strongly related to the sequence of lines of code being developed. It takes the sequence of lines already coded into consideration to suggest new lines to be coded using sequential pattern mining techniques to obtain these suggestions. First, the entire source code is analyzed to discover recurring sequential patterns representing frequently coded line sequences. After that, during the coding stage, the sequence of lines already coded is matched to the beginning of one of the previously obtained patterns, and the remainder of the pattern is automatically suggested.

Nguyen et al. [18] proposed an improvement to the approach of Hindle et al. [12]. Instead of using a strict lexical model, they introduce the Statistical Semantic Language Model for Source Code, where lexical elements are mapped to semantic tokens. Each token stores semantic information, such as role (variable, operator, data type, function call, keyword, etc.), scope, and also dependencies to other tokens. As in Hindle et al. [12], an n-gram model is adopted, but in this case, it is extended to an n-gram type model, where the functionality of a code is captured and used to influence token probabilities.

Raychev et al. [21] proposed a structured prediction to source code by representing the code as a variable dependency network. Their approach trains a probabilistic model by using existing data to predict the properties of new programs. Each JavaScript variable is represented as a single node, while their pairwise interactions are modeled as a conditional random field. This field is then trained to predict the types and names of all variables within a snippet of code.

Gu et al. [19] proposed DeepAPI, an approach based on deep learning that allows users to query in natural language for certain API. Their approach learns the sequence of natural words in a query and relates to a sequence of associated APIs by encoding the word sequence into a context vector to generate the API recommendation.

Nguyen et al. [20] also proposed a new approach, Dnn4C, that uses Deep Neural Network models to incorporate syntax and type contexts in a program to complement existing lexical code elements to predict the next code token. The authors associate code tokens with syntactic annotations to determine the syntax context and the code with the annotations for the type context. These contexts are extracted and translated to vectors, together with the lexical vectors, and are passed as input to their predicting model.

Although we could find several approaches in the literature, they do not evaluate important aspects of the code recommendation that affect the usage of all of them. Among these aspects, we could cite the ideal number of offered recommendations, the usage of filters to show only recommendations that have a minimal confidence, and the validity of recommendations in the face of software evolution. Thus, the main contribution of this paper is a detailed study of those aspects of code recommendation and their implications.

5. Conclusion

This work presented a set of studies about sequential code patterns recommendation. Nowadays, multiple works in the literature aim at recommending code from previously written code. However, there was little knowledge about how these patterns could be used effectively in code recommendations.

The first contribution of this paper is the development of an experimental infrastructure that allowed the evaluation of code recommendation over the entire history of widely-known open-source projects. For a matter of comparison, some existing work [5] assessed only 10 commits of each project, reaching 31 method bodies evaluated in the most analyzed project. In this work, considering only the Spring Security project, we assessed 649 commits. In RxJava, we assessed 2,897 method bodies.

With the availability of this infrastructure, we could evaluate the overall code recommendation effectiveness according to different perspectives. In our first study, where we investigated the impact of filtering out the code recommendation suggestions by their rank, we identified that the top five ranked suggestions are the ones that deliver the best results. However, the suggestions ranked in subsequent positions could also be useful, which led us to state that code recommendations can take advantage of a pagination structure.

Our second study filtered out suggestions according to their confidence values. This evaluation showed us that the code recommendations that deliver the best results are the ones with the highest confidence values, which indicates that confidence is indeed an appropriate metric to classify the code recommendation patterns. Moreover, we could observe that, with a restrictive confidence filter, it is possible to reach correctness values above 80%.

The third study evaluated the code recommendation performance as the source code evolves. We were able to identify that during the first evaluated commits, when the patterns are still up-to-date, the Automation metric is around 30%. In addition, we observed that the code recommendation performance degrades as the source code evolves since patterns become outdated with this evolution.

Regarding the patterns applicability, we only explored code recommendation patterns extracted and applied in the same project in the current evaluation. However, as many of these patterns involve external libraries, they can be applied in other projects. A very interesting evolution of our work would be the classification of these patterns according to the libraries they invoke. This could allow providing code recommendations across projects. A potential future work would be be to analyze the effect of working with smaller sequences (blocks). Also, another future work would be to classify the recommendations and also detect cloned code fragments. Lastly, adding control structures to the recommendation is also an interesting future work.

6. Acknowledgments

Funding: This work was supported by CNPq [grant numbers: 306137/2017-8 and 310444/2018-7], FAPERJ [grant number: E26/010.101250/2018], and CAPES.

References

References

[1] R. Pressman, Software Engineering: A Practitioner's Approach, McGraw, 2009.

[2] R. Robbes, M. Lanza, How program history can improve code completion, in: IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, 2008, pp. 317–326. doi:10.1109/ASE.2008.42.
URL http://dx.doi.org/10.1109/ASE.2008.42

[3] M. Bruch, Eclipse code recommenders, available in http://www.eclipse.org/recommenders/ (2012).

[4] L. L. N. da Silva Junior, T. N. de Oliveira, A. Plastino, L. G. P. Murta, Vertical code completion: Going beyond the current ctrl+space, in: Brazilian Symposium on Software Components (SBCARS), Natal, RN, Brazil, 2012, pp. 81–90.

[5] L. L. N. da Silva Junior, A. Plastino, L. G. P. Murta, What should i code now?, Journal of Universal Computer Science 20 (5) (2014) 797–821.

[6] R. Hill, J. Rideout, Automatic method completion, in: IEEE International Conference on Automated Software Engineering (ASE), Washington, DC, USA, 2004, pp. 228–235. doi:10.1109/ASE.2004.19.
URL http://dx.doi.org/10.1109/ASE.2004.19

[7] R. Holmes, G. C. Murphy, Using structural context to recommend source code examples, in: International Conference on Software Engineering (ICSE), New York, NY, USA, 2004, pp. 117 – 125.

[8] D. Mandelin, L. Xu, R. Bodík, D. Kimelman, Jungloid mining: Helping to navigate the api jungle, SIGPLAN Not. 40 (6) (2005) 48–61. doi:

10.1145/1064978.1065018.
URL http://doi.acm.org/10.1145/1064978.1065018

[9] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, T. N. Nguyen, Graph-based pattern-oriented, context-sensitive source code completion, in: International Conference on Software Engineering (ICSE), Piscataway, NJ, USA, 2012, pp. 69–79.
URL http://dl.acm.org/citation.cfm?id=2337223.2337232

[10] N. Sahavechaphan, K. Claypoolr, Xsnippet: Mining for sample code, in: International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Portland, OR, USA, 2006, pp. 413–430.

[11] S. Thummalapenta, T. Xie, Parseweb: a programmer assistant for reusing open source code on the web, in: International Conference on Automated Software Engineering (ASE), Atlanta, GA, USA, 2007, pp. 204–213.

[12] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: International Conference on Software Engineering (ICSE), Piscataway, NJ, USA, 2012, pp. 837–847.
URL http://dl.acm.org/citation.cfm?id=2337223.2337322

[13] J. Han, M. Kamber, Data Mining: Concepts and Techniques (3rd edition), Morgan Kaufmann, 2011.

[14] R. Kohavi, et al., A study of cross-validation and bootstrap for accuracy estimation and model selection, in: International Joint Conference on Artificial intelligence (IJCAI), San Francisco, CA, USA, 1995, pp. 1137–1145.

[15] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Addison-Wesley Professional, 1994.

[16] J. Jacobellis, N. Meng, M. Kim, Cookbook: In situ code completion using edit recipes learned from examples, in: Companion Proceedings of the 36th

International Conference on Software Engineering, New York, NY, USA, 2014, pp. 584–587. `doi:10.1145/2591062.2591076`.
URL `http://doi.acm.org/10.1145/2591062.2591076`

[17] T. Kinnen, Supporting reuse in evolving code bases using code search, Master's thesis, Technical University Munich, Munich, Germany (2013).

[18] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, A statistical semantic language model for source code, in: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE), New York, NY, USA, 2013, pp. 532–542. `doi:10.1145/2491411.2491458`.
URL `http://doi.acm.org/10.1145/2491411.2491458`

[19] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep api learning, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 631–642. `doi:10.1145/2950290.2950334`.
URL `http://doi.acm.org/10.1145/2950290.2950334`

[20] A. T. Nguyen, T. D. Nguyen, H. D. Phan, T. N. Nguyen, A deep neural network language model with contexts for source code, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 323–334.

[21] V. Raychev, M. Vechev, A. Krause, Predicting program properties from "big code", in: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 111–124. `doi:10.1145/2676726.2677009`.
URL `http://doi.acm.org/10.1145/2676726.2677009`