On the Uniqueness of Code Redundancies

Bin Lin, Luca Ponzanelli, Andrea Mocci, Gabriele Bavota, Michele Lanza
REVEAL @ Faculty of Informatics — Università della Svizzera Italiana (USI), Switzerland

***Abstract*—Code redundancy widely occurs in software projects. Researchers have investigated the existence, causes, and impacts of code redundancy, showing that it can be put to good use, for example in the context of code completion. When analyzing source code redundancy, previous studies considered software projects as sequences of tokens, neglecting the role of the syntactic structures enforced by programming languages. However, differences in the redundancy of such structures may jeopardize the performance of applications leveraging code redundancy.**

We present a study of the redundancy of several types of code constructs in a large-scale dataset of active Java projects mined from GitHub, unveiling that redundancy is not uniform and mainly resides in specific code constructs. We further investigate the implications of the locality of redundancy by analyzing the performance of language models when applied to code completion. Our study discloses the perils of exploiting code redundancy without taking into account its strong locality in specific code constructs.

***Keywords*-code redundancy; code completion; empirical study**

I. Introduction

Code redundancy, namely identical parts of code occurring multiple times, is common in software projects [1], and manifests itself in different forms. At a coarse-grained level, developers may explicitly duplicate code snippets with different intentions, for example to break through given programming language limitations, or to construct reusable coding templates [2]. While literature often suggests that this kind of redundant code, called *code clones*, is to be avoided as it can lead to code bloat, not all code redundancies are harmful [3]. Moreover, numerous practical applications leveraging code redundancy have been implemented for different purposes, such as locating bugs [4], supporting refactoring [5], detecting plagiarism [6], and supporting code completion [7].

A particular kind of redundancy, considering code at the token level, has been the subject of recent studies, and proven to be effective for numerous applications. To understand how redundant software is, at the token level, Gabel and Su [8] fragmented source code into fixed-length sequences (*i.e.,* token-level n-grams) and measured uniqueness of software by quantifying the sequence redundancy. The authors examined 6,000 projects and found that software is highly repetitive when the sequences are short, *e.g.,* given sequences of six tokens, more than half of the code is redundant. Also, Tu *et al.* [9] reported on the *localness* of software, showing that code exhibits repetitive forms in local contexts at the file level, *i.e.,* repetitions of a specific n-gram localized in few files.

Hindle *et al.* [7] showed that source code is “natural”, which means it is highly repetitive and predictable, even more than natural language.

This does not come as surprise: Unlike natural language, programming languages are more constrained by syntax, and these constraints are likely to correlate with repetitiveness. Moreover, Ray *et al.* [10] studied a large set of bug fix commits from 10 Java projects, and investigated the relationship between buggy code and code “naturalness”. As a result, they found that buggy code is less “natural”.

However, some constructs (*e.g.,* method and class declarations) are more constrained than others (*e.g.,* sequences of method calls) and thus one might wonder if every part of the source code is equally redundant. Currently, there is no detailed analysis regarding this particular matter. Understanding where code redundancy resides is important as it might improve the performance of applications that leverage code redundancy.

Hindle *et al.* [7] developed a simple code completion engine for Java based on an n-gram language model and examined their engine on five Apache projects: Ant, Maven, Log4J, Xalan, and Xerces. For each project, they trained a trigram model from the corpus of the tokenized source code, and used two tokens to predict the next token. They combined this approach with the default Eclipse code recommendation engine, and the results suggested that an n-gram language model can effectively improve the performance of the recommendation engine. Since different parts of source code might have different levels of redundancy, we are interested in investigating whether the n-gram model based completion performs equally well in different parts of code, from which we can further infer whether unevenly distributed redundant code impacts the applications leveraging code redundancy.

We address this problem with two different studies performed on a large-scale dataset composed of 2,640 Java project repositories. First, we analyze the overall redundancy rate of the source code and then compare it with the redundancy rates of 12 source code constructs (*e.g.,*`import` and `class` declarations, `catch` blocks, etc.). Our results suggest that although code redundancy is common in software, it is very localized in specific code constructs (*e.g.,* in `package` declarations). Then, we explore the influence of the significantly different code redundancy rates observed for the code constructs on the performance of the language model based code completion. We find that while the language model is very accurate when recommending code tokens belonging to typically redundant constructs, its performance strongly decreases when suggesting tokens related to poorly redundant code constructs. In essence, our findings highlight the importance of considering the strong locality of code redundancy when exploiting it.

Structure of the paper. We describe our problem context in Section II. Section III explores the redundancy of different code constructs, and Section IV focuses on an application, *i.e.,* the language model based code completion, to analyze the impact of the unequal code redundancy. Section V discusses the threats that affect the validity of our studies. Finally, Section VI illustrates the related work and Section VII outlines the conclusions.

II. Study Context

Table I summarizes the dataset used for our studies.

TABLE I: Dataset Statistics

	Overall	**Per Project**		
		Mean	Median	St. Deviation
Java files	1,461,290	554	237	1,123
Tokens	1,079,112,838	4,087,549	152,873	970,681
ELOC	146,886,573	55,639	20,892	125,762
Forks	314,594	119	25	412
Stars	864,227	327	50	1,194
% Java code	-	91.5	92.4	5.6

The *study context* consists of 2,640 open source Java projects hosted on GitHub, mined on Nov 21 2016 using the following constraints:

- **Programming language.** Projects need to have at least 80% of their effective lines of code (ELOC, lines of code without comments and empty lines) [1] written in Java. Java is the reference language for the infrastructure used in this study.
- **Activity level.** To exclude inactive projects, they need to have at least one commit in the three months preceding the data collection.
- **Popularity.** The number of forks[1] and stars[2] of a repository are two proxies for its popularity on GitHub. Forking a repository means getting a copy of the repository to implement changes not affecting the original project. Starring a repository allows GitHub users to express their appreciation for the project. Projects with less than ten stars and no forks are excluded from the dataset, to avoid the inclusion of likely irrelevant projects.
- **Size.** Projects must have at least 50 files and 5,000 ELOC. Again, the goal is to filter out irrelevant projects.

2,714 projects satisfy these constraints. We removed 74 projects that could not be correctly parsed by the tools we use (*e.g.,* Antlr, srcML). Table I reports descriptive statistics for size and popularity of the selected projects, showing a high degree of diversity of the dataset in terms of both these attributes. The complete list of projects considered in the studies is available in our replication package [11].

III. Study I: Source Code Redundancy

The *goal* of the study is to assess to what extent source code is redundant both when considering it as a whole (*i.e.,* the complete code base of a software project) as well as when focusing on specific code constructs (*e.g.,* when considering `import` declarations). The *context* of the study consists of the 2,640 Java projects detailed in Section II. While previous research already investigated the source code redundancy phenomenon [8][7][12], to the best of our knowledge there are no studies (i) run on such a scale[3], and (ii) analyzing the redundancy rate of different code constructs.

A. Research Questions

We aim at answering the following research questions (RQ):

RQ$_1$: *How redundant is source code?* This RQ aims at assessing to what extent source code is redundant. In the context of RQ$_1$ we analyze the code redundancy when considering the complete code base of a software project. RQ$_1$ will corroborate/confute the findings of previous studies reporting the high redundancy of source code [8], [7], [12]. Moreover, the results of RQ$_1$ serve as a reference for RQ$_2$, in which we assess the redundancy rate of different code constructs.

RQ$_2$: *To what extent are different code constructs redundant?* This RQ sheds light on the redundancy rate of different code constructs, missing in the current literature. Knowing the redundancy rate of different constructs is necessary to design techniques and tools assuming the high repetitiveness of code, *e.g.,* language models supporting code completion [7]. While these techniques work fairly well in general, they might perform poorly when dealing with specific parts of the code being always unique or rarely repetitive.

B. Data Extraction

To answer our research questions and measure code redundancy we adopt a methodology similar to the one used by Gabel and Su [8]. In particular, for each project P_i in our dataset, we perform the following steps:

Code sequencing. We tokenize the P_i's source code and extract tokens' sequences of length l (*i.e.,* token-level l-grams). The sequences extraction is performed in each P_i's Java file starting from its first token (*e.g.,*`package`) and using a sliding window of length l advancing at steps of one token. For example, assuming $l = 9$, from the code statement `for(i=0; i<n; i++)` the following sequences are extracted: "`for(i=0; i<n`", "`(i=0; i<n;`", "`i=0; i<n; i`", "`=0; i<n; i++`", "`0; i<n; i++)`". We analyze code redundancy for sequences of different lengths by varying l from 3 to 60 in steps of 3 (*i.e.,* 3, 6, 9, etc.). This process resulted in the extraction of over 1 billion tokens and around 1 billion sequences. Also, we tokenize each sequence at two different abstraction levels: *no abstraction* and *token type only*. For the *no abstraction* approach, the sequence `if (a > b)` is tokenized into the list of tokens "`if`, `(`, `a`, `>`, `b`, `)`". For token types only, the same sequence is tokenized into a list of lexical classes "`IF`, `LB` (left bracket), `ID` (identifier), `GT` (greater than), `ID`, `RB` (right bracket)". Token types are generated with Antlr4 [13].

[1] https://help.github.com/articles/fork-a-repo/

[2] https://help.github.com/articles/about-stars/

[3] The study by Gabel and Su [8] is run on 6,000 projects. However, they sample a limited number of tokens (∼1,500) from each project, while we analyze all tokens from each project (on average, 432,290 tokens per project).

Sequence redundancy detection. We mark each of the extracted sequences as either "redundant" (*i.e.,* there exists at least one repetition of the sequence in P_i) or "not redundant" (*i.e.,* the sequence is unique in P_i). Differently from Gabel and Su [8], we look at code redundancy within the scope of each single project: We do not consider a sequence $s_j \in P_i$ as redundant if it also appears in another project P_k. This choice is dictated by the fact that some "practical" applications of code redundancy make more sense when only considering the code from a single project.

For example, in the case of the language model used to support code completion [7], the authors built a different model for each system. This is needed since each project has its own domain and thus its own vocabulary. Since in our second study we investigate if and how the differences in the redundancy rate of different code constructs impact the performance of such techniques, we decided to focus on the code redundancy within each single project.

Linking tokens to code constructs. To address RQ$_2$ we need to identify the code construct to whom the analyzed tokens belong. To this aim, we parse the source code by relying on the srcML infrastructure [14] and assign each token to one of the twelve code constructs listed in Table II (in **bold** the tokens belonging to the specific code construct). We extract matched code constructs without considering whether they contain other constructs or not. For example, the code sequence "`for(int i=0; i<n; i++)`" is classified as a "for control" construct, although it contains a variable declaration "`int i=0;`".

TABLE II: Identified code constructs

Construct	Example
package	**`package com.abc;`**
import	**`import java.io.*;`**
if condition	**`if(a == b)`** {...}
while condition	**`while(a > n)`** {...}
for control	**`for(int i=0; i<n; i++)`** {...}
class declaration	**`class Square extends Shape`** {...}
method declaration	**`public int getX`** {...}
method call	**`System.out.println("Hello!");`**
method body	`public int getX` **`{ return x; }`**
variable declaration	**`int x = 0;`**
catch parameter	`catch` **`(Exception e)`** {...}
catch block	`catch (Exception e)` **`{break;}`**

While other code constructs could be extracted, we maintain that the number and diversity of constructs considered in our study to be sufficient to observe differences in the redundancy rate of different parts of the source code.

C. Data Analysis

We answer RQ$_1$ by reporting box plots depicting the *redundancy rate* of tokens belonging to sequences (i) tokenized by using both the *no abstraction* and the *token type only* representation and (ii) having different lengths l. The redundancy rate is computed as the number of tokens belonging to sequences marked as "redundant" divided by the total number of tokens in the analyzed sequences [8].

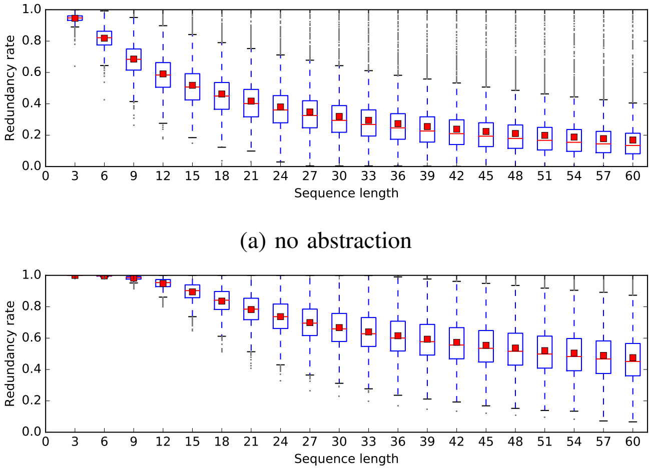
To answer RQ$_2$, we compare via box plots the redundancy rate of tokens belonging to different code constructs. We also statistically compare the redundancy rate of the different constructs by exploiting the Mann-Whitney test [15] with results intended as statistically significant at $\alpha = 0.05$.

To control the impact of multiple pairwise comparisons (*e.g.,* the redundancy of tokens belonging to the `package` construct is compared against the redundancy of tokens belonging to the *if condition*, the *while condition*, etc.), we adjust p-values using the Holm's correction [16]. We also estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure [17] for ordinal data. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \le |d| < 0.33$, medium for $0.33 \le |d| < 0.474$, and large for $|d| \ge 0.474$ [17].

D. Results

We discuss the achieved results according to the two RQs.

1) RQ$_1$: How redundant is source code? Fig. 1 shows the boxplots depicting the tokens' redundancy rate for sequences having increasing lengths, both when *no abstraction* is used as well as when considering only the *token type*.



(a) no abstraction

(b) token type only

Fig. 1: Redundancy rate for sequences having different lengths

The red square in each boxplot represents the mean value of the distribution. We draw three main conclusions:

1) **When considering the project's code as a whole, source code is highly redundant**. For tokens belonging to sequences of length 3, the redundancy rate is very high, also without abstraction (*i.e.,* when considering exact copies of the 3 tokens), with a median of 0.95. High redundancy rates (> 0.5) are generally observed for sequences of length up to 15.
2) **The longer the sequences the lower the redundancy rate.** The trend depicted in Fig. 1 is clear, and highlights a sort of logarithmically decreasing function when observing the median values of the boxplots from left (short sequences) to right (long sequences).

This is to be expected, since it is much less likely to find duplications of long sequences as compared to shorter ones.

3) **When only considering the token type, the redundancy rate substantially increases.** Again, this is an expected results, considering the abstraction level introduced when only looking for token types. For example, while the two sequences `if(a > b)` and `if(c > d)` are considered as different in the *no abstraction* approach, they are considered as redundant from the *token type only* perspective.

Our findings thus corroborate the observations by Gabel and Su [8], and confirm the high redundancy of source code. Our next RQ investigates *where* the redundancy is, when looking more closely at the source code.

2) RQ$_2$: To what extent are different code constructs redundant? Fig. 2 shows the boxplots depicting the redundancy rate for different types of code constructs when considering sequences of different lengths without applying abstraction[4]. Concerning the impact of the sequence length on the redundancy rate, all code constructs follow the same trend previously observed for the whole project. However, it is evident that tokens belonging to different types of code constructs do not exhibit the same redundancy rate.

Fig. 3 compares the redundancy rate of code constructs when the sequence length is 9. The redundancy rate of the whole project is also shown in Fig. 3 (boxplot on the right), serving as a baseline for comparison. The main message highlighted by Fig. 3 is that the redundancy rate of different code constructs significantly differs. This is clear, for example, when comparing `import` declarations (median=0.97) and `while` conditions (median=0.57). Such strong differences are confirmed by the statistical analysis in Fig. 4[5].

In the heatmap in Fig. 4, a white block indicates that the difference in terms of redundancy rates of two code constructs is not statistically significant (adjusted p-value $\geq$ 0.05). Blocks with four different grayscale values from light to dark represent a significant difference accompanied by a negligible, small, medium and large effect size, respectively. Confirming what can previously observed, `import`, `package`, and `catch` parameter constructs are significantly more redundant than other constructs. All of the statistical comparisons result in a significant difference, and 74% of the cases have medium or large effect sizes. Thus, the statistical analysis confirms the high variability of redundancy rate for different types of code constructs.

Up to now we considered as redundant a token from a sequence repeated at least once [8]. We also investigated the frequency of redundant sequences (*i.e.,* how many times sequences are repeated in the code). Indeed, the frequency with which a sequence is repeated in the code impacts techniques leveraging code redundancy, like language models that need to learn what the likely sequences of code tokens are.

The analysis being computationally expensive, we performed it on 30 randomly selected projects. Fig. 5 shows, using a logarithmic scale, the frequency of the 144,494 unique tokens sequences of length 9 extracted from Microsoft Thrifty.

While the code redundancy, measured as explained in Section III-B, is quite high for this system (0.80)—indicating that most of tokens belong to redundant sequences—the median frequency of the redundant sequences is just 2, with a third quartile of 4. This means that at least 75% of the redundant sequences in this system are repeated at most 4 times in the code, while there are very few sequences repeated hundreds of times (leading to the long tailed distribution in Fig. 5). The most redundant sequence (743 repetitions) is `(org.apache.thrift.-protocol.`, generally used in `catch` statements (*e.g.,*`catch (org.apache.thrift.protocol.-TProtocolException))`. Other very frequent sequences are those related to `import` statements (*e.g.,*`import com.microsoft.thrifty.schema.`—161 times).

When looking at the frequency of redundant sequences, most of the code redundancy is in very specific parts of the code. Indeed, we observed a long tailed frequency distribution shown in Fig. 5 for all the 30 selected systems[6]. Such a characteristic of code redundancy can strongly impact approaches leveraging it. This is the focus of our second study.

IV. Study II: Language Models & Code Completion

The *goal* of this study is to investigate the performance of an n-gram language model aimed at recommending the next code token to write (*i.e.,* the n^{th} token) given $n-1$ written tokens. Basically, we assess the accuracy of the language model proposed by Hindle *et al.* [7] for code completion both overall (*i.e.,* when used in any part of the source code) as well as when focusing on specific code constructs.

A language model is a probability distribution that estimates how often a sentence occurs in a textual dataset. Language models are widely employed in many domains such as speech recognition and code completion. The n-gram model is one of the most commonly used language models and it determines the probability of having a word w_i given the previous n-1 words. Such a probability is denoted by $p(w_i|w_{i-1}, w_{i-2}, \ldots, w_{i-n+1})$, where $w_{i-n+1}, \ldots, w_{i-1}, w_i$ are n continuous words. The probability that w_i follows $w_{i-n+1}, \ldots, w_{i-2}, w_{i-1}$ is estimated by training the language model on a training test, composed of textual documents. When applying the language model to software-related tasks, the training set is composed of code documents.

The most common way to evaluate the performance of a n-gram model is instead to run it on an previously unseen set of test documents (again, code documents in the case of software-related tasks) known as the test set, and assess its ability to predict the actual word w_i following a sequence of $n-1$ consecutive words extracted from the test set (this process is repeated for many sequences) [18].

[4] Results for the *token type only* approach is in our replication package [11].

[5] Results for sequences of different lengths are in [11].

[6] Results available in our replication package [11].

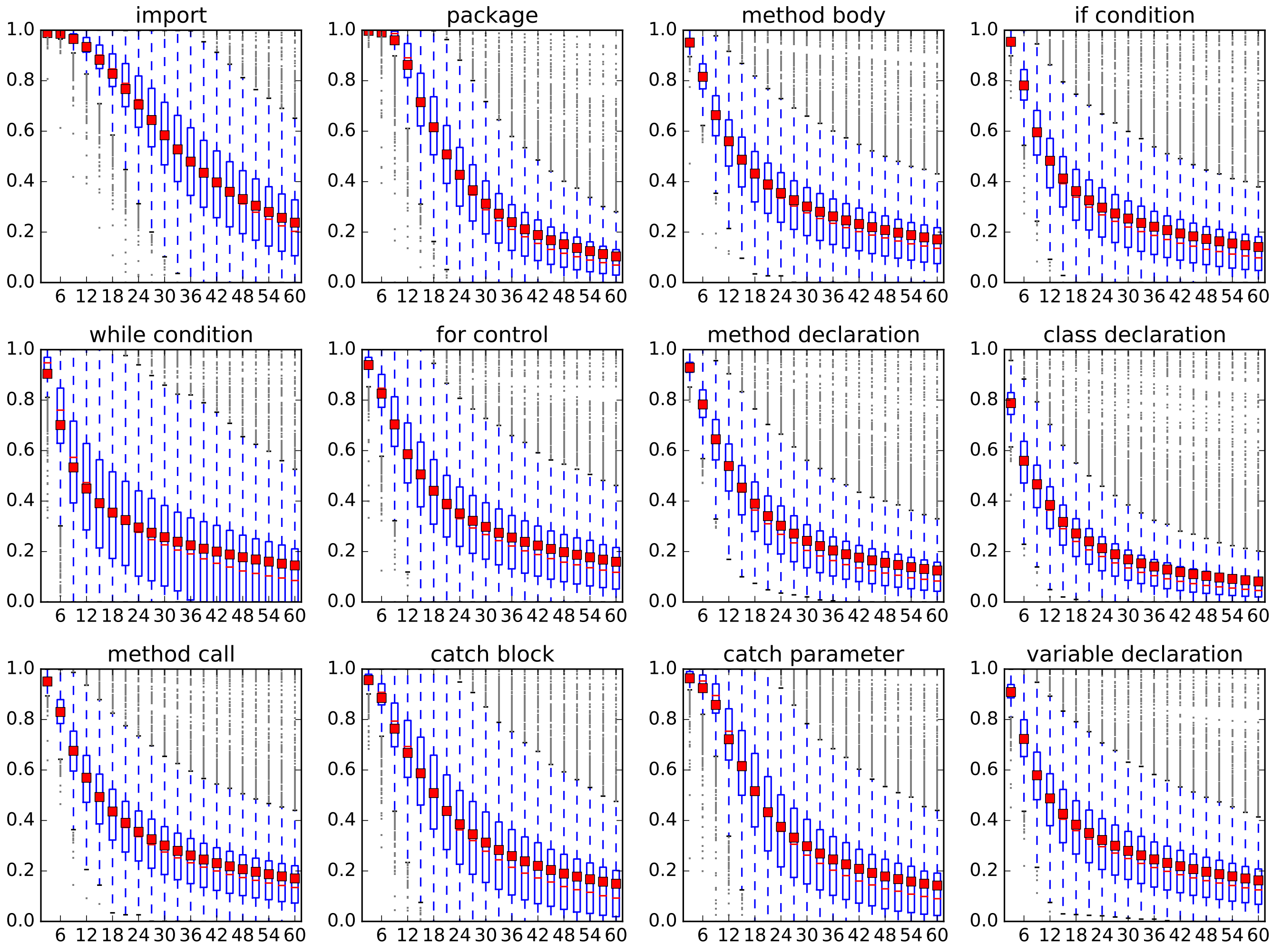


Fig. 2: Redundancy rate for different types of code constructs when no abstraction is applied.

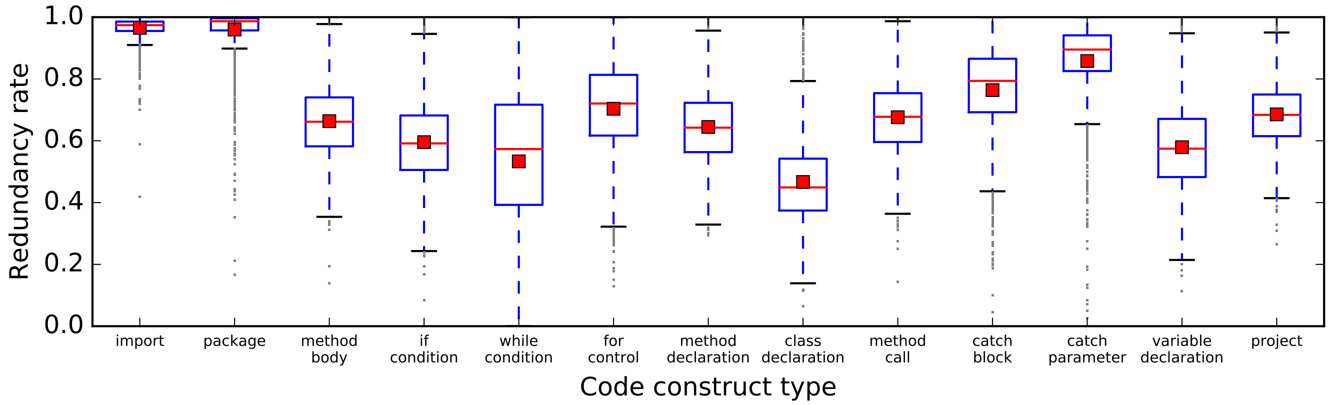


Fig. 3: Redundancy rate of different code constructs when no abstraction is applied and the sequence length is 9.

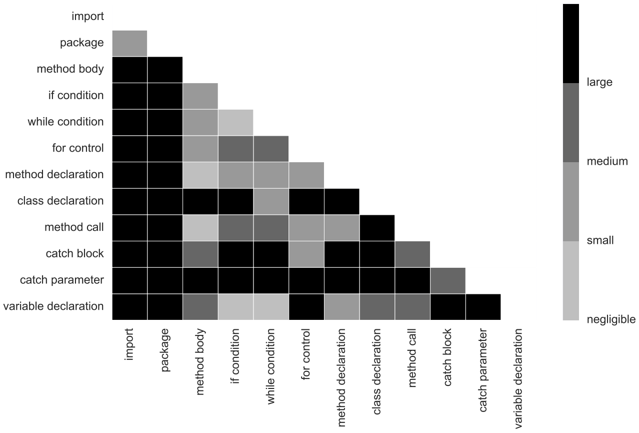


Fig. 4: Statistical comparisons for the redundancy rates of different types of code constructs for sequences of length 9.

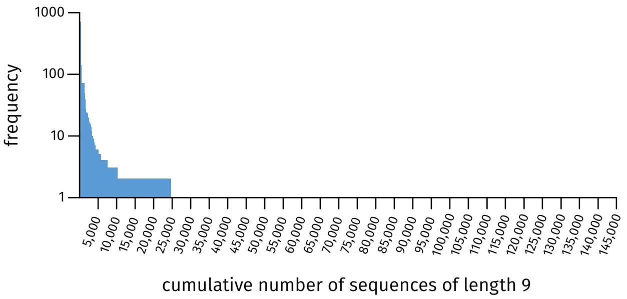


Fig. 5: Microsoft Thrifty: Cumulative frequency for sequences of length 9.

Our conjecture is that the substantially different redundancy rates observed in Study I for the different code constructs might influence the performance of the language model and suggest its applicability only to specific parts of the code (*i.e.,* the ones having high redundancy).

A. Research Questions

The study aims at answering the following RQs:

RQ$_3$: *How effective is the language model in supporting code completion?* This RQ assesses the performance of the n-gram language model when applied to code completion. The evaluation approach followed in this study is similar to the one of Hindle *et al.* [7], where they evaluated the a 3-gram language model on five systems. We (i) run a much larger evaluation involving 2,640 subject systems, and (ii) study the impact of the n parameter on the model performance.

RQ$_4$: *How effective is the language model in supporting code completion for different code constructs?* This RQ investigates whether and how the predictive performance of the n-gram language model varies on different code constructs (*i.e.,* the same 12 constructs considered in Study I). To the best of our knowledge, this is the first study running such an analysis.

RQ$_4$'s findings will shed some light on the importance of considering the strong locality of code redundancy.

B. Data Extraction

To answer our research questions we perform the following steps for each project P_i in our dataset:

1. Create training and test sets. We randomly split the P_i's Java files into a training set accounting for 90 % of P_i's ELOC, and a test set composed by the remaining 10%. Our training/testing strategy is different with respect to the one adopted by Hindle *et al.* [7]. They randomly selected 200 files from each subject system, using 160 for training and 40 for testing. Such an approach does not consider the whole project's code base, and does not provide a clear indication of the "amount of code", intended as ELOC, actually used for training and testing; indeed, this strongly depends on the size of the specific files selected for training and testing.

2. N-grams extraction. As done in Study I, we tokenize the P_i's Java source code in both training and test set. Note that *no abstraction* is used in this study, since we want to support code completion by recommending to the developer the exact token to write given the previous $n - 1$ tokens (as done in [7]). We vary n from 3 (the original value used in [7]) to 15 at steps of one (*i.e.,* 3, 4, 5, etc.).

3. Linking tokens to code constructs. As done in Study I, we map each token to one of the code constructs listed in Table II. This allows us to answer RQ$_4$, by reporting the performance of the language model when predicting tokens belonging to different code constructs.

After collecting this data, for each project P_i and for each considered value of n, we train the language model on the n-grams extracted from the P_i's training set, obtaining a model $M_{P_i,n}$. Then, we run $M_{P_i,n}$ on the n-grams extracted from the P_i's test set, trying to predict for each n-gram the n^{th} token given the $n - 1$ continuous tokens preceding it. Overall, this resulted in the testing of the language model on a minimum of 104,953,587 n-grams (for $n = 15$) and a maximum of 106,726,166 (for $n = 3$).

C. Data Analysis

We answer RQ$_3$ by showing boxplots reporting the percentage of tokens correctly predicted by the language model (*i.e.,* its accuracy) for each of the experimented n values. Since the language model provides a ranked list of tokens likely following the provided $n - 1$ tokens (with the most likely on top), we compute the model accuracy when considering the top t recommendations it generates, varying t from 1 to 10 at steps of one. For example, when considering $t = 1$, we consider a recommendation as correct only if the correct token appears in the first position of the ranked list, while for $t = 10$ the recommendation is tagged as correct if the correct token appears in the top-10.

Concerning RQ$_4$, we compare the accuracy of the language model when predicting tokens belonging to different code constructs. This is done via boxplots and statistical tests, following the same procedure adopted in RQ$_2$.

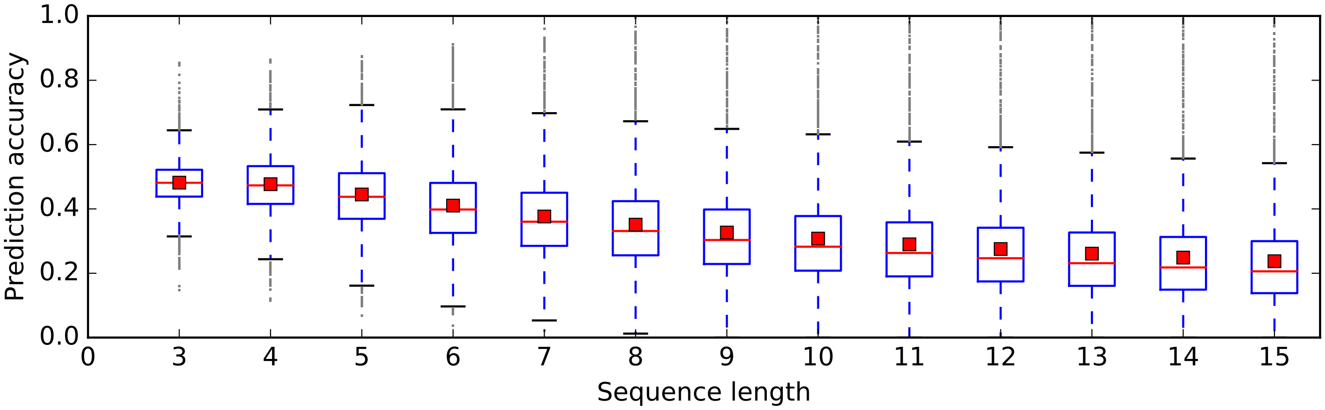


Fig. 6: Prediction accuracy rates of the language model when supporting code completion (top 1 recommendation).

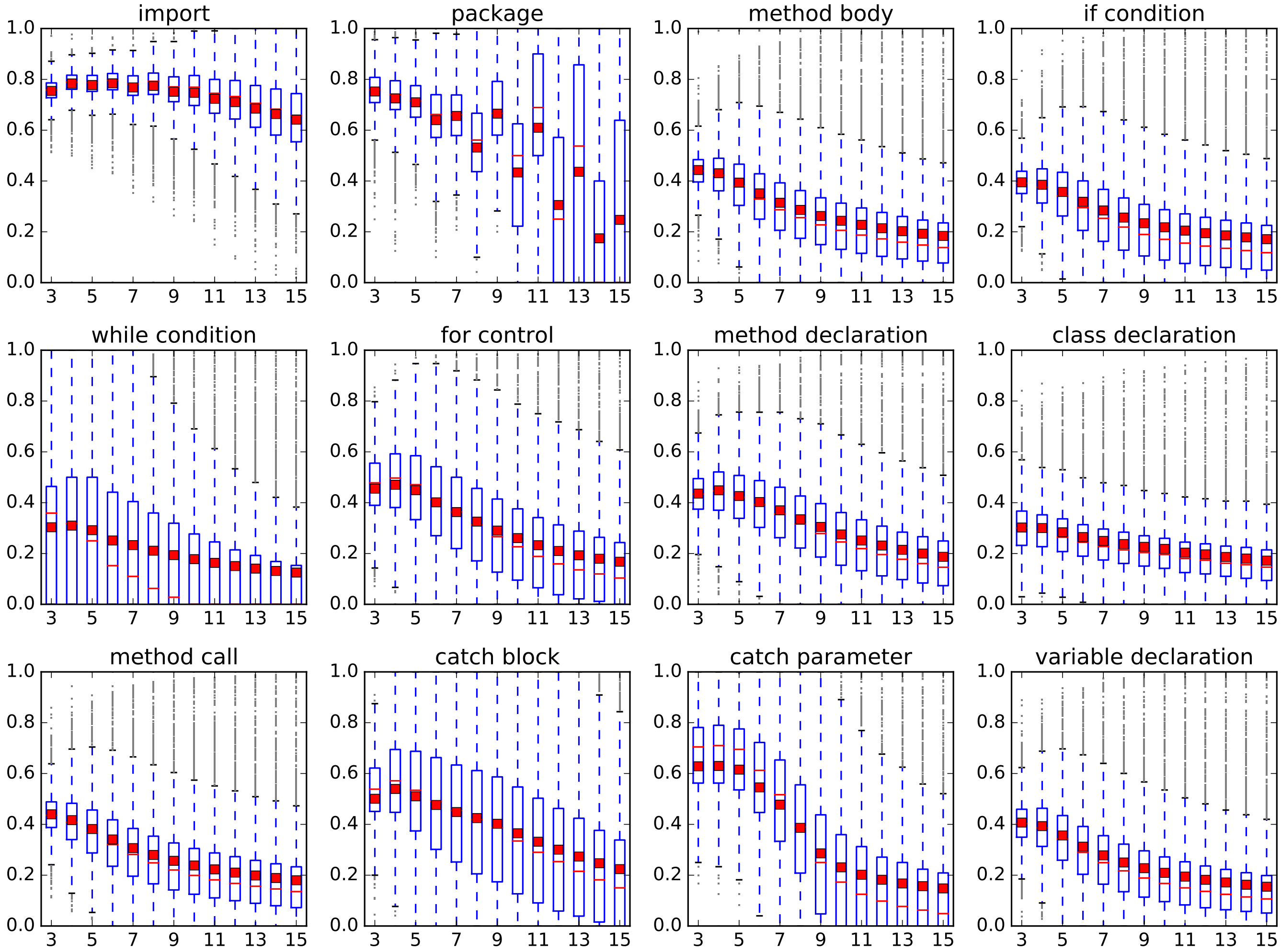


Fig. 7: Accuracy of the language model when supporting code completion on different constructs (top-1 recommendation)

D. Results

1) RQ3: How effective is the language model in supporting code completion? Fig. 6 shows the accuracy of the n-gram language model when used to support code completion. In particular, Fig. 6 reports the accuracy achieved when only considering the top ranked suggestion (*i.e.,* the token having the highest probability of following the $n-1$ tokens) when varying n between 3 and 15.

The language model achieves its maximum accuracy (median = 0.48) when $n = 3$, and its performance regularly decreases with the increasing of n (worst accuracy achieved at $n = 15$). Such a finding might seem counterintuitive. Indeed, one would expect that the more information is fed into the language model, *i.e.,* the higher the number of $n-1$ subsequent tokens provided to the model, the easier is for the model to guess the n^{th} following token. Increasing the number of tokens fed into the model does also (i) increases the possible noise provided to it, and (ii) reduces to "locality" of the n-1 fed tokens (*i.e.,* increases the likelihood of having tokens belonging to different statements). Let us discuss this using the following example statements:

```
import org.program-comprehension.*;
import java.io.*;
public static void main ...
```

Considering our experimental design, when using n=3, we start reading the first two tokens (`import org`) and ask the language model to recommend the third one (`.`). Then, we feed (`org.`) and again ask the language model to suggest the third token (`program-comprehension`). This process is continued until we reach the last token in the file. As we can see, for low values of n we provide very *localized* sequences of $n-1$ tokens to predict the n^{th} one. In other words, the $n-1$ tokens are generally part of the same code statement and narrow down the possible tokens that can follow them.

When $n = 15$, in the example above we provide as input to the language model the whole first two import statements (for a total of 14 tokens), asking the language model to predict the 15^{th} token (*i.e.,* `public`). In this case it is challenging for the language model to guess the correct token, due to the poor localization of the fed information (the 14 input tokens belong to statements unrelated with the one in which we ask the language model to support the auto completion).

Our results support the findings by Bruch *et al.* [19] and Nguyen *et al.* [20], and highlight the importance of exploiting contextual information when supporting code completion. The accuracy obtained when considering the top t recommendations with t going from 2 to 10 is available in our replication package [11] and is consistent with the same observations.

1) RQ4: How effective is the language model in supporting code completion for different code constructs?: Fig. 7 shows the accuracy of the n-gram language model (for different values of n) when used to support code completion on different code constructs (again, when the top recommendation is considered, other results in [11]). Overall, the trend is the same previously observed for the whole project: The model performs better for small n values.

What heavily varies is the performance of the language model when applied to the different types of code constructs. To zoom into this analysis, Fig. 8 and 9 compare the performance of the 3-gram language model on the different constructs via boxplots and statistical tests, respectively. Hereafter, we discuss the results for $n = 3$ as in the original paper by Hindle *et al.* [7], while results for other values of n are available in the replication package [11]. The achieved results highlight that:

- **The performance of the language model varies a lot across different types of code constructs.** This is clear both in the boxplots (Fig. 8) as well as from the results of the statistical analysis (Fig. 9), in which several significant differences accompanied by a medium/large effect size are observed. For example, the performance of the language model is very good when supporting code completion for `import` and `package` statements (median=0.76), while it strongly drops when working on `while` conditions (median=0.36).
- **There is a clear correlation between the redundancy rate of code constructs, and the performance of the language model when applied on them.** While this is evident when putting together the results of our two studies, we also compute the correlation between the redundancy rate of code constructs and the accuracy of the language model in predicting tokens belonging to them by using the Spearman rank correlation analysis [21]. We obtained a correlation coefficient of $\rho = 0.71$, highlighting a *strong* correlation on the basis of the guidelines provided by Cohen [21].

We further dig into the results by looking for the code tokens correctly predicted by the language model when setting $n = 3$ on the same 30 randomly selected systems used in Study I. We discuss in detail the results for Microsoft Thrifty, on which 13,176 out of 29,763 tokens have been correctly predicted (44% accuracy).

The top ten correctly predicted tokens on this system are: "`.`", "`(`", "`)`", "`;`", "}", "{", "`public`", "`thrift`", "`apache`", "=", accounting for a total of 9,494 (72%) of the correctly predicted tokens. Only two of the top-10 correctly predicted tokens are project specific (*i.e.,* "`thrift`" and "`apache`") and they are correctly predicted since, as shown in Study I, frequently used in `catch` statements. In addition, 1,073 correctly predicted tokens (a further 8%) is represented by Java keywords. The results obtained from other 29 systems are in line with these findings and confirm that most of the correctly predicted tokens are not project specific.

We believe that these findings are of paramount importance when considering the use of language models for supporting code completion. Indeed, while the performance of the language model could be overall acceptable (*e.g.,* 44% accuracy on Microsoft Thrifty), it is mostly effective in recommending tokens (i) belonging to very specific parts of the code (*e.g.,* `import` and `catch` statements), and (ii) mainly representing syntactic sugar of the programming language.

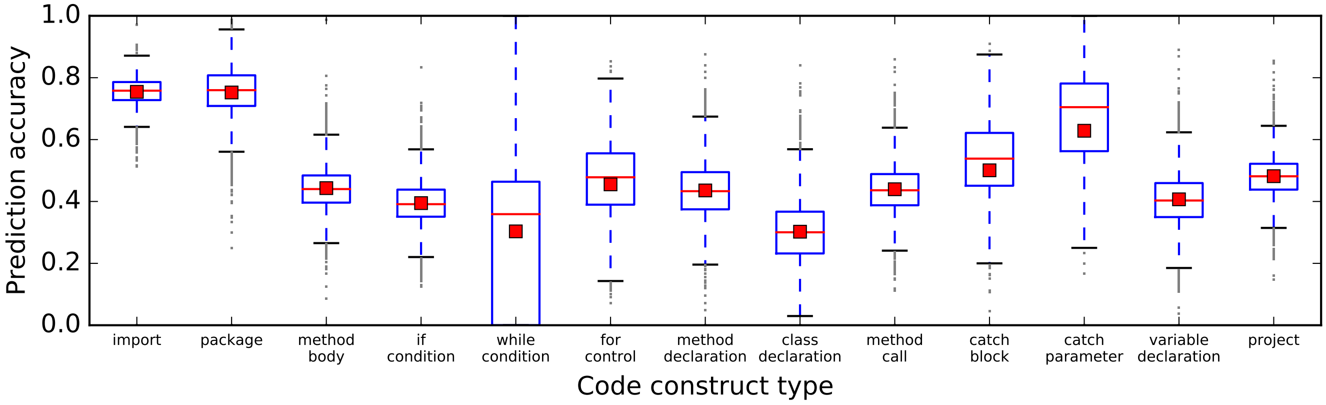


Fig. 8: Accuracy of the 3-gram model when supporting code completion on different constructs (top-1 recommendation)

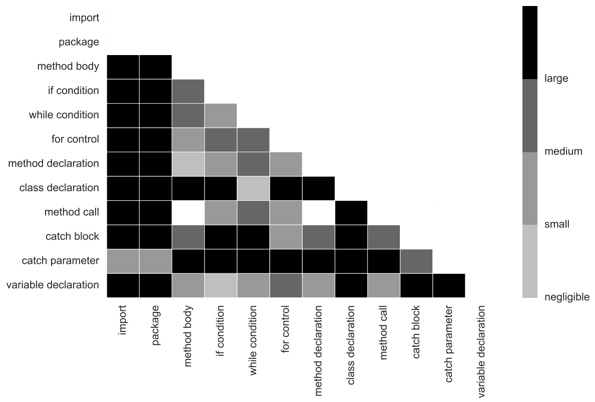


Fig. 9: Statistical comparisons for the accuracy of the 3-gram language model on different code constructs

Our results indicate that language models alone cannot effectively support code completion and that, as proposed by Hindle *et al.* [7], they can only *complement* recommendations generated by other techniques. Also, our findings clearly highlight the strong impact that unevenly redundancy rates of code constructs can have on applications assuming the high redundancy of source code.

V. Threats to Validity

Threats to *construct validity* concern the relation between theory and observation. In this work they are mainly due to the measurements we performed.

In Study II, we assess the performance of the language model presented by Hindle *et al.* [7] in supporting code completion by using their same experimental design. In particular, given a project P_i, we train the language model on a set of P_i's files and test it on all the n-grams (as explained, we experimented with different values of n) extracted from files belonging to the test set. As done in [7], each file in the test set was scanned from the beginning to the end to extract all its n-grams on which the language model was then evaluated.

Such an approach aims at simulating the code writing by the developer: She writes the first $n-1$ tokens, and uses the code completion to recommend the n^{th} token, then she writes other $n-1$ tokens, and again uses code completion to suggest the next token, etc. Clearly, developers do not write code by following such a linear approach from the beginning to the end, and we acknowledge such a threat.

Note also that the performance we report for the language model cannot be directly compared with the one reported in the original paper by Hindle *et al.* [7]. Indeed, while we report the raw accuracy of the language model when used to support code completion, in [7] the authors show the gain in terms of accuracy obtained over the Eclipse built-in code completion module. Since our primary goal was to show how the different redundancy rates of code constructs impact the performance of techniques exploiting such a redundancy, we preferred to report the language model accuracy by itself.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. In Study I, when assessing software redundancy, we did not experiment with all possible sequence lengths, but we limited our analysis to sequences going from 3 to 60 tokens at steps of 3. Still, the trend observed in the achieved result is quite clear, and shows that, as expected, the redundancy rate decreases with the increase of the sequence length (see Fig. 2). We do not expect to observe anything different by further increasing the sequence length.

In both our studies, we did not consider all possible code constructs that can be extracted from Java systems. However, the number and diversity of the considered constructs have been sufficient to observe differences in the redundancy rate and in the accuracy of the language model.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalizability of our findings. While our two studies have been performed on a large code base including 2,640 projects, we are aware that (i) all subject projects are written in Java, thus calling for the need of analyzing software projects written in other programming languages, and (ii) we limited our analysis to open source projects ignoring industrial systems.

VI. Related Work

A. Code Redundancy

Code clones, *i.e.,* code fragments similar to other ones by some given definition of similarity [22], are a common form of code redundancy, and have been widely studied. We limit our discussion to few of the works focusing on clones. Baker [23] inspected two systems and tried to find maximal sections of code over a certain length which are exactly the same or only differ in parameter names. Their results indicate that around 20% of the code is duplicated or near-duplicated. Roy and Cordy [24] examined 15 Java and C systems, and reported that $\sim$15% of the Java methods and 2.5% of the C functions are exact clones. Kapser and Godfrey [25] conducted two case studies and reported that 50% of the clones were related to function clones. With another case study on the Apache web server, they later showed the existence of "cloning hotspots": 17% of the code contained 38% of the clones [26].

Mockus [27] analyzed 13.2 million source code files from open source projects, and reported that over 50% of files were reused across projects. While our work is naturally related to code clones, we focus on the code redundancy phenomenon at a lower granularity level, with the goal of investigating (i) how it varies in code constructs, and (ii) how this variations impacts the performance of techniques leveraging code redundancy.

Other studies have explored code redundancy from a different perspective. Barr *et al.* [28] found that 42% of the code changes can be largely reconstituted from existing code. Nguyen *et al.* [29] reported that 12.1% of the routines (*i.e.,* a portion of code that performs a specific task, such as methods) are repeated between 2 and 7 times in projects. Finally, the study by Gabel and Su [8] is certainly the most related to our work. Indeed, Study I represents a *differentiated replication* of the investigation presented in [8], featuring a different and larger code base and investigating at a fine-grained level how code redundancy changes across code constructs.

B. Code Completion

Code completion is one of the killer features of modern IDEs, and researchers have proposed different methods to improve code completion accuracy. Again, due to the lack of space we focus our discussion on a few representative works.

By mining existing code, Bruch *et al.* [19] (i) filter out candidates from the list of tokens recommended by the IDE that are not relevant to the current working context and (ii) rank candidates based on how relevant to the context they are. These features help in substantially improve the standard IDE code completion engine.

Also Nguyen *et al.* [20] exploited context-sensitive information in their `GraPacc` approach, showing its effectiveness in supporting code completion. `GraPacc` models API usage patterns by relying on a graph representation, where nodes represent actions (*e.g.,* method calls) and controls (*e.g.,* while) points, and edges represent control and data flow dependencies between nodes. Context information such as the relation between API elements and other code elements is considered for ranking most fitted API usage patterns.

Raychev *et al.* [30] extracted sequences of method calls from a large codebase and trained a language model on them. They applied this model to support the autocompletion of method calls, achieving an accuracy of 90% when considering the top three results. We experimented in Study II the previously discussed language model proposed by Hindle *et al.* [7], showing that its performance substantially varies when applied to constructs characterized by different redundancy.

The language model proposed by Hindle *et al.* was improved by Nguyen *et al.* [31], [29] and by Tu *et al.* [9]. Nguyen *et al.* [31] presented a statistical semantic language model for source code extending the standard language model by annotating each token with its type and semantic role. Also, they exploit a more advanced n-gram topic model to support code completion. In a related work of the authors, they also proposed the use of an AST-based language model instead of the n-gram language model to recommend the next valid syntactic template and detect common syntactic templates [29].

Tu *et al.* [9] enriched the language model with a cache exploiting a specific *localness* of software, *i.e.,* repetitions of a specific n-gram localized in few files. Other approaches [9], [31], [29] achieve improvements over the language model proposed by Hindle *et al.* [7].

In our study we chose to adopt the simplest approach (*i.e.,* Hindle *et al.* [7]) since our goal was not to experiment with the best code completion tool available, but to show that approaches leveraging code redundancy should consider its strong locality in specific code constructs.

VII. Conclusion

We examined the redundancy of code constructs, and investigated the impact of its inequality on an application leveraging code redundancy, namely n-gram based code completion.

Our results indicate that while software is quite redundant when considered as a whole, the redundancy is localized in specific code constructs. Such a characteristic of code redundancy strongly impacts the performance of application exploiting code redundancy, like n-gram based completion.

Our future work will focus on the definition of smarter code completion tools leveraging our findings, and on customizing the use of the language model on the basis of the specific code constructs on which it is applied.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "HI-SEA" (SNF Project No. 146734).

References

[1] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of ICSM 1999 (15th IEEE International Conference on Software Maintenance)*. IEEE, 1999, pp. 109–118.

[2] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proceeding of ISESE 2004 (2004 International Symposium on Empirical Software Engineering)*. IEEE, 2004, pp. 83–92.

[3] C. Kapser and M. W. Godfrey, "'cloning considered harmful' considered harmful," in *Proceeding of WCRE 2006 (13th Working Conference on Reverse Engineering)*. IEEE, 2006, pp. 19–28.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *Proceeding of WCRE 2000 (7th Working Conference on Reverse Engineering)*. IEEE, 2000, pp. 98–107.

[6] S. Burrows, S. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software-Practice and Experience*, vol. 37, no. 2, pp. 151–176, 2007.

[7] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of ICSE 2012 (34th International Conference on Software Engineeering)*, 2012, pp. 837–847.

[8] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of FSE 2010 (18th SIGSOFT International Symposium on Foundations of Software Engineering)*, 2010, pp. 147–156.

[9] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of FSE 2014 (22nd International Symposium on Foundations of Software Engineering)*, 2014, pp. 269–280.

[10] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *Proceedings of ICSE 2016 (38th International Conference on Software Engineeering)*. ACM, 2016, pp. 428–439.

[11] B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza, "Replication package." https://icpc-redundancy.github.io/icpc-2017.zip.

[12] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of FSE 2014 (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering)*. ACM, 2014, pp. 472–483.

[13] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[14] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *Proceedings of ICSM 2013 (29th IEEE International Conference on Software Maintenance)*. IEEE, 2013, pp. 516–519.

[15] W. J. Conover, "Practical nonparametric statistics," 1999.

[16] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[17] R. J. Grissom and J. J. Kim, "Effect sizes for research: A broad practical approach," *Mahwah, NJ: Earlbaum*, 2005.

[18] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of ACL 1996 (34th annual meeting on Association for Computational Linguistics)*. Association for Computational Linguistics, 1996, pp. 310–318.

[19] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of ESEC/FSE 2009 (7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering)*. ACM, 2009, pp. 213–222.

[20] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *Proceedings of ICSE 2012 (34th International Conference on Software Engineeering)*, 2012, pp. 69–79.

[21] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.

[22] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[23] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceeding of WCRE 1995 (2nd Working Conference on Reverse Engineering)*. IEEE, 1995, pp. 86–95.

[24] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *Proceeding of WCRE 2008 (15th Working Conference on Reverse Engineering)*. IEEE, 2008, pp. 81–90.

[25] C. Kapser and M. W. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings of IWPSE 2004 (7th International Workshop on Principles of Software Evolution)*, pp. 85–94.

[26] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.

[27] A. Mockus, "Large-scale code reuse in open source software," in *Proceedings of FLOSS 2007 (1st International Workshop on Emerging Trends in FLOSS Research and Development)*. IEEE, 2007, pp. 7–7.

[28] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of FSE 2014 (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering)*. ACM, 2014, pp. 306–317.

[29] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *Proceedings of MSR 2016 (13th International Workshop on Mining Software Repositories)*. ACM, 2016, pp. 362–373.

[30] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of PLDI 2014 (35th ACM SIGPLAN Conference on Programming Language Design and Implementation)*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[31] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of ESEC/FSE 2013 (9th Joint Meeting on Foundations of Software Engineering)*, 2013, pp. 532–542.