University of Molise

Department of Biosciences and Territory

S.S.D.: INF/01

CROWDSOURCED DOCUMENTATION AND CODE SNIPPETS

SALVATORE GEREMIA

Ph.D. Program Coordinator
Prof. Giovanni Fabbrocino

Supervisor
Prof. Massimiliano Di Penta

April 2020

This research was done under the supervision of Prof. Massimiliano Di Penta, with the financial support of the University of Molise, from November 1st, 2016 to October 31th, 2019.

The final version of the thesis has been revised by Prof. Lerina Aversano and Prof. Annibale Panichella.

The doctoral committee was composed by:

Prof. Andrea Di Sorbo	University of Sannio, Italy
Prof. Carmine Gravino	University of Salerno, Italy
Prof. Rocco Oliveto	University of Molise, Italy

First release, January 2020. Revised in April 2020.

Salvatore Geremia: *Crowdsourced Documentation and Code Snippets.*
A thesis submitted for the degree of Doctor of Philosophy.

ABSTRACT

The desire of those who make software, as well as of those who use it, is that it should be of quality. The software quality depends on a multitude of factors including functionality, reliability, efficiency, and maintainability, which somehow reflect the actions taken by the one who implements the software: the developer. For example, if a developer chooses to use a code snippet developed by someone else or if he leverages methods provided by an external library, he should ask himself how this choice could affect the overall quality of the software. For the same reason, a developer involved in software maintenance should be very confident about the code understandability, he will be working on.

In this thesis we investigate the characteristics of Stack Overflow answers in order to identify what is the rational that leads a developer to choose a snippet to integrate into his software project. To this end, we analyzed and compared the characteristics of the documentation and code snippets leveraged and non-leveraged by developers on GitHub. The analysis showed the factors that best discriminate between leveraged and non-leveraged answers are: the score, the number of comments and the posting date of the answer. A specific analysis on documentation quality was carried out by examining a set of Application Program Interfaces (API). The objective was to investigate the relationship between the first use of a low-quality documentation API and the introduction of bugs in the code. This hypothesis was confirmed by the results. A further analyzed point concerns the understanding level perceived by the developer when dealing with code snippets written by others. Specifically, we have tried to understand whether naturalness calculated through a model focused on the developer's knowledge was a predictor for the developer's understanding of code. The results showed code snippets that are familiar, and so more natural for the developer, tend to mislead him from the understanding of the code.

ABSTRACT

Il desiderio di chi realizza un software, così come quello di chi lo utilizza, è che questo sia di qualità. La qualità del software dipende da una miriade di fattori tra cui la funzionalità, l'affidabilità, l'efficienza e la manutenibilità, caratteristiche che in qualche modo riflettono le azioni compiute da chi è responsabile della realizzazione del software: lo sviluppatore. Ad esempio, nel caso in cui uno sviluppatore volesse includere all'interno del software un frammento di codice (*code snippet*) implementato da terzi, oppure utilizzare le funzioni fornite da una libreria esterna, dovrebbe domandarsi in che misura questa scelta possa incidere sulla qualità complessiva del software. Per la stessa ragione, uno sviluppatore incaricato di fare manutenzione su un software dovrebbe essere certo di aver compreso il codice su cui andrà ad operare.

In questa tesi si indagano le caratteristiche delle risposte ai quesiti postati su Stack Overflow al fine di individuare il razionale che spinge uno sviluppatore a scegliere lo snippet da inserire all'interno del proprio progetto software. A questo scopo, sono state analizzate e comparate le caratteristiche della documentazione e dei code snippet utilizzati (*leveraged*) e non utilizzati (*non-leveraged*) in progetti GitHub. Dall'analisi è emerso che i fattori che riescono meglio a discriminare le risposte leveraged da quelle non-leveraged sono: il punteggio, il numero di commenti e la data di pubblicazione della risposta. Un'analisi specifica sulla qualità della documentazione e delle sue componenti informative è stata effettuata esaminando un set di Application Program Interface (API). L'obiettivo era quello di investigare la relazione tra il primo utilizzo di API aventi una documentazione di bassa qualità e l'introduzione di errori (*bug*) nel codice, ipotesi successivamente confermata dai risultati ottenuti. Un'ulteriore aspetto analizzato riguarda il livello di comprensione percepito dallo sviluppatore alle prese con frammenti di codice scritto da altri. Nello specifico, si è cercato di capire se la naturalezza calcolata attraverso un modello incentrato sulla conoscenza dello sviluppatore fosse un predittore per la comprensibilità del codice da parte di quest'ultimo. I risultati hanno mostrato come porzioni di codice che presentano caratteristiche familiari, e per questa ragione percepite più naturali, tendano a trarre in inganno lo sviluppatore, fuorviandolo dalla comprensione del codice.

PUBLICATIONS

1. Simone Scalabrino, **Salvatore Geremia**, Remo Pareschi, Marcello Bogetti, and Rocco Oliveto, (2018). Freelancing in the economy 4.0: How information technology can (really) help. In *Social Media for Knowledge Management Applications in Modern Organizations* (pp. 290-314). IGI Global.

2. **Salvatore Geremia**, and Damian A. Tamburri, (2018). Varying defect prediction approaches during project evolution: A preliminary investigation. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)* (pp. 1-6). IEEE.

3. **Salvatore Geremia**, Gabriele Bavota, Rocco Oliveto, Michele Lanza, and Massimiliano Di Penta, (2019). Characterizing Leveraged Stack Overflow Posts. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (pp. 141-151). IEEE.

CONTENTS

1 INTRODUCTION 1
1.1 Origin of chapters and thesis contributions 3
2 CHARACTERIZING LEVERAGED STACK OVERFLOW POSTS 5
2.1 Introduction 5
2.2 Study Design 7
2.2.1 Data Collection 9
2.2.2 Analysis Methodology 13
2.2.3 Replication Package 16
2.3 Results 16
2.3.1 To what extent are code snippets from "non-leveraged" answers used in GitHub projects? . 16
2.3.2 Which are the characteristics of SO answers that have been leveraged by developers? 16
2.3.3 Which is the performance of a recommender system in identifying posts that are likely to be leveraged by developers? 19
2.4 Threats to Validity 24
2.5 Related work 25
2.5.1 Reusing Code From the Internet 25
2.5.2 Prediction Tasks on Stack Overflow 26
2.5.3 Stack Overflow in Recommender Systems . . . 27
2.6 Conclusion 28
3 A DEVELOPER-CENTRIC NATURALNESS MODEL FOR PREDICTING CODE UNDERSTANDABILITY 29
3.1 Introduction 29
3.2 Background and Related Work 32
3.2.1 Measuring Code Understandability 32
3.2.2 Naturalness of Code 33
3.3 Motivating study 34
3.3.1 Research Question and Design 34
3.3.2 Analysis of the results 38
3.3.3 Take Away 38
3.4 Empirical Study Design 38
3.4.1 Study Context and Data Collection 39
3.4.2 Data Analysis 43
3.5 Empirical Study Results 45
3.5.1 Discussion and Implications 49
3.6 Threats to Validity 51
3.7 Conclusion 52
4 ON THE RELATIONSHIP BETWEEN API QUALITY AND THE SOFTWARE FAILURE PRONENESS 55
4.1 Introduction 55

4.2 Empirical Study Design 56
4.2.1 Research Questions 56
4.2.2 Analysis Methodology 57
4.3 Results 59
4.3.1 What are the elements composing the documentation of Java APIs? 59
4.3.2 What is the quality of API documentation, and of its elements? 62
4.3.3 Does a low API documentation quality relate with higher defect-proneness? 63
4.4 Threats to Validity 67
4.5 Related work 67
4.6 Conclusion 68
5 CONCLUSION 69

BIBLIOGRAPHY 71

LIST OF FIGURES

Figure 1 Feature correlation dendrogram. 14
Figure 2 MCC achieved by the Random Forest. 22
Figure 3 Precision_1 achieved by the Random Forest. . . 23
Figure 4 Example of JDBC snippet. 30
Figure 5 Empirical investigation methodology: preliminary study and study on developer-centric naturalness. 32
Figure 6 Example of deceptive code. 50
Figure 7 Taxonomy of knowledge types. 60
Figure 8 Normal Q-Q Plots of overall API quality and average API elements quality. 65
Figure 9 Boxplot of API quality. 66

LIST OF TABLES

Table 1 Factors considered in our study. 8
Table 2 RQ_1: Effect size for statistically significant differences. 17
Table 3 RQ_1: Paired analysis of features' distribution on 104 discussions having both leveraged and non-leveraged answers. 20
Table 4 Prediction accuracy of the experimented classifiers in different configurations. 21
Table 5 MDI of features used to predict post reference. 24
Table 6 Results of the Kendall's correlation between understandability and naturalness. 38
Table 7 Number of methods for investigated topics. . . 41
Table 8 Effect of global and developer-centric naturalness, of topic knowledge, and of their interaction using two-way permutation test. 46
Table 9 Actual understandability and deceptiveness for global and developer-centric naturalness. . . . 47
Table 10 Comparison on the subset with Wilcoxon rank-sum test for developer-centric naturalness. . . 48
Table 11 Number of analyzed methods (#M) and total number of methods (#T) for Maven categories. 57
Table 12 Number of API documentation (#N) for different overall documentation rates (API rate). . . 63
Table 13 Documentation Element Types with number of occurrences (#N) and quality statistics. 64

ACRONYMS

ABU	Actual Binary Understandability
API	Application Programming Interface
AU	Actual Understandability
AUC	Area Under the Receiver Operating Characteristics curve
BC	Binary Deceptiveness
CD	Continuous Deceptiveness
CWCM	Cognitive Weight Complexity Measure
LOC	Lines Of Code
LM	Language Model
FN	False Negative
FP	False Positive
GUI	Graphical User Interface
JDBC	Java DataBase Connectivity
MCC	Matthews Correlation Coefficient
MDI	Mean Decrease Impurity
PBU	Perceived Binary Understandability
SO	Stack Overflow
TAU	Timed Actual Understandability
TNPU	Time Needed for Perceived Understandability
TN	True Negative
TP	True Positive

1 INTRODUCTION

Since 1957, when the first high-level programming language (FORTRAN) was released, the world of software programming has witnessed the birth of over 8500 programming languages. For this reason it is impossible to estimate the amount of code produced by developers over the last 60 years, just consider that only the Internet services offered by Google exceed 2 billion lines of code[1]. Such an amount of existing code has had a significant influence on the way software is developed and has increasingly encouraged the reuse of existing code posted on public repositories (*e.g.*, GitHub) or on question and answer websites (*e.g.*, StackOverflow).

> *The goal of this thesis is to study the quality of crowdsourced documentation and code snippets, with the aim of determining its impact on software quality, and on development practice.*

One of the most popular question and answer websites where you can discuss programming is, undoubtedly, Stack Overflow. On average, a new answer is posted every five seconds. Because of the amount of data, the main challenge for developers is choosing code snippets to use within their code. However, and not surprisingly, not every Stack Overflow post is useful from a developer's perspective. In this thesis, and particularly in Chapter 2, we investigate the characteristics of code snippets and their documentation in order to identify the reasons why a developer chooses to use that code within their GitHub project. We refer to these posts as *leveraged* posts. We study the characteristics of *leveraged* posts as opposed to the *non-leveraged* ones, focusing on community aspects (*e.g.*, the reputation of the user who authored the post), the quality of the included code snippets (*e.g.*, complexity), and the quality of the post's textual content (*e.g.*, readability). Then, we use these features to build a prediction model to automatically identify posts that are likely to be *leveraged* by developers. Results of the study indicate that post meta-data (*e.g.*, the number of comments received by the answer) is particularly useful to predict whether it has been leveraged or not, whereas code readability appears to be less useful. A classifier can classify leveraged posts with a precision of 65% and recall of 49% and non-leveraged ones with a precision of 95% and recall of 97%. This opens the road towards an automatic identification of "high-quality content" in Stack Overflow.

1 https://informationisbeautiful.net/visualizations/million-lines-of-code/

Another issue related to the use (*e.g.*, maintenance activities) of a code snippet made by an external developer involves the understandability of the code. A software developer who recognizes within a snippet code a pattern seen or used in the past may perceive that code as familiar (*natural*) and, consequently, understand it more easily. This feature, if verified, would pave the way for the capability to predict a developer's code. In Chapter 3 we investigate the relationship between *naturalness* of code snippets and the degree of understandability achieved by the developers. In support of this assumption we implemented a developer-centric model for naturalness and we tested it on 52 software developers with different experiences and knowledge. Our results show that (i) generic naturalness models are not useful for predicting understandability, and (ii) the lower the developer-centric naturalness of a code artifact, the higher the probability that developers get deceived, *i.e.*, they believe they understood the code while they did not. Our results open the road towards the application of the developer-centric naturalness in further studies and for building recommender systems.

In addition to code examples, the web is full of public APIs (Application Program Interface) that developers can integrate into their software projects. The use of an API, although it simplifies the development process by considerably shortening its time, has some pitfalls. The main one is the understanding of the API by the developer, in order to be able to use it correctly, so the API documentation plays a key role. A superficial, outdated, or generally low quality documentation may negatively affect the use of the API, leading the developer to make more errors and consequently increases the possibility of introducing bugs. It is easy to assume that as the developer gets to know the API by using it, the number of errors made decreases. For this reason, Chapter 4 examines 800 documentation methods extracted from Java APIs and used within GitHub projects, assesses their quality and investigates the correlation between the quality of the documentation and the introduction of bugs by the developer. In particular, the analysis focuses on bugs introduced at the time of the first use of a given API. Six researchers were involved in the analysis of the documentation of the selected methods. Their task was to examine the content of the documentation, identifying the types of knowledge present, assigning to the various paragraphs a quality assessment and indicating an overall assessment of the entire documentation. At the end of the manual analysis phase we exploited the SZZ algorithm [68] to investigate the correlation between the introduction of errors and the documentation quality of the API methods used by the developer.

1.1 ORIGIN OF CHAPTERS AND THESIS CONTRIBUTIONS

Although the research works illustrated in the thesis and organized in different chapters are linked to a single research stream, each chapter is self-contained. The choice to organize the content of the thesis in this way derives from a twofold motivation, the first concerns the possibility of reading the individual chapters making it easier to fully understand them, the second is to reflect the exact correspondence between the chapters presented and the works published, under review, and in preparation for submission.

Chapter 2 was published in the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM) in 2019 (see *Publications*).

Chapter 3 is currently under review for the Journal of Systems and Software (JSS). The research work required the collaboration of two PhD students, Valentina Piantadosi and myself, and four professors, Simone Scalabrino, Rocco Oliveto, Gabriele Bavota and Massimiliano Di Penta, involving three universities: University of Molise, University of Sannio and Università della Svizzera italiana. More specifically, my main contributions were (i) to build the Developer-Centric Naturalness Models (see Section 3.4), (ii) to mine the methods from a set of Java projects on GitHub, (iii) to develop the tool to calculate the use of specific topics within the methods, (iv) to parameterize the use of training data for the construction of the naturalness models. I also contributed to the selection and validation of the questions used for the survey and the analysis of the results.

Chapter 4 is in preparation for submission. This research work was carried out during the six months spent as visiting student at the Università della Svizzera italiana. The chapter presents only a part of the entire carried out research work, the one in which I am the first author.

2

CHARACTERIZING LEVERAGED STACK OVERFLOW POSTS

2.1 INTRODUCTION

Software developers often need to acquire new pieces of knowledge to deal with the ever-increasing complexity of modern software systems. Thus, developers are engaged with a continuous information-seeking process performed by interacting with teammates, by reading different forms of documentation, and by consulting online resources such as question and answer (Q&A) websites.

Q&A websites have become a prominent point of reference for software developers [74], also due to the vast availability of the information they provide. Stack Overflow (SO), the most popular Q&A website related to computer programming, counts, at the date of writing, over 16 Million questions and 25 Million answers[1]. Such a vast amount of data represents a precious source of information that can be automatically mined to provide support for software engineering tasks.

Indeed, many researchers proposed recommender systems built on top of the information mined from SO. These include, for example, techniques recommending SO discussions relevant for a given task at hand [17, 56], providing support for the automatic documentation of source code [3, 72, 80], or recommendation of code elements [10, 72, 86].

One of the main challenges for these recommender systems is the identification of SO posts of *high-quality* and that could be useful for developers. Researchers have dealt with this challenge by integrating heuristics aimed at discarding low-quality content. For example, AutoComment [80], a tool using information from SO posts to automatically document source code, only uses information items from posts that have been positively judged by the SO community. Although heuristics can be in many cases appropriate to provide accurate recommendations, they do not consider the extent to which the recommended posts reflect properties of posts that, in the past, have been considered useful by developers. Thus, the following questions remain unanswered:

> *What is a useful SO post? Is it an answer that received many up-votes, or coming from a well-reputed user? Do the properties of snippets contained in the post matter?*

1 https://stackoverflow.blog/2018/09/27/stack-overflow-is-10/

Given that many SO answers have inspired solutions in open-source projects [6, 85], we conjecture that answers useful in the past for somebody are likely to be useful in the future. Thus, we empirically investigate which are the characteristics of SO answers that have been previously "leveraged" by developers, and compare them with that of "non-leveraged" answers. We assume an answer to have been leveraged if it has been mentioned (*i.e.*, linked) at least once in the source code of an open source project hosted on GitHub. Such a link might indicate the willingness of a developer to (i) indicate the reuse of the answer's code snippets, (ii) document the rationale of an implementation choice, or (iii) simply refer to the answer as an interesting source of information. **Note that, while for a linked answer it is safe to assume that it has been leveraged, a non-linked answer might have been leveraged without being explicitly linked. For example, the code snippet contained in the answer might have been copied without putting a proper attribution in the source code about the snippet's provenance [5]. We discuss how this issue impacts our findings in Section 2.3.1.**

We use data from SOTorrent [7], an open database containing the official SO data dump and including references from GitHub files to SO posts. We analyze the last available release (2018-02-16) containing 42 Million SO posts and 6 Million post references from GitHub. We filter out from this set the subset of data referring to SO answers including at least one Java code snippets. This was done to (i) have a homogeneous set of posts for which it is possible to compute community aspects (*e.g.*, the reputation of the user who authored the post), quality of the included code snippets (*e.g.*, complexity), and quality of the post's textual content (*e.g.*, readability); (ii) check whether the snippets of the answers classified as non-leveraged (*i.e.*, not explicitly linked) have been reused in GitHub projects without a proper attribution, thus threatening our classification of the answer as "non-leveraged" (details in Section 2.2). Through this filtering, we obtained 19,342 posts, 3,214 of which *leveraged* at least once in projects hosted in GitHub, and 16,128 that were not leveraged.

Using this dataset, we first statistically compare the distribution of 22 factors related to community aspects, quality of the code snippets, and quality of the posts' text, between *leveraged* and *non-leveraged* posts. Then, we use such factors as independent variables to build classifiers able to predict whether a post is likely to be leveraged or not (dependent variable). We experiment with four different classifiers (*i.e.*, Bayesian Network, J48, Logistic Regression, and Random Forest) in several configurations.

Results show that the Random Forest is the classifier obtaining the best classification accuracy, with AUC=0.856 and Matthews Correlation Coefficient (MCC)=0.528. More specifically, the classifier achieves a precision of 65% and recall of 49% for *leveraged* posts, and a preci-

sion of 95% and recall of 97% for *non-leveraged* posts. While our model is still far from providing a highly precise classification, the achieved results pave the way to more research on the automatic identification of "high-quality content" in SO.

2.2 STUDY DESIGN

The *goal* of this study is to investigate the characteristics of SO answers that have been previously *leveraged* by developers, as opposed to other answers (*non-leveraged*) for which there is no evidence of their usefulness. The *perspective* is of researchers interested to understand what makes a good SO answer, and possibly to exploit this information to better recommend SO posts. The study *context* consists of 19,342 answers posted on SO (3,214 leveraged and 16,128 non-leveraged), all related to the Java programming language, and always containing a source code snippet.

In our study we assume a SO answer to have been leveraged if it has been linked at least once in the code of a GitHub project. However, a non-linked answer might have been leveraged without being explicitly linked [5]. For this reason, we formulate the following preliminary research question:

> **RQ_0:** *To what extent are code snippets from "non-leveraged" answers used in GitHub projects?*

We answer RQ_0 by selecting a sample of SO answers classified as *non-leveraged* in our dataset (through the process detailed later on in the study design), with the goal of verifying how many of their code snippets have been reused in Java GitHub projects without an explicit reference. This will give an idea of how reliable is our classification of SO answers as leveraged and non-leveraged. As shown in Section 2.3.1, only a small percentage of *non-leveraged* is misclassified, meaning that its code snippet has been used in a GitHub project without a proper reference.

After this preliminary analysis, we investigate whether the value distributions for various kinds of answer's features, characterizing both its content and its author, change between *leveraged* and *non-leveraged* answers. Therefore, we ask our first research question:

> **RQ_1:** *Which are the characteristics of SO answers that have been leveraged by developers?*

We focus on three kinds of properties that can be objectively measured in SO answers and, thus, we compared them in *leveraged* and *non-leveraged* answers: (i) community aspects including, the reputation of the user who posted the answer, whether the answer has been marked as the "accepted answer" by the user who asked the question, etc.; (ii) the quality of the code snippet included in the answer,

Table 1: Factors considered in our study.

Factor	Description
Community Factors	
Is Accepted	1 if the answer is accepted, 0 otherwise
Answer Score	Answer upvotes minus answer downvotes
Comment Count	The number of answer's comments
Creation Date	The date when the answer was created
User Reputation	A score summarizing the reputation of a user on SO
User Up Votes	The number of *upvotes* received by a user
User Down Votes	The number of *downvotes* received by a user
Code Quality Factors	
Snippet LOC	The lines of code of the answer's code snippet(s)
Snippet Complexity	The cyclomatic complexity [41] of answer's code snippet(s)
Snippet Readability	The readability of the answer's code snippet(s) [11]
Text Readability Factors	
# Words	The number of words in the answer text
# Sentences	The number of sentences in the answer text
# Characters	The number of characters in the answer text
# Syllables	The number of syllables in the answer text
# Complex Words	The number of words composed of at least 3 syllables
ARI [67]	$4.71 \cdot \frac{\#Characters}{\#Words} + 0.5 \cdot \frac{\#Words}{\#Sentences} - 21.43$
SMOG [40]	$1.043 \cdot \sqrt{\#ComplexWords \cdot \frac{30}{\#Sentences}} + 3.1291$
SMOG Index [40]	$\sqrt{\#ComplexWords \cdot \frac{30}{\#Sentences}} + 3$
Flesch Kincaid [34]	$0.39 \cdot \left(\frac{\#Words}{\#Sentences}\right) + 11.8 \cdot \left(\frac{\#Syllables}{\#Words}\right) - 15.59$
Flesch Reading Easy [23]	$206.835 - 1.015 \cdot \left(\frac{\#Words}{\#Sentences}\right) - 84.6 \cdot \left(\frac{\#Syllables}{\#Words}\right)$
Coleman Liau [15]	$\left(5.89 \cdot \left(\frac{\#Characters}{\#Words}\right)\right) - \left(30 \cdot \left(\frac{\#Sentences}{\#Words}\right)\right) - 15.8$
Gunning Fog [26]	$0.4 \cdot \left[\left(\frac{\#Words}{\#Sentences}\right) + 100 \cdot \left(\frac{\#ComplexWords}{\#Words}\right)\right]$

assessed using state-of-the-art quality metrics that can be measured on a (possibly incomplete) code snippet (*e. g.*, cyclomatic complexity, readability); (iii) the quality of the answer's textual content, mainly assessed through metrics capturing its readability. The complete list of features is described in detail below and reported in Table 1.

We use the knowledge acquired answering $\mathbf{RQ}_1$, and in particular we use the factors studied in $\mathbf{RQ}_1$ as independent variables to devise an approach based on a machine learning technique to predict whether a given SO answer will be leveraged or not (dependent variable). Therefore, we pose our second research question:

> $\mathbf{RQ}_2$: *Which is the performance of a recommender system in identifying posts that are likely to be leveraged by developers?*

In the context of $\mathbf{RQ}_2$, we also investigate how the prediction accuracy is influenced by (i) the choice of the machine learning algorithm, (ii) the balancing of the training set, and (iii) the amount and temporal recency of the posts used to build the training set. Finally, we

analyze the importance of each independent variable for prediction purposes.

2.2.1 *Data Collection*

As the first step to answer our research questions, we collected SO answers that have been *leveraged* or *non-leveraged*. We use SOTorrent, a dataset provided by Baltes *et al.* [7] and containing the official SO data dump "augmented" with information about links going from GitHub files to SO posts. Thanks to SOTorrent, it is possible to know which SO posts have been linked in open source projects hosted on GitHub. In our study, we decided to focus only on SO answers since we measure on these posts a number of characteristics that are only available for answers (*e. g.*, whether an answer is the one marked as *accepted* by the user who posted the question). Also, we decided to only consider answers containing at least one Java code snippet. Again, this is done since (i) among the post characteristics we study, we also consider code quality aspects of the included snippets, *i. e.*, complexity, readability, and size, and (ii) the code snippet will be used in Section 2.4 to verify whether the code from non-leveraged answers have been reused (without an explicit link to the answer), thus threatening our the classification of posts as non-leveraged.

We used Google BigQuery[2] to select from the table `Posts` all answers that are *leveraged* (*i. e.*, linked in at least one Java file) and that contain at least one code snippet:

```
SELECT * FROM [sotorrent-org:2018_12_09.Posts]
WHERE Id IN (
  SELECT PostId
  FROM [sotorrent-org:2018_12_09.PostReferenceGH]
  WHERE FileExt = ".java")
AND PostTypeId = 2
AND Body LIKE "\%<pre\%<code>\%</code>\%</pre>\%"
```

This query returned 3,437 *leveraged* answers. Then, we select from the table `Posts` the *non-leveraged* answers containing at least one Java code snippet:

```
SELECT * FROM [sotorrent-org:2018_12_09.Posts]
WHERE Id NOT IN (
  SELECT PostId
  FROM [sotorrent-org:2018_12_09.PostReferenceGH])
AND PostTypeId = 2
AND Body LIKE
  "\%<pre class=\"lang-java\%<code>\%</code>\%</pre>\%"
```

This query returned 18,592 *non-leveraged* answers.

2 https://cloud.google.com/bigquery/

One important difference must be noticed between the two used queries. In the query selecting the *non-leveraged* answers, we made sure that the included code snippet was manually set to be a Java snippet by the SO user posting the answer.

This can be seen from the `class="lang-java"` property used in the query. This property is optional, and can be set by the SO user to obtain a better formatting of the posted code snippet. By querying the SOTorrent dataset, we noticed that only a minority of answers, even among those reporting a Java snippet, used the `class="lang-java"`.

Since the number of *non-leveraged* answers is much greater than the number of the *leveraged* answers, we decided to use this filtering when collecting the *non-leveraged*, since, in any case, the number of returned answers was high enough to run our study (18,592). This was not the case for *leveraged* answers, with only a few dozens answers using the `class="lang-java"` property. For this reason, we applied a "weaker" filter to identify *leveraged* answers, simply ensuring that the answer was linked in at least one Java file. However, through manual analysis, we noticed that not all linked answers include a Java snippet. For example, in some cases developers linked in a Java file a Stack Overflow answer containing a snippet written in C, just to indicate that they reimplemented in Java the algorithm in the posted answer.

Since, as detailed later, our study infrastructure is tailored for Java, we applied the following cleaning process on the 3,437 *leveraged* answers. First, we considered as relevant for our study, all the *leveraged* answers satisfying at least one of three conditions:

1. It was posted in response to a question explicitly marked with the "java" tag;
2. It was posted in response to a question containing the word "java" in the title;
3. It contained a code snippet explicitly marked as a Java snippet, as previously discussed for the *non-leveraged* answers.

This process resulted in 1,904 *leveraged* answers classified as relevant for our study. Then, we excluded the 17 answers in which the included snippet was explicitly marked as non-Java (*e.g.*, `class="lang-php"`). Finally, the first author manually analyzed the remaining 1,516 answers (3,437 - 1,904 - 17 = 1,516), selecting for inclusion in our study those that included a Java snippet. In the end, we considered as valid 3,228 *leveraged* answers out of the 3,437 initially obtained by querying SOTorrent.

Finally, we excluded all answers posted after Jan, 1 2018 (note that SOTorrent reported, in the analyzed database, posts up to Dec 2018). This choice was dictated by the following observation: A very recent post could be *non-leveraged* simply because no one "had enough time"

to leverage it, and not due to its characteristics[3]. For this reason, we excluded almost one year of data from our study, to factor out, at least in part, the "time" confounding factor. This left us with 19,342 posts, of which 3,214 *leveraged* and 16,128 *non-leveraged*.

For each of the selected answers, we extracted the following set of characteristics, or factors, listed in Table 1:

- **Community Factors.** This category includes characteristics of the answers that are related to their presence on a Q&A website. We consider factors acting as proxies for the "quality" of the answer (*i. e.*, *Is Accepted* and *Answer Score*); factors assessing the reputation of the user who posted the answer (*i. e.*, *User Reputation*, *User Up Votes*, and *User Down Votes*); and factors representing metadata of the answer, meaning its *Creation Date* and the number of comments it received (*i. e.*, *Comment Count*).

- **Code Quality Factors.** We only consider in our study answers including at least one Java code snippet. For a given answer, we take the set of contained code snippets, and we concatenate them together as a single snippet. Then, we measure its size, in terms of Lines Of Code (*Snippet LOC* in Table 1), its complexity, assessed with McCabe Cyclomatic Complexity [41], and its readability, assessed with the metric proposed by Buse and Weimer [11]. This metric combines a set of low-level code features (*e. g.*, identifiers length, number of loops, etc.) and has been shown to be 80% effective in predicting developers' readability judgments. We used the implementation of such a metric provided by the original authors[4].

- **Text Readability Factors.** We use several state-of-the-art text metrics to assess the readability of the text contained in the answer. These metrics are built on top of some basic information about the text to evaluate, such as the number of words and sentences composing it, the number of used complex words (*i. e.*, those composed of at least three syllables), etc. Starting from this information, different text readability metrics have been proposed. For example, the SMOG readability formula [40] estimates the years of education a reader needs to understand the given text. All text readability metrics used in our study have been computed using an open-source API available on GitHub[5].

While the extracted data is sufficient to answer RQ_1 and RQ_2, the analysis performed does not contemplate the extent to which answers

3 Note that the choice of the cutting point was made at the time the analysis started, early January 2019.

4 http://tinyurl.com/kzw43n6

5 https://github.com/ipeirotis/ReadabilityMetrics

classified as non-leveraged have still been used, without proper reference, in GitHub projects, thus invalidating their classification as "non-leveraged". To deal with this issue, in RQ_0 we selected a sample of 500 answers classified as *non-leveraged* and verified whether their code snippets have been used in GitHub projects. Note that out of $\simeq$ 16k non-leveraged snippets, a sample of 500 ensures a confidence interval of $\pm$ 4.3% for a confidence level of 95%. Since searching in the whole GitHub is not doable in a reasonable amount of time, we applied the following process to reduce the "search space" (*i. e.*, the files on GitHub where to search for the snippets of interest). We used the StORMed [55] island parser to extract from each of the 500 snippets the data types they use. Then, we excluded the snippets only using primitive types (*e. g.*, `int`) and/or the data types already defined in Java (*e. g.*, `String`). The excluded snippets were replaced with others randomly selected, to meet our goal of sampling 500 snippets for this analysis. A manual analysis of all extracted types was performed to remove noise identified through the island parser. For example, we found snippets using `YourType` or `foo` as data type. Again, these were excluded and replaced with other snippets.

Once completed the collection of the 500 snippets with the related types, we used the GitHub search APIs[6] to search for source code files including, in their text, the name of at least one of the types used by the selected snippets. We only searched in Java files (`language : java`). Since the GitHub APIs only report a maximum of 1,000 results per request (in our case, 1,000 files containing a given type), for each type t we sent several requests searching for Java files containing t and having a size in bytes included in a specific range r. In particular, we started with $r = [0, \ldots, 2500]$ (*i. e.*, all files with a size between 0 and 2.5 kB), storing all found files in a given folder. Then, we increased r at steps of 2500 Bytes (*i. e.*, the second search looked for files containing t and having a size between 2.5 and 5 kB), until we reached the maximum size supported by the GitHub APIs, in which only files smaller than 384 kB are searchable. This process provided us with a total of 230k files containing at least one of the searched types and took approximately 20 days.

Finally, given S one of the 500 snippets, we used the Simian clone detector[7] to identify type-2 clones between S and the set of files downloaded from GitHub using the same type(s) present in S. We chose Simian due to the fact that it can easily be run on non-compilable code. We set five as the minimum number of lines to be matched in order to classify a snippet as cloned in a Java file.

6 https://developer.github.com/v3/search/
7 https://www.harukizaemon.com/simian/

2.2.2 *Analysis Methodology*

We answer $\mathbf{RQ}_0$ by reporting the percentage of analyzed *non-leveraged* answers, for which we found a clone of their code snippet in GitHub project (*i. e.*, the percentage of misclassified *non-leveraged* answers).

We address $\mathbf{RQ}_1$ by statistically comparing the value distributions of the factors described in Table 1 between *leveraged* and *non-leveraged* posts. Especially, we use the Wilcoxon Rank Sum Test [79] assuming a significance level $\alpha = 5\%$, and the Cliff's delta (d) effect size [25].

We split the data into ten buckets based on the creation date of the answers. In particular, the first bucket contains the 10% oldest answers, while the tenth groups the 10% newest ones. Then, we show how the differences vary when considering datasets having a different size and different recency. We start by only considering the first bucket (oldest 10%). Then, the first two (oldest 20%), and so on up to 90% at steps of 10%.

These datasets (*i. e.*, first 10%, first 20%, etc.) are also the training sets used in RQ_2 where we test the machine learning models on unseen data.

Thus, when performing statistics to address RQ_1, we do not consider the last bucket (most recent 10%). Since the analysis involves multiple comparisons, we adjust p-values using the Benjamini and Hochberg [9] procedure, which ranks p-values, keeping the largest p-value unaltered, multiplying the second largest by the number of p-values divided by the rank (*i. e.*, two), and treating the remaining ones similarly to the second.

One of the limits of our study is that there may be other factors, beyond those captured by the considered features, that determined whether or not a post has been leveraged. To mitigate this threat, we identified posts having both leveraged and non-leveraged answers (104 in total in our dataset). For such posts, we performed a paired analysis *within-post* of the feature distribution, this time using a paired test, *i. e.*, the Wilcoxon Rank Sign Test [79], as well as using the paired Cliff's delta effect size [25].

To answer $\mathbf{RQ}_2$, we build classifiers based on machine learning techniques, using the factors in Table 1 as independent variables (*i. e.*, features) and the categorical variable *leveraged*/*non-leveraged* as dependent variable. Before applying the machine learners, we identify groups of factors that correlate and, for each group, we only keep the one that better correlates with the dependent variable. In detail, we use the *R varclus* function of the *Hmisc* package, producing a hierarchical clustering of features based on their correlation, in turn, computed with a specified correlation measure (Spearman's ρ rank correlation). Then, we identify clusters by cutting the tree at a given level of ρ^2 that we set at $\rho^2 = 0.64$, which corresponds to a strong

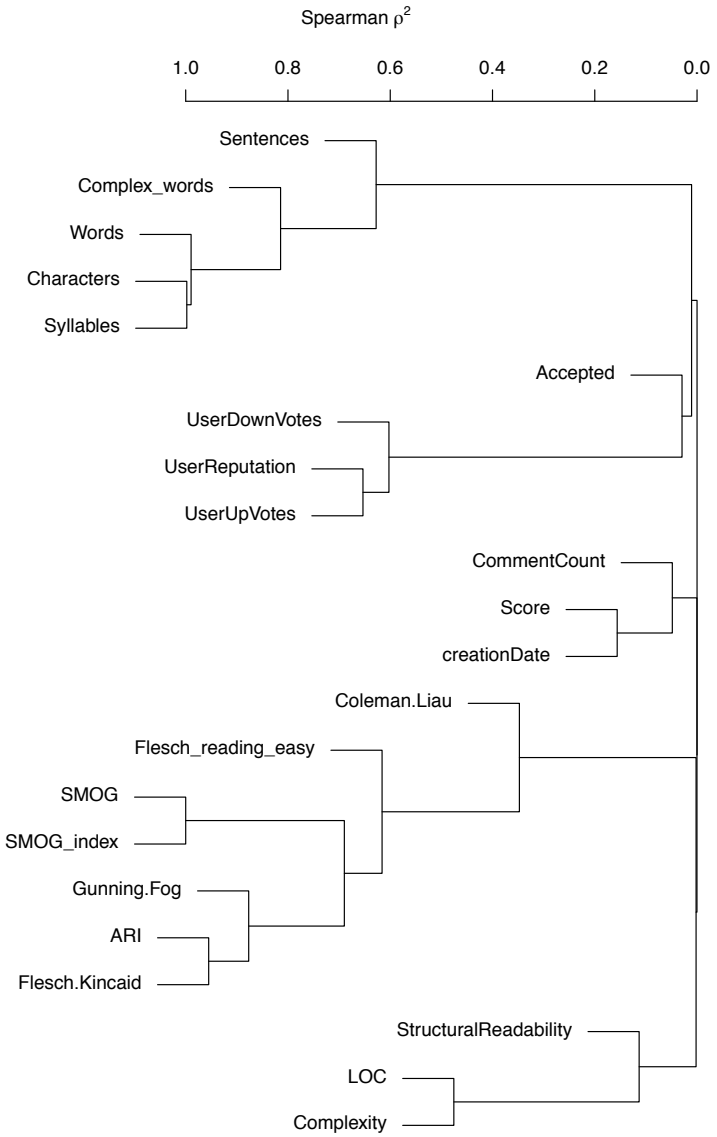

Figure 1: Feature correlation dendrogram (the y-axis shows Spearman's ρ^2)

correlation [14] (*i. e.*, $\rho = 0.8$). As a result of this analysis, we removed the following features:

- User upvotes (correlates with user reputation);
- # Syllables, # Characters, # Complex Words (all correlate with Words); and
- ARI, SMOG, SMOG Index, and Gunning Fog (all correlate with Flesch-Kincaid).

Figure 1 reports the clustering dendrogram of features obtained with the *R varclus* function (the y-axis represents the ρ^2).

We used the ten data buckets previously created to train/test the machine learning algorithms. We start using the first bucket as a train-

ing set for the models, and the remaining nine as a test set. This process is iterated nine times, increasing each time the amount of data (*i.e.*, number of buckets) used for training by 10%. In any case, we make sure that the answers used for training are older than the ones used for testing. For example, in the second iteration, the first two buckets (*i.e.*, 20% oldest answers) are used for training and the remaining 80% of data for testing. In the last iteration, 90% of data is used for training and 10% for testing.

Such a process was preferred to a 10-fold cross-validation since it avoids the use of "data from the future" when predicting the instances in the test set. Indeed, without using the constraints that answers in the training set must be older than the ones in the test set, there is the risk of using, for example, answers from 2018 to predict whether an answer from 2016 will be leveraged or not. This is clearly not representative of a real usage scenario for the predictor; indeed, when predicting whether a new answer will be leveraged, we can only use data from the past to train the classifier. Also, using this experimental design allows to study the impact on the accuracy of the classifier of (i) the amount of used training data, and (ii) the recency of the data used for training.

We experimented four different machine learning techniques implemented in Weka [27], namely Decision Trees (J48), Bayesian classifiers, Random Forests, and Logistic Regression. We used such techniques with their default configuration.

In our dataset we have many more *non-leveraged* than *leveraged* answers. To take into account such a strong unbalancing, we experimented each machine learning technique when (i) not balancing the training sets; (ii) balancing the training sets by under-sampling the majority class by means of the Weka implementation of the *SpreadSubSample* filter; and (iii) balancing the training sets by generating artificial instances of the minority class using the Weka *SMOTE* filter.

We report Precision, Recall, Accuracy, Area Under the Receiver Operating Characteristics curve (AUC), and Matthews Correlation Coefficient (MCC) [39]. Precision and Recall are reported for both prediction categories: *leveraged* (Precision_{l} and Recall_{l}) and *non-leveraged* (Precision_{nl} and Recall_{nl}). We use MCC, since it is a measure used in machine learning assessing the quality of a two-class classifier especially when the classes are unbalanced. It ranges between -1 and 1 (0 means that the approach performs like a random classifier) and it is defined as:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(FN + TN)(FP + TN)(TP + FN)}}$$

The MCC can be interpreted as follows: $MCC < 0.2$ indicates a low correlation, $0.2 \leqslant MCC < 0.4$ a fair, $0.4 \leqslant MCC < 0.6$ a moderate,

$0.6 \leqslant MCC < 0.8$ a strong, and $MCC \geqslant 0.8$ a very strong correlation [14].

We also report information about the importance of the considered factors using the Mean Decrease Impurity (MDI) [37], which measures the importance of variables on an ensemble of randomized trees.

2.2.3 *Replication Package*

The data used in our study is publicly available[8]. In particular, we provide (i) the subject answers together with the factors (see Table 1) we measured on each of them; (ii) the created buckets, together with the training (both the balanced and the unbalanced) and test sets as `arff` files to be used in *WEKA*; and (iii) the detailed results for the accuracy of each machine learning algorithm.

2.3 RESULTS

In this section we present and discuss results aimed at addressing the research questions formulated in Section 2.2.

2.3.1 *To what extent are code snippets from "non-leveraged" answers used in GitHub projects?*

We found that, out of the 500 snippets considered as *non-leveraged*, only 30 (6%) have at least one detected clone in the considered GitHub files. Thus, while we acknowledge a certain level of noise in our analysis (*i. e.*, misclassification of leveraged snippets as non-leveraged), we believe that the findings reported in the following are unlikely to be substantially influenced by such a noise.

2.3.2 *Which are the characteristics of SO answers that have been leveraged by developers?*

Table 2 reports Cliff's delta (d) effect size, together with its interpretation (*i. e.*, L = Large, M = Medium, S = Small, N = Negligible) for the studied factors when using buckets of different size and recency, as explained in Section 2.2. A positive effect size value indicates that the value for a given factor (*e. g.*, the *Comment Count*) is higher in the group of leveraged SO answers, while the opposite holds for negative values. Empty cells in Table 2 represent combinations of factors/-datasets for which the obtained p-value is not statistically significant.

By looking at Table 2, the first thing that leaps to the eyes is that results are consistent among the different buckets, with very minor vari-

8 https://github.com/leveraged-SO/leveraged-SO

Table 2: RQ_1: Effect size for statistically significant differences (N=Negligible, S=Small, M=Medium, L=Large).

Feature	% Dataset Size								
	10%	20%	30%	40%	50%	60%	70%	80%	90%
Is Accepted									
Answer Score	+.72 (L)	+.79 (L)	+.82 (L)	+.82 (L)	+.81 (L)	+.82 (L)	+.82 (L)	+.82 (L)	+.82 (L)
Comment Count	+.52 (L)	+.54 (L)	+.54 (L)	+.53 (L)	+.53 (L)	+.52 (L)	+.51 (L)	+.50 (L)	+.50 (L)
User Reputation		+.06 (N)	+.04 (N)					−.03 (N)	
User Up Votes			−.05 (N)	−.10 (N)	−.11 (N)	−.10 (N)	−.09 (N)	−.10 (N)	−.09 (N)
User Down Votes		−.07 (N)	−.09 (N)	−.14 (N)	−.16 (S)	−.16 (S)	−.15 (S)	−.17 (S)	−.16 (S)
Snippet LOC		+.07 (N)	+.09 (N)	+.10 (N)	+.12 (N)	+.14 (N)	+.16 (S)	+.16 (S)	+.16 (S)
Snippet Complexity	+.15 (S)	+.18 (S)	+.20 (S)	+.21 (S)	+.22 (S)	+.23 (S)	+.24 (S)	+.24 (S)	+.24 (S)
Snippet Readability							+.03 (N)	+.04 (N)	+.04 (N)
# Words	−.07 (N)	−.05 (N)	−.05 (N)	−.06 (N)	−.04 (N)	−.03 (N)	−.04 (N)	−.04 (N)	−.05 (N)
# Sentences				−.04 (N)					
# Characters	−.07 (N)	−.05 (N)	−.05 (N)	−.06 (N)	−.04 (N)	−.03 (N)	−.04 (N)	−.04 (N)	−.05 (N)
# Syllables	−.07 (N)	−.05 (N)	−.05 (N)	−.06 (N)	−.03 (N)	−.03 (N)	−.03 (N)	−.04 (N)	−.04 (N)
# Complex Words	−.07 (N)	−.05 (N)	−.04 (N)	−.04 (N)					−.03 (N)
ARI	−.07 (N)	−.06 (N)	−.04 (N)	−.04 (N)	−.05 (N)	−.05 (N)	−.05 (N)	−.05 (N)	−.05 (N)
SMOG	−.06 (N)	−.05 (N)				−.03 (N)	−.03 (N)	−.03 (N)	−.03 (N)
SMOG Index	−.06 (N)	−.05 (N)				−.03 (N)	−.03 (N)	−.03 (N)	−.03 (N)
Flesch Kincaid	−.07 (N)	−.05 (N)			−.04 (N)	−.04 (N)	−.04 (N)	−.04 (N)	−.04 (N)
Flesch Reading Easy									
Coleman Liau									
Gunning Fog	−.06 (N)	−.07 (N)	−.04 (N)	−.04 (N)	−.05 (N)	−.05 (N)	−.05 (N)	−.05 (N)	−.05 (N)
Creation Date	−.39 (M)	−.45 (M)	−.43 (M)	−.40 (M)	−.39 (M)	−.36 (M)	−.37 (M)	−.40 (M)	−.44 (M)

ations. For example, the *Answer Score* factor always exhibits a large, positive effect size, suggesting that leveraged SO answers usually have higher score values as compared to non-leveraged ones. Based on the available data, this factor seems to be the most prominent one in characterizing leveraged answers, followed by *Comment Count*.

Among the community factors, it is surprising to see that the *Accepted Answer* does not play a major role here. However, it is worth noticing that not all SO questions have an accepted answer. Indeed, as also extensively discussed in the Stack Exchange community[9], not all users take the time to mark as accepted one of the answers they got in response to a question they posted on the website.

Most of the user-related factors (*i. e.*, factors characterizing the user who post the answer) exhibit negligible and not statistically significant differences between leveraged and non-leveraged answers. The only exception here is for the *User Down Votes*, that exhibit a small effect size with the increase in the size of the dataset. It is important to note the negative sign of the effect size, indicating that the number of downvotes is higher in non-leveraged answers.

For what concerns the quality of the code snippet embedded in the answer, the code readability does not seem to play a major role, while snippets that are larger (*Snippet LOC*) and more complex (*Snippet Complexity*), tend to be leveraged more, even though the effect size is small. While this result might look surprising, it indicates that developers are more likely to leverage non-trivial code rather than very simple small snippets. In other words, when developers look for information on SO and link a specific post in their source code, this code is likely to be more complex than that in non-leveraged posts.

The readability of the text contained in the answer (see factors from # *Words* to *Gunning Fog* in Table 2) only exhibits negligible differences between the two compared sets of posts, possibly due to the fact that these metrics were not conceived to assess the readability of technical documents, such as the one in SO discussions.

Finally, the post *Creation Date* exhibits a statistically significant difference and medium effect size. The negative values here indicate something quite expected: Oldest posts are more likely to have been leveraged by developers. This finding also justifies our temporal buckets-based approach both in RQ_1 and, especially, in the training of the classifier in RQ_2. Indeed, since the "age" of a post plays a role in our data, it is important to analyze how the accuracy of a classifier varies when training it on older or newer data.

While the analysis of Table 2 provided us with good insights on the characteristics of leveraged and non-leveraged answers, it is important to highlight that it has been performed on SO answers belonging to different discussions. In particular, for a given SO discussion, it is

9 https://meta.stackexchange.com/questions/119197/problem-with-users-not-accepting-answers

possible that none of the answers we analyzed have been leveraged, while for others it could happen that all the considered answers have been linked in some open source project.

Despite this does not introduce any noise in our data, it does not help in controlling for possible confounding factors. Indeed, it might be possible that some answers belonging to a SO discussion are not reused because their topic has a very narrow interest, rather than because of the factors we considered, *e.g.*, because it received few comments or because it came from a user with a low reputation.

For these reasons, we replicated the previous analysis when only considering the 104 SO discussions in our dataset having at least one leveraged and one non-leveraged answer (*i.e.*, paired analysis). We consider the dataset composed by the 90% of data, to avoid having a too-small number of discussions to consider for this analysis. Table 3 reports the achieved results.

On this (much smaller) dataset, most of our main findings are confirmed, meaning the role played by *Answer Score*, *Comment Count*, *User Down Votes*, *Creation Date*. Instead, we did not observe any significant difference for factors related to the quality of the code snippet. However, this might be due to the small size of this dataset. The main difference is that the User-related factors also play a significant role, with a large effect size for *User Reputation*. Also, when focusing only on the answers to a specific question, it seems that developers tend to leverage more accepted answers.

Despite the slightly different results obtained in the unpaired and in the paired scenario, both analyses showed the presence of statistically significant differences in the distribution of some of the considered factors between leveraged and non-leveraged answers. This suggests the possibility of automatically discriminate between these two answers' sets, as we investigate in RQ_2.

RQ_1 summary: post-related features such as *Answer Score*, *Comment Count* and post *Creation Date* exhibit significant and large/medium differences between leveraged and non-leveraged posts. Snippets in leveraged posts are longer and more complex than in non-leveraged posts. Finally, readability metrics play a negligible role.

2.3.3 *Which is the performance of a recommender system in identifying posts that are likely to be leveraged by developers?*

Table 4 reports the accuracy of the machine learning classifiers in the different configurations we experimented. Table 4 reports the overall results achieved across the nine buckets used for the training. This means that for each run (*i.e.*, the first run refers to the usage of 10% for training, 90% for testing; the second 20%-80%; etc. up to 90%-10%), we counted the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Then, we count

Table 3: RQ_1: Paired analysis of features' distribution on 104 discussions having both leveraged and non-leveraged answers.

Feature	***p*-value**	***d***	**Magnitude**
Is Accepted	< 0.01	0.36	medium
Answer Score	< 0.01	0.73	large
Comment Count	< 0.01	0.76	large
User Reputation	< 0.01	0.49	large
User Up Votes	0.01	0.26	small
User Down Votes	< 0.01	0.30	small
Snippet LOC	0.44	−0.05	negligible
Snippet Complexity	0.39	0.10	negligible
Snippet Readability	0.14	0.12	negligible
# Words	0.07	0.10	negligible
# Sentences	0.07	0.14	negligible
# Characters	0.07	0.09	negligible
# Syllables	0.08	0.08	negligible
# Complex words	0.14	0.06	negligible
ARI	0.27	−0.10	negligible
SMOG	0.14	−0.14	negligible
SMOG Index	0.14	−0.14	negligible
Flesch Kincaid	0.14	−0.13	negligible
Flesch Reading Easy	0.14	0.15	small
Coleman Liau	0.39	−0.07	negligible
Gunning Fog	0.14	−0.12	negligible
Creation Date	< 0.01	−0.65	large

the overall number of TP as the sum of the TP identified in all nine runs and apply the same procedure also for TN, FP, FN. Based on such numbers, we computed the statistics reported in Table 4.

The Random Forest consistently ensures slightly better performance than the other algorithms, independently from the adopted balancing strategy. The balancing of the training set does not help the Random Forest in obtaining a better classification. Indeed, its performance is quite stable, with the balancing (both under and oversampling) helping the recall of the *leveraged* class while penalizing its precision. In general, the performance achieved by the top configuration, while being far from those of a perfect model, are satisfactory.

The AUC of 0.856 indicates that the model is much better than a random classifier (AUC = 0.5), and the MCC reports a moderate correlation (0.528). Particularly relevant for the purpose of our study is the precision achieved by the classifier when identifying *leveraged*

Table 4: RQ_2: Prediction accuracy of the experimented classifiers in different configurations. Configurations are reported in descending order by MCC (Matthews Correlation Coefficient).

Machine Learner	SMOTE	Under sampling	Recall_{l}	Precision_{l}	Recall_{nl}	Precision_{nl}	Accuracy	AUC	MCC
RandomForest	✗	✗	0.490	0.654	0.973	0.949	0.928	0.856	0.528
RandomForest	✓	✗	0.521	0.615	0.966	0.952	0.925	0.866	0.526
RandomForest	✗	✓	0.591	0.537	0.948	0.958	0.914	0.869	0.516
J48	✓	✗	0.529	0.552	0.956	0.952	0.916	0.790	0.494
J48	✗	✗	0.510	0.553	0.958	0.950	0.916	0.743	0.485
Bayesian	✗	✗	0.441	0.583	0.968	0.944	0.918	0.848	0.464
Bayesian	✗	✓	0.510	0.508	0.949	0.950	0.908	0.845	0.458
J48	✗	✓	0.592	0.437	0.922	0.956	0.891	0.746	0.449
Bayesian	✓	✗	0.315	0.655	0.983	0.933	0.921	0.833	0.419
Logistic	✗	✗	0.150	0.764	0.995	0.919	0.916	0.777	0.315
Logistic	✗	✓	0.234	0.524	0.978	0.925	0.909	0.762	0.309
Logistic	✓	✗	0.241	0.461	0.971	0.926	0.903	0.762	0.287

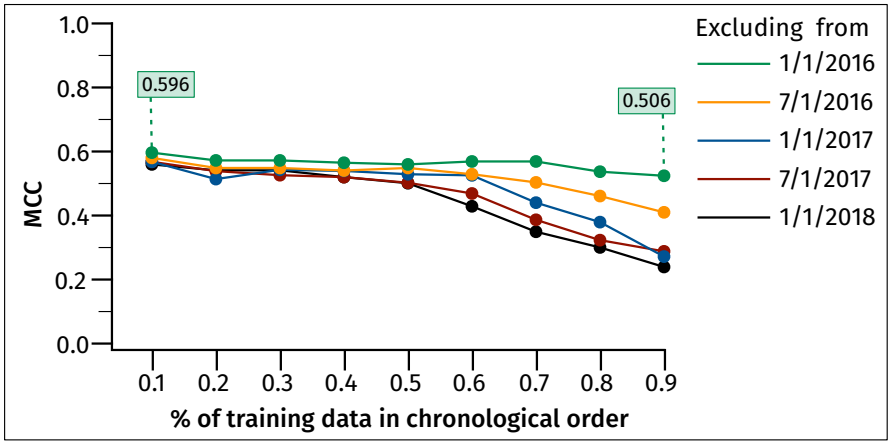

Figure 2: RQ_2: MCC achieved by the Random Forest with different datasets and percentage of training data.

answers ($Precision_l$ = 0.654). This means that 65% of the answers identified as *leveraged* by the classifier have been linked in at least one open-source project on GitHub.

Once identified the best classifier/configuration, we further dig into factors that affect its performance, starting from the amount and recency of training data. As explained in Section 2.2, we excluded from our study answers posted after Jan 1, 2018, to avoid recent posts. Starting from this dataset (from now on referred to as D^1_{18}), we study the impact of the recency of used data by creating other four datasets, obtained excluding from D^1_{18} the most recent answers at steps of six months. Thus, the first dataset includes only answers posted until Jul 1, 2017 (D^7_{17}), the second until Jan 1, 2017 (D^1_{17}), and so on, until D^1_{16}. This analysis aims at verifying our conjecture that older data are more "reliable" to build our prediction model since "old" answers that have never been leveraged are unlikely to be leveraged in the future, while recent *non-leveraged* answers might not being exploited yet due to their recency. To also study the impact of the training data on the accuracy of the prediction model, we split also these four datasets into ten buckets, as already described for the original D^1_{18} dataset (*i. e.*, the first 10% contains the oldest answers).

Fig. 2 and 3 report the MCC and the $Precision_l$, respectively, obtained by the Random Forest (i) on the five different datasets, containing SO answers characterized by different recency (see the five lines having different colors), and (ii) when using different buckets for training (*i. e.*, oldest 10%, 20%, etc., see x-axis).

As conjectured, older data ensures better prediction accuracy. Indeed, the green line, indicating the D^1_{16} dataset, is on the top of both graphs, thus showing a higher MCC/$Precision_l$ achieved by the Random Forest in D^1_{16} as compared to newer datasets. Second, the amount of training data does not strongly influence the model's accuracy. The MCC goes down with the increase of the training set size.

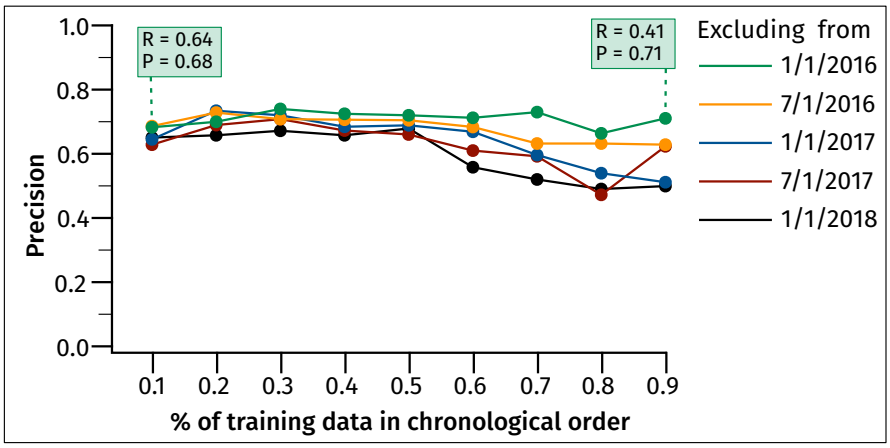

Figure 3: RQ_2: $Precision_l$ achieved by the Random Forest with different datasets and percentage of training data.

However, this result can be explained by the interaction between the "post recency" and the "training set size" factors. Indeed, due to our experimental design, when we increase the training set size, we add more recent answers to it (since the buckets are chronologically ordered). Thus, if we move from 10% to 90% of training data, we are adding in the training set answers that are much more recent than those contained in the first bucket (10%). As previously shown, the Random Forest works better on older data. Thus, the possible boost in accuracy given by the increase of the training dataset size is counterbalanced by the increased recency of the training answers.

Finally, Table 5 reports the values of Mean Decrease in Impurity (MDI) of features used to predict *leveraged* answers. The results refer to the D^1_{18} dataset: We computed the MDI values when using each of the nine training sets (from 10% to 90% of data) and, then, computed the average and the 95% confidence interval for each feature.

Table 5 confirms the major role played by features related to *community factors* (*i. e.*, *Is Accepted*, *Comment Count*, *Answer Score*, *User Reputation*, and *User Down Votes*). Following, are the features characterizing the answer's code snippet (*i. e.*, *LOC*, *Snippet Complexity*, and *Readability*), while less relevant for the prediction are factors related to the readability of the answer's text.

RQ_2 summary: The Random Forest is the classifier ensuring the best accuracy, with an AUC of 0.86 and an MCC of 0.53. The classification accuracy is strongly influenced by the recency of the data, indicating that the categorization of older answers as *leveraged* and *non-leveraged* is more reliable than that of more recent answers. The *community factors* are considered as the most important features to discriminate between *leveraged* and *non-leveraged* answers.

Table 5: RQ_2: Mean Decrease Impurity (MDI) of features used to predict post reference.

Feature	MDI
Is Accepted	0.502 ± 0.020
Comment Count	0.416 ± 0.006
Answer Score	0.396 ± 0.003
User Reputation	0.372 ± 0.004
LOC	0.356 ± 0.003
User Down Votes	0.351 ± 0.002
Snippet Complexity	0.323 ± 0.009
Snippet Readability	0.322 ± 0.005
# Words	0.304 ± 0.006
# Sentences	0.296 ± 0.005
Flesch Kincaid	0.284 ± 0.005
Flesch Reading Easy	0.268 ± 0.003
Coleman Liau	0.260 ± 0.007
Creation Date	0.243 ± 0.006

2.4 THREATS TO VALIDITY

Construct validity. The most crucial threat is the way we determine whether a Stack Overflow post has been leveraged. As explained in Section 2.2, we rely on the dataset created by Baltes *et al.* [7], which use post hyperlinks in the source code to determine that a post has been leveraged. Possibly, a better approach, which we plan to use in our future work, would be to complement Baltes *et al.*'s heuristic with clone detection [85] which, however, would be very expensive from a computational point of view due to the need for searching each of the ~18k *non-leveraged* snippets in the whole GitHub. As explained in Section 2.2, at least we give an estimation of the percentage of *non-leveraged* snippets that are likely to be misclassified (6%). A second threat to construct validity is that the resulting data may depend on the accuracy of the text readability metrics, *i. e.*, ARI, SMOG, SMOG Index, Flesch Kincaid, Flesch Reading Easy, Coleman Liau, and Gunning Fog.

Internal validity. We used default settings for the machine learning algorithms, therefore it is possible that a better calibration could improve their results. Nevertheless, this means that the achieved results represent a lower bound of the classifications' performance. Although we consider features characterizing SO posts from different perspectives, it is entirely possible that the developers' criteria for choosing whether to leverage a post or not are based on factors we do not capture. We have mitigated this threat by performing RQ_1, a paired,

within-post, analysis of the considered features for 104 posts having both leveraged and non-leveraged answers.

External validity. We are aware that our work is (i) limited to Java-related posts (due to the need for using Java metrics extraction, in particular, readability metrics), and (ii) it relies on observable evidence of post leverages as reported by Baltes *et al.* [7]. Further work is needed to replicate the study on posts containing snippets written in other languages.

2.5 RELATED WORK

We discuss the related literature about (i) reuse of code from the Internet, (ii) empirical models built on top of SO data, and (iii) development of recommender systems based on SO.

2.5.1 *Reusing Code From the Internet*

Xia *et al.* [83] showed that one of the common activities performed by software developers while searching on the Web is to look for code snippets to reuse. Based on their results, it is not surprising that many researchers studied the reuse of code snippets across the Internet.

Sojer and Henkel [69] focused on the legal and economic risks of code reuse from the Internet. They surveyed 869 professional developers to investigate whether the reuse of code snippets from the Internet is a common practice in commercial software. Furthermore, the analysis shows a growth in the importance of code reuse from the Internet in recent years. For such reasons, work aimed at better characterizing SO posts worthwhile of being reused — like the one proposed in our work — is highly desirable.

Baltes and Diehl [6] presented a large-scale empirical study investigating the ratio of unattributed usages of SO code snippets on GitHub. In particular, they analyzed the *copy-paste* and *ad-hoc* usage of code snippets into public Java GitHub projects. The results of their study show that half of the developers copying SO snippets do not put proper attribution for the reused code, resulting in three-quarter of reused code snippets that are not correctly attributed on GitHub projects. Attribution and licensing issues are out of the scope of our work, but, for example, licensing constraints are certainly a factor that can constrain code reuse [76].

Yang *et al.* [84] analyzed more than 3 million SO code snippets across four programming languages: C#, Java, JavaScript, and Python, to study the usability of snippets contained in *accepted answers*. They found that less than 4% of Java snippets are parsable and only 1% are compilable. The situation is substantially different for Python, for which they found 72% of code snippets to be parsable and 26% to also be runnable. Unlike our study, Yang *et al.* were interested in un-

derstanding whether code snippets *can be easily reused*, while we focus on factors influencing the reuse of SO posts on GitHub.

Yang *et al.* [85] analyzed 1.9 Million Python code snippets posted on SO and more than 900k Python projects on Github to investigate if SO code snippets are used in real projects. They performed three types of analysis: a perfect match between the SO snippet and the GitHub code, differences due to syntactic elements, and partial clones. They found 1,566 copy-paste blocks, 9,044 blocks which vary for spaces, tabs etc., and more than 405k GitHub blocks that are partial clones of SO snippets. Rather than using clone detection, we use references inside source code (from Baltes *et al.* dataset [7]) to capture reuse, also because as explained in the introduction we are interested to capture reuse in a broad sense, not limited to snippet copy-paste. However, in future, the SOTorrent dataset could be combined with clone detection results.

Other works analyzed the reuse of SO code snippets in Android apps [1, 2, 22], looking at their impact on the app's code quality in terms of reliability [1] and security[2, 22].

A specific study on snippets licensing and attribution was conducted by Le An *et al.* [5]. By analyzing 399 Android apps, they found 266 snippets potentially reused from SO, 1,226 SO posts containing examples from the apps, and a total of 1,279 licensing violations. Considering licensing is complementary to our purpose: once we determine whether a post has been leveraged, it might be necessary to determine whether one could legally reuse its code or not.

2.5.2 *Prediction Tasks on Stack Overflow*

Related to our work are also approaches that try to predict/classify properties on SO items (*e. g.*, questions, answers).

Regarding the prediction of questions quality, Correa and Sureka [18] proposed a model to detect questions that are likely to be deleted.

They used 47 features divided into four categories: question content (*e. g.*, number of URLs, code snippet length), syntactic style (*e. g.*, number of words in title/body), user profile (*e. g.*, number of previous questions/answers), and community-generated (*e. g.*, average question score, average number of accepted answers). To train the model, they selected 235k deleted and non-deleted questions (470k in total). Results show that the model is able to correctly classify both types of questions with a 66% accuracy.

Xia *et al.* [82] proposed an approach to predict deleted questions which combines two independent classifiers, one based on textual features and the other built on the 47 features used by Correa and Sureka [18]. Results show that combining multiple features can help to better discriminate deleted from non-deleted questions.

Ponzanelli *et al.* [57] used machine learning and genetic algorithms to identify poor-quality questions at their creation time. They used three families of metrics as independent variables to assess the questions' quality: SO metrics (*i. e.*, length of the question, the textual similarity between title and body, number of tags), readability metrics, and popularity metrics (*i. e.*, badged answer and question count, and badges-tags coverage). The "question quality" dependent variable is considered to be *good*, if the question is not closed, not deleted, with an accepted answer, and with a score between 1 and 6, *very good*, if it is *good*, but with a greater score, *bad*, if it is not closed, not deleted, and with a score below 0, and *very bad*, if it is closed or deleted. Results show that the popularity of the author is more important than textual features to determine the quality of a new question. In our work, we also consider author- and post-related features and, indeed, our results indicate that some of them are good predictors of reuse.

There have also been different works dealing with the prediction of relevant tags for SO questions. Xia *et al.* [81] proposed TagCombine, a framework able to analyze objects in software information sites and provide tags recommendations. Saini and Tripathi [63] and Wang *et al.* [77] also proposed approaches to predict SO question tags. We do not consider how tags could influence reuse, but this is certainly a direction for future work.

In summary, while several studies investigated the characteristics of SO posts, to the best of our knowledge, our study is the first to investigate the possibility to *predict* SO posts that are likely to be leveraged in open source projects.

2.5.3 *Stack Overflow in Recommender Systems*

Among the various sources available on the Web, Q&A Websites and in particular SO, have been exploited by many recommender systems for software developers to identify pieces of information relevant for a given task at hand. Examples of these systems include (but are not limited to): techniques to identify code elements contained in SO answers [60]; the automatic recommendation of SO discussions for a given task run by the developer in the IDE [17, 56]; and support for the automatic documentation of source code using information mined from SO [3, 72, 80].

However, it is important to highlight that our study can disclose empirical evidence about the possibility of automatically identify SO posts that are "worth reusing". Such information can be of paramount importance for the development of better recommender systems able to select relevant posts *as humans would do*.

2.6 CONCLUSION

In this chapter we studied the characteristics of Stack Overflow (SO) answers that have been *leveraged* by developers, who explicitly linked such answers in their source code. Different characteristics of the *leveraged* answers, including user-related, post-related, and snippet-related characteristics, have been compared to those of *non-leveraged* posts. Results of the comparison highlighted, through statistical analysis, significant differences between *leveraged* and *non-leveraged* answers. In particular, the differences were large for what concerns post-related and community-related factors, such as the quality of the answer as perceived by the SO users (*e. g.*, answer score), and the engagement it created on the platform (*e. g.*, comment count).

Starting from this result, we experimented with machine learning algorithms, assessing their ability to automatically discriminate between *leveraged* and *non-leveraged* answers. We obtained encouraging results (AUC = 0.856 and MCC = 0.528). This points to the possibility of integrating, in recommender systems, "filtering" mechanisms able to identify "high-quality" content in the sea of data available on Q&A websites. This present work is thus only a first step in that area, paving the way for several research directions.

A more comprehensive study can be run by considering as leveraged posts not only those explicitly linked in source code files of open source projects, but also those from which code snippets have been reused [85]. This can help in enlarging the set of leveraged posts, better studying their characteristics and, possibly, improving the performance of the prediction models.

Related to this point is the possibility to factor into our models additional predictor variables. For example, whether the code snippet included in the answer is easy to reuse (*e. g.*, can be parsed [84]). Indeed, our models use a limited set of predictor variables that can be easily expanded.

Finally, the long term goal is the integration of the prediction models able to discriminate high-quality content into one of the recommender systems exploiting SO (*e. g.*, [56]) to verify, through user studies, whether they contribute to recommend more relevant information items.

3

A DEVELOPER-CENTRIC NATURALNESS MODEL FOR PREDICTING CODE UNDERSTANDABILITY

3.1 INTRODUCTION

Developers often struggle reading and understanding code. Despite the common beliefs, code writing only takes a minority of the developers' time, while up to 70% is spent in code comprehension [43]. Being able to objectively measure how *understandable* source code is would be the first step to recognize which parts of code need attention and should be improved to ease code maintenance. Measuring code understandability is, however, far from trivial, probably due to its subjective nature (*i. e.*, a code component can be understandable for a given developer while being difficult to comprehend for another one). Previous studies tried to address the problem of automatically measuring code understandability [65, 66]. However, the achieved results showed the strong limitations of state-of-the-art metrics, unable to exhibit any meaningful correlation with code understandability. For example, Scalabrino et al. [66] experimented with 121 code-related, documentation-related, and developer-related metrics, showing that none of them is able to capture code understandability as perceived by developers.

Intuitively, if a developer knows the programming language constructs, APIs, and code templates used in a source code snippet, she will be able to understand it more easily. In other words, it can be said that the code is "predictable" for the developer. Previous work studied the source code predictability, defining the concept of *naturalness of software* [29]. Hindle et al. [29] using the adjective "natural" indicate that *"code, despite being written in an artificial language (like C or Java) is a natural product of human effort"* and that's why *"programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks"*. Software naturalness is captured through statistical language models, that have also been successfully used in the context of software-related tasks, such as providing code recommendation [48, 59].

Language models need to be *trained* on a corpus of code artifacts: given the size of the context k, the model infers the probability that a given sequence of tokens $\langle t_1, ..., t_k \rangle$ is followed by any other token from the corpus. For example, for $k = 3$ and given the context *e. g.*, $\langle$`for`, `(`, `int`$\rangle$, the probability that the next token is `i` may be relatively high since such an identifier is very commonly used in `for`

Figure 4: Example of JDBC snippet.

```
Class.forName(DB_DRIVER);
Connection c = DriverManager.getConnection(DB_CONNECTION, DB_USER,
     DB_PASSWORD);
String query = "SELECT * FROM users";
PreparedStatement s = c.prepareStatement(query);
ResultSet r = s.executeQuery();
while (r.next()) {
    System.out.println(r.getString("name"));
}
```

loops. Given a source code fragment, language models can be used to compute its naturalness: the higher the quantity of "surprising" (*i.e.*, unlikely) tokens in the code, the lower its naturalness. Ray et al. [58] showed that unnatural code is more likely to contain bugs and, in this chapter, we argue that this might be due to the poor understandability of unnatural code (*i.e.*, developers struggle to understand unnatural code and, as a consequence, they tend to introduce bugs while working on it). Thus, we conjecture that code naturalness can be a good proxy for code understandability, provided that a strong assumption usually made to compute code naturalness is dropped: language models have always been trained using a quite large code corpus which supposedly represents what a *typical* developer of a given programming language is or is not likely to write. While in most of the application contexts this is an acceptable assumption (*e.g.*, code suggestions), to capture code understandability this assumption might be too strong. For example, code understanding depends on the past knowledge and experience of developers [66]: different developers may be more or less familiar with different snippets. Consider the example in Figure 4: back-end developers could be very familiar with such a pattern of API calls since they often deal with similar code. On the other hand, front-end developers may find the same snippet confusing since they are used to other types of APIs, *e.g.*, GUI-related ones.

In this chapter, we investigate the relationship between naturalness and source code understandability by relying on a developer-centric model of code naturalness. Given a developer and its experience in some programming-related topics, we train a language model by only considering a corpus of code snippets related to the topics she has experience with. For example, if a developer is very proficient with multi-threading programming, the corpus will contain many code snippets using multi-threading. The number of code snippets in the training corpus for a given topic (*e.g.*, multi-threading), will be proportional to the self-assessed experience of the developer on that

topic. The developer-centric model aims at simulating the mental language model of a specific developer.

Based on this idea, we conduct an empirical study in which we aim at verifying if the naturalness computed with a developer-centric model is a good predictor for any aspect of code understandability. As a first step, we perform a preliminary empirical investigation to motivate the need for a developer-centric model for naturalness. More specifically, we investigate whether source code naturalness correlates with its understandability. This study provided negative results, and motivates the need for a deeper understanding of the naturalness, *i. e.*, introducing the developer-centric model.

Therefore, we conduct a second study, involving 52 developers, on the relationship between developer-centric naturalness and source code understandability. The study consists of two phases. In the first phase, we assessed the experience of such developers in eight Java-related topics we identified (*e. g.*, collections, reflection, threads, etc.). We used such information to build a developer-centric language model for each of them and exploited this model to compute the developer-centric naturalness for a number of code snippets. We selected, for each developer, both *natural* and *unnatural* snippets and we asked the same developers to read and understand these two sets of snippets and to answer some verification questions aimed at assessing their actual understanding of the snippets. Using a design similar to the one used in previous works on understandability measurement [65, 66], we collected information about the (i) perceived comprehension (*i. e.*, how much the developer believes to have understood each snippet), (ii) actual comprehension (*i. e.*, how much she was actually able to answer verification questions about the snippet), and (iii) time needed to understand the snippet. Finally, we compute the correlation between the developer-centric naturalness and different proxies of code understandability. We also compare the developer-centric model with a generic language model usually employed to compute code naturalness.

The overall methodology of our work, consisting of the first motivating study and the second, two-phase study on developer-centric naturalness, is depicted in Figure 5.

Our results show that code naturalness computed using a developer-centric model is significantly associated with a higher code deceptiveness: when faced with unnatural code, developers more often perceived that they understood the code (*i. e.*, perceived understandability) while, in reality, they did not (*i. e.*, actual understandability). Such a correlation does not hold for the generic language model. Our results pave the way to (i) a possible usage of developer-centric naturalness for the automatic assessment of code understandability; and (ii) future applications of the developer-centric naturalness models (*e. g.*, developer-specific code completion models).

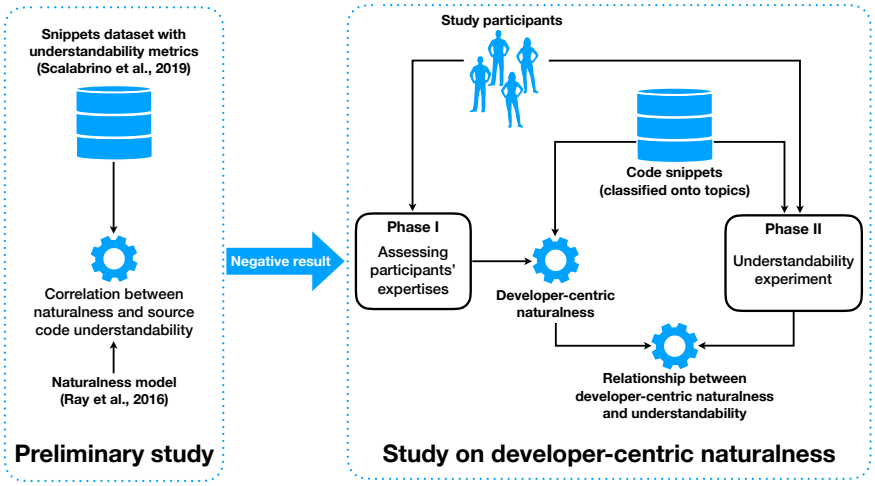

Figure 5: Empirical investigation methodology: preliminary study and study on developer-centric naturalness.

3.2 BACKGROUND AND RELATED WORK

In this section we introduce approaches and metrics proposed in the literature to measure source code understandability. Also, we provide background notions about source code naturalness and discuss studies aimed at correlating naturalness with other source code properties.

3.2.1 *Measuring Code Understandability*

Lin and Wu [36] proposed a model for evaluating code understandability. The model uses Principal Component Analysis (PCA) and fuzzy mathematics on a set of metrics (*e. g.*, understandability of documentation, of components, of data, etc.) assumed to capture understandability aspects based on the analysis of the literature. The model has not been evaluated.

Misra and Akman [44] presented the Cognitive Weight Complexity Measure (CWCM) metric and compared it to existing cognitive complexity metrics. The basic idea behind the CWCM is to weight the difficulty in comprehending a given component based on the control structures it contains. The evaluation of CWCM presented in Misra and Akman [44] is purely theoretical, based on the number of properties proposed by Weyuker [78] that CWCM satisfies.

Thongmak and Muenchaisri [71] considered aspect-oriented software dependence graphs to assess the understandability of aspect-oriented software, while Srinivasulu, Sridhar, and Mohapatra [70] estimated software understandability using an outlier detection approach. Specifically, they calculated seven metrics (*e. g.*, comment ra-

tio, number of components, Halstead Complexity) on nine projects, and determined the rough entropy factor to identify the outliers. Considering the metric values, the identified outliers correspond to either highly understandable or not understandable projects.

Capiluppi, Morisio, and Lago [12] also presented a metric to capture code understandability using as proxies (i) the number of files in the system's modules (*i. e.*, directories), and (ii) the relative size of the files. The metric has not been empirically evaluated.

The most recent work investigating code understandability is the one by Scalabrino et al. [66]. The authors attempted to correlate 121 different metrics (*e. g.*, cyclomatic complexity, lines of code, developer's experience) to source code understandability. They conducted a study with 63 participants who were asked to understand code snippets, for a total of 444 evaluations. Then, they correlated the 121 metrics to three code understandability proxies: (i) the perceived understandability (*i. e.*, whether the participant felt to have understood a given code snippet), (ii) the actual level of understanding (*i. e.*, the ability of the participant to answer questions about the snippet), and (iii) the time spent by the participant in understanding a snippet. Their results show that no metric is able to capture code understandability.

Related to understandability are also the studies that tried to define metrics for the readability of code [11, 64] which, however, is only one of the aspects playing a role in code understandability and, as shown by Scalabrino et al. [66], does not exhibit strong correlations with code understandability.

To the best of our knowledge, this is the first work aimed at relating naturalness with software understandability. On the one hand, we show how a global naturalness model cannot be related to understandability. On the other hand, we build a developer-centric naturalness model which, instead, exhibits a relationship with a specific aspect of software understandability.

3.2.2 *Naturalness of Code*

Gabel and Su [24] studied the uniqueness of software by analyzing the extent to which token-level n-grams are redundant in a corpus of 6k open source projects. They found that source code is highly repetitive, with over 50% of n-grams found in multiple projects for low values of n.

On this same line of research, Hindle et al. [29] showed that source code is "natural", meaning that it is highly repetitive and predictable as compared with traditional English text. When compared to natural language, source code is even more repetitive due to the strong syntax constraints imposed by programming languages. Successive studies

confirmed this finding [45], and some of them highlighted that these repetitions are stronger in specific areas of the source code [35, 73].

Based on these observations, many approaches modeled the source code by using statistical language models with the goal of supporting software engineering tasks such as code completion [29, 47, 48, 59], code migration [46], rename refactoring [4, 35], enforcing stylistic consistency of code [4], code review [28], deobfuscation [75], identification of automatically generated code [20], mutation testing [31] and *defect prediction* [58].

Our assumption is the code naturalness can also play a role in automatically assessing code understandability.

3.3 MOTIVATING STUDY

The *goal* of this study is to investigate the relationship between naturalness and source code understandability. As explained in the introduction, the purpose of this preliminary investigation is to understand whether the existing naturalness measures are sufficient to capture source code understandability properties. The *context* of our study consists of a dataset of 50 Java snippets previously used by Scalabrino et al. [65, 66] to correlate understandability with 121 source code metrics. The level of understandability associated with each snippet has been computed through a study performed with 63 developers. Details on the computed understandability proxies are provided in the following.

3.3.1 *Research Question and Design*

This preliminary study aims at answering the following research question:

To what extent is the naturalness of a given code snippet related to its understandability?

We leverage the understandability proxies measured by Scalabrino et al. [65, 66] and available in their dataset. They asked a pool of 63 study participants (developers and CS students) to understand eight Java methods (randomly selected from a pool of 50 Java methods) in a survey-based study run through a Web application. For each snippet, participants were asked to comprehend it and, once done to click on the button "I understood the method" or the button "I cannot understand the method". This binary metric is defined as the *perceived understandability*. In both cases, the Web application stored the time spent, in seconds, by the developer for the method's understanding before clicking on one of the two buttons. If the participant clicked on the button "I understood the method", the method was then hidden and she was required to answer three *verification questions* on the

hidden method (see Scalabrino et al. [66] for more details). In this way, the authors captured the *actual understandability*. Using this data, Scalabrino *et al.* measured the following understandability proxies:

- **Perceived Binary Understandability** (*PBU*). This binary metric assumes 0, if the participant answered "I cannot understand the method"; otherwise, it assumes 1.
- **Actual Understandability** (*AU*). This metric obtains 0, if the participant clicked on the button "I cannot understand the method"; otherwise, it was computed as the percentage of correct answers on the three verification questions mentioned above.
- **Actual Binary Understandability** ($ABU_{k\%}$). This metric is derived from *AU* and, with $k = 50\%$, it assumes a binary value used to classify snippets of code as understandable or not. $ABU_{50\%}$ is *true* when participants correctly answer at least two of the three verification questions, and *false* otherwise.
- **Time Needed for Perceived Understandability** (*TNPU*). This metric measures, in seconds, the time needed by the participant to inspect the method and click on the "I understood the method" button. This metric is not computed for methods that are not understood by the participant.
- **Timed Actual Understandability** (*TAU*). This is a measure of the time needed to obtain the *actual understandability*. It gets a value of 0 if the developer perceives that she did not understand the snippet. Otherwise, it is computed as:

$$TAU = AU\left(1 - \frac{TNPU}{\max TNPU}\right)$$

  where *AU* and *TNPU* are the variables previously defined. The higher *AU*, the higher *TAU*, while the higher *TNPU*, the lower *TAU*.
- **Binary Deceptiveness ($BD_{k\%}$)**. This is a binary categorical variable derived from *PBU* and $ABU_{k\%}$, which is *true* if *PBU* is *true* and $ABU_{k\%}$ is *false*, and *false* otherwise. $BD_{k\%}$ indicates whether a developer can be deceived by a method in terms of its understandability (*i.e.*, she incorrectly thinks she understood the method).

In our study, we also add another deceptiveness metric, named **Continuous Deceptiveness** (*CD*). With *CD* we indicate how much a method deceived a developer trying to understand it, *i.e.*, to what extent the developer says that he understood the method (in terms of *perceived understandability*) but in reality, this is not the case (*i.e.*, the actual understandability is low). The *Continuous Deceptiveness* is calculated as follows:

$$PBU \times (1 - AU)$$

where *PBU* is the *perceived understandability*, and *AU* is the percentage of verification answers correctly answered by the participant. If a developer perceives that she did not understand the method, the *CD* is zero, since for sure the snippet of code did not deceive the developer. Otherwise, if a developer perceives that she understood a code snippet, but this is not actually the case, then the deceptiveness increases on the number of wrong answers (reaching a maximum of 1 for no correct answers provided).

The naturalness is measured using the model proposed by Ray et al. [58]. To train the model, we used 40 Java projects selected as follows. Our goal was to train the model on code written by a high number of developers, with the assumption that such a model could be more representative of the code naturalness as perceived by Java developers. We used Google BigQuery to extract contributors for each GitHub project. Then, we extract the total number of contributors and we order all projects in descending order of contributors. Unfortunately, the dataset did not allow to automatically distinguish Java projects from non-Java projects. Thus, starting from the project with the maximum number of contributors, we verified manually whether each project was indeed a Java project, and we ignored all non-Java and forked projects. After that, we leveraged a greedy algorithm to select the 40 Java projects covering the greatest number of unique contributors (details are provided in Algorithm 1). Consequently, the additional greedy algorithm returns 40 Java projects with a number of unique contributors equal to 21,219 developers over a total of 673,675 unique developers of all GitHub projects, observing a confidence interval lower than 0.87% computed for a significance level of 99%. The total number of unique developers is an overestimation because this number includes developers of potentially non-Java projects, and consequently, also the confidence interval is an upper bound.

Algorithm 1 Project selection algorithm

Data: GitHub projects P_G ordered in descending order and developers D

Result: selected projects P

1: $P \leftarrow p_1 \in P_G$, where p has the maximum number of developers
2: $D \leftarrow D_{p_1}$
3: $P_G \leftarrow P_G - \{p_1\}$
4: **while** $|P| \leqslant 40$ **do**
5: $\quad P \leftarrow p_i \in P_G$, where p_i adds the maximum number of developers to D
6: $\quad D \leftarrow D \bigcap D_{p_i}$
7: $\quad P_G \leftarrow P_G - \{p_i\}$
8: **end while**

Afterward, we perform a *preprocessing* on the source code of the selected projects, to allow the computation of source code metrics and naturalness. We used *srcML*[1] to parse the Java classes in each project and extract the methods they contain. Then, for each method (i) we remove comments; (ii) we run an abstraction on strings, characters, and numbers, assigning to each of them a specific token (*e. g.*, all strings were replaced with the token `$$string$$`); and (iii) we separate each token (*e. g.*, identifiers) with a space. As output, the tool returns:

- the un-tokenized code without comments in a file `.java` (or original code);
- the tokenized code stored in lines as a file `.java.lines`;
- the tokenized code written in a single line for the training of the *language model* (LM) in a file `.java.tokens`.

The `.java.tokens` files are used to train the LM and to obtain the naturalness values for the 50 Java snippets used in our previous work [65, 66]. Our model is integrated into the tool of Zhaopeng[2], which provides the *entropy* of a given code snippet (*i. e.*, the *unnaturalness*). Specifically, if the value of the entropy is positive, the method is unnatural. Instead, if the value of the entropy is negative, the method is natural.

After having obtained the entropy values, we correlate them with the previously defined metrics of understandability [65, 66], using the Kendall rank correlation coefficient (*i. e.*, Kendall's τ) [33]. We followed Cohen's guidelines [14] to interpret the correlation coefficient. It is assumed that there is *no correlation* when $0 \leqslant |\tau| < 0.1$, *small correlation* when $0.1 \leqslant |\tau| < 0.3$, *medium correlation* when $0.3 \leqslant |\tau| < 0.7$, and *strong correlation* when $0.7 \leqslant |\tau| < 1$.

Also, we compare the values of *CD* and of *AU* between two groups: *natural* and *unnatural*. The groups were obtained by splitting on the mean of naturalness values ($\simeq 7.59$). For the comparison, we use a Mann-Withney test to check if the difference between natural snippets and unnatural snippets is statistically significant. We reject the null hypothesis if the *p-value* is lower than 0.05, and, in case of significance difference, we also use Cliff's delta [13] to assess the magnitude of such a difference. We consider the difference *negligible* if $|\delta| < 0.148$, *small* if $0.148 \leqslant |\delta| < 0.33$, *medium* if $0.33 \leqslant |\delta| < 0.474$, and *large* for $|\delta| \geqslant 0.474$ [25].

1 https://www.srcml.org/
2 https://bitbucket.org/tuzhaopeng/cachelm_for_code_suggestion

Table 6: Results of the Kendall's correlation between understandability and naturalness.

Measure	Correlation	p-value
PBU	0.006	0.87
TNPU	0.017	0.65
AU	−0.020	0.58
TAU	−0.027	0.43
$ABU_{k\%}$	−0.021	0.60
$BD_{k\%}$	−0.033	0.39

3.3.2 *Analysis of the results*

Table 6 reports the results in terms of Kendall's τ rank correlation. As it is possible to notice, entropy does not correlate with any metric of understandability.

Indeed, the obtained correlations are all very low and all values are located in the range of the **no correlation**. Moreover, we did not obtain any significant p-value for the computed correlations.

We then compare *natural* vs. *unnatural* code snippets based on the *CD* and the *AU*. In the first case, there is no significant difference (p-value = 0.79): both for *natural* and *unnatural* we have a median *CD* of 33.33%. In the same way, there is no significant difference (p-value = 0.75) in the *AU* between the *natural* (66.67% median) and *unnatural* (33.33% median) groups.

3.3.3 *Take Away*

Based on the results of this preliminary study, we infer that naturalness is not a good predictor of understandability. Indeed, the achieved results show that understandability does not correlate with the adopted measure of naturalness.

We conjecture that a possible reason for this result is the developer-centric nature of code naturalness. For this reason, we attempted to define a new model of naturalness that is built ad hoc for a given developer, based on her knowledge. In the next section, we explain such a naturalness model, and empirically investigate to what extent it correlates to understandability.

3.4 EMPIRICAL STUDY DESIGN

The *goal* of our empirical study is to investigate the relationship between developer-centric naturalness and source code understandability.

As found in the study described in Section 3.3, a global naturalness model is not able to capture source code properties related to understandability. We conjecture that a naturalness model tailored to a specific developer can help in assessing code understandability. In other words, a source code snippet is perceived as more natural if a developer has already seen/used similar code in the past which, in turn, helps in the snippet understanding. For this reason, we want to investigate if a developer-centric naturalness model behaves differently, and we design a study to answer the following research question:

How does the understandability vary for source code snippets exhibiting different levels of developer-centric naturalness?

3.4.1 *Study Context and Data Collection*

The *context* of our study consists of Java code snippets (*Objects*) and Java developers (*Subjects*). While Scalabrino et al. [66] provide a dataset on code understandability with 444 human evaluations on 50 Java snippets, this dataset is not suitable for our study. Indeed, in order to compute developer-centric naturalness, we need to collect information about the knowledge of the developers involved in the study, to see whether they understand more easily code snippets that are natural for them.

To this aim, we conduct a two-phase survey with a pool of Java developers. In the first phase (*P1*) we collect information about the knowledge of developers for a set of relevant topics concerning Java development. In the second phase (*P2*) we use the information gathered in *P1* to (i) build a developer-centric naturalness model, and (ii) select the snippets that each developer would have to understand in the study.

P1: Topic Knowledge Assessment. We invited 52 developers to a preliminary survey in which we assessed their knowledge about Java topics. All developers have been reached through personal contacts of some authors. In this study, we consider 8 popular Java topics, namely: collections, files, graphical user interfaces (GUIs), database connectivity (JDBC), reflection, servlet, sockets, and threads. The choice is based on the topics covered in the programming courses at the University of Molise, where all the participants studied. We report in Section 3.5 some demographic information about the sample we surveyed.

For each topic, we ask the developer to self-assess her own knowledge. Each developer can either declare no knowledge or else a given level of knowledge expressed on a five-point Likert scale [50] ranging from 1 (very low) to 5 (very high). Then, unless the developer has declared no knowledge, we ask five multiple-answer questions about

the specific topic. Otherwise, we assumed that the developer actually had no knowledge of the topic. We use the official tutorials provided by Oracle[3] to define the questions for each topic. For each developer, we estimate the percentage of knowledge about each topic as the percentage of correct answers given to the five questions we asked. 50 developers completed our survey (~96%), but 15 did not take part in the second phase of our study because they did not agree to continue.

Building the Developer-Centric Naturalness Models. We use the data collected in the first phase to build, for each developer, a developer-centric naturalness model. To train the model, we need to collect code snippets related to each topic. First, we select 100 Java projects on GitHub having the highest number of stars and, for each project, we extract, using the SRCML tool [16], all methods having at least 5 LOCs (*i. e.*, we discarded trivial methods).

After that, we need to associate the extracted Java methods with the eight topics considered in the study. To this aim, we first use the `import` statements to associate each class to a possible topic. For example, `import java.sql.*` or any specific import of that package would be associated with database connectivity. The association was realized through a manually built mapping between APIs and topics. The mapping is available in our replication package[4]. This process gives us classes *possibly* containing methods related to our topics of interest. Note that in this stage, a class can be associated with multiple topics (*e. g.*, it has imports related to both the File and the GUI topics).

Then, for each method of the selected classes, we statically resolve the type of their parameters and local variables as well as of the class's instance variables they use with the goal of associating each variable to one of the eight topics. Java primitive types (*i. e.*, `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `void`) are discarded, so as `String` variables because such a type occurs very frequently. Variables belonging to other types are mapped, thanks to the `import` statements, to the eight topics. For example, a variable of type `File` is mapped to the File category thanks to the existing mapping between its `import java.io.File` and the File category. If a variable cannot be associated with any of the eight topics, it is included in a category named *"Other"*.

Finally, we assign each method to the topic to which most of its variables' usages are linked, given that the class containing it has also been linked to that same topic thanks to the analysis of the `import` statements. Methods assigned to the *"Other"* category are discarded. Table 7 reports the number of resulting methods (in the following also referred to as "snippets") for each topic. Note that some topics are more represented than others. Since, as explained in the following, we used the snippets in these categories to train our developer-centric

3 https://docs.oracle.com/javase/tutorial/
4 https://git.io/JvI2A

Table 7: Number of methods for investigated topics.

Topic	# methods
Collections	32,251
File	11,694
GUI	1,076
JDBC	2,061
Reflection	1,399
Servlet	1,343
Socket	2,225
Thread	3,914

naturalness model, we set a threshold $max_s = 1{,}076$ representing the maximum number of snippets that can be considered from each category when training the naturalness model. The value for max_s is given by the less represented topic (*i.e.*, GUI).

Once each snippet has been associated with a topic, we can build a developer-centric naturalness model N_d of each developer d based on her knowledge about the eight topics, that we estimated in P1 thanks to the survey. We assume that the developer's estimated knowledge can be a proxy for the number and types of code snippets such a developer has seen/written in the past. For example, suppose that a developer managed to correctly answer all our five questions about the *File* topic, three out of five questions related to the *GUI* and *JDBC* topics, and she declared no knowledge for the other five topics. In this case, the training set for the naturalness model consists of max_s randomly selected snippets from the *File* topic (*i.e.*, 1,076 snippets), $0.6 \times max_s$ (0.6 is given by $3/5$ correctly answered questions) snippets from the *GUI* topic (646), and $0.6 \times max_s$ snippets from the *JDBC* topic (646). The composition of the training set aims at mirroring the developer's knowledge in the naturalness model: the higher the developer's knowledge on topic T_j, the higher the number of T_j's snippets in the training (up to max_s). Note that, by construction, the training set size for different developers (models) can be different, and it is larger for developers having high knowledge of several topics.

Snippet Selection for Understandability assessment. Once the developer-centric models have been created, it is possible, given a snippet s_i and a developer d, to estimate the level of naturalness of s for d. For the purpose of our study, we want to relate such a level of naturalness with the understandability d has of s.

To this aim, we selected from the methods in the systems previously used for training the developer-centric naturalness models, all those sized 50 ± 20 LOC and assigned to a topic different from *"Other"*. This resulted in 5,028 selected methods. The choice of enforcing the

size of the methods between 30 and 70 LOC (50±20) was dictated by the will of selecting methods that were not too easy nor too difficult to understand.

From this population of snippets, we selected the snippets to assign to each participant for the understandability task. In the snippets' assignment, a number of constraints must be considered. First, we wanted to limit the number of snippets assigned to each participant to avoid tiring effects. We decided to target ~10 snippets per participant. Second, we wanted each participant to work with code snippets related to topics on which she was knowledgeable or not, as well as on snippets assessed as natural/unnatural by the related developer-centric naturalness model.

To this aim, for a given developer d, we first selected three topics based on the results of P1: (i) the topic on which d had the highest knowledge, (ii) the topic on which d had the lowest knowledge, and (iii) the topic with the median knowledge. In case d had several topics having the lowest (or highest) knowledge (*e. g.*, several topics with zero knowledge), a random one was selected among them. For each topic t selected for d, we sorted all the snippets belonging to such a topic based on the naturalness provided by the model N_d. Then, our goal was to select from each of these three topics the two snippets having the highest naturalness and the two having the lowest naturalness, for a total of 4 (snippets) $\times$ 3 (topics) = 12 snippets to understand for each participant. However, considering that (i) each of the 35 participants had a different naturalness model, and (ii) for each snippet we had to formulate three questions to measure their understandability (details follow), this was unpractical since required the formulation of up to 1,260 questions — 35 (participants) $\times$ 12 (snippets) $\times$ 3 (questions). To reduce such a manual effort, we selected the top and the bottom 10% of snippets in each distribution related to a topic t for each developer d. Then, we run an additional greedy algorithm, described in Algorithm 2. This algorithm takes all candidate sets C_d, where each C_d is related to a developer d. While the cardinality of C is greater than 0, it selects the snippet that covers the largest number of developers, called s. The selected snippet s is inserted in S. Instead, in C are removed all candidate sets C_d related to developers that have already two assigned snippets. In summary, the goal of the algorithm was to select a minimum number of snippets that allowed us to have, for each topic of each participant, the required number of snippets (4 for each topic), two of which having a high naturalness (top 10%) and two having a low naturalness (bottom 10%). In total, we selected 65 snippets.

P2: Understandability Assessment. Once snippets have been selected, we need to assess their understandability from the participants' perspective. We use a Web application to collect the evaluations of the participants. Such a Web application shows each snippet in iso-

Algorithm 2 Snippet selection algorithm

Data: candidate sets C_d for each developer $d \in D$
Result: selected snippets S

1: $S \leftarrow \emptyset$
2: $C \leftarrow \{C_d \ \forall d \in D\}$
3: **while** $|C| > 0$ **do**
4: Select s that maximizes $|\{C_d \text{ s.t. } s \in C_d\}|$
5: $S \leftarrow S \cup s$
6: $C \leftarrow \{C_d \text{ s.t. } |S \cap C_d| < 2\}$
7: **end while**

lation (*i. e.*, each method on a different page) to participants. The Web application allows the study participants to also browse the methods/classes invoked/used by each snippet. The participants were also allowed to browse the Web to look for additional information. This was done to simulate a typical understanding process performed by developers, in which they obviously have access to the network.

We asked participants to carefully read and fully understand each method. As a response, participants had to select one of two options: "I have understood the method" or "I cannot understand the method". If the participants select the first option, they were required to answer three *verification questions* about the method they just inspected. The provided answers were stored for future analysis. In total, we have collected 396 evaluations.

3.4.2 *Data Analysis*

To answer our research question, we focused on two proxies for code understandability. The first one, *AU* (Actual Understandability), was previously defined by Scalabrino et al. [66]. We measure *AU* as the number of correct answers divided by the number of questions (3). The second proxy we considered is the continuous version of *BD* (Binary Deceptiveness) [66]. Binary Deceptiveness is equal to 1 if the snippet deceived the developer and made her answer "I understood the method" while this was not the case (*i. e.*, she answered correctly to less than two questions), to 0 otherwise.

We also made the understandability proxy continuous, by introducing the *CD* (Continuous Deceptiveness), assessing the extent to which a method deceived the participant. *CD* is defined as $PBU \times (1 - AU)$, where *PBU* (Perceived Binary Understandability) is 1 if the developer answered "I have understood the method" and 0 otherwise. Note that, if a participant perceives that she did not understand the method, we can be sure that the method did not deceive such a participant. Otherwise, if the participant perceives that she has understood the method,

then the higher the number of wrong answers, the higher the deceptiveness.

We use **AU** and **CD** as *dependent variables* and we analyze the effect of the *topic knowledge* and the *naturalness* of the snippet as assessed by the developer-centric model on them. In other words, we use a binary **knowledge** metric on the main topic of the snippet and its **naturalness** for the participant as *independent variables*. A developer is considered *expert* (knowledge = 1) about the topic of a given snippet if its knowledge was greater than or equal to 0.5, while it is *non-expert* (knowledge = 0) otherwise. In other words, we did not consider *relative* knowledge that we used to select the snippets, but rather the *absolute* knowledge. Therefore, a participant could also be an *expert* about all the snippets she evaluated, if she had at least 0.5 of knowledge on all the topics. A snippet s is *natural* (naturalness = 1) for a developer d if s comes from the top 10% of the candidate snippets ordered by the developer-centric naturalness ($N_d(s)$); s is *unnatural* (naturalness = 0) if it comes from the bottom 10%.

To analyze the effect of the independent variables on the dependent ones, we run two types of statistical analyses. After performing a normality test (using the Shapiro-Wilk test), we found that our dataset does not follow a normal distribution. For this reason, we need to use non-parametric tests. First, we use the permutation test [49] to analyze the effect of knowledge and naturalness on, in turn, *AU* and *CD*. We also take into account the interaction between the independent variables. The permutation test is a non-parametric alternative to the n-way Analysis of Variance (ANOVA), and in our case can be used to analyze the effect of multiple independent variables on a dependent variable, as well as the effect of their interaction.

We also run an alternative analysis to further confirm our findings and to understand the magnitude of the difference. Specifically, we divide the evaluations collected into four groups, based on the variables *knowledge* and *naturalness* previously defined:

1. *expert–natural* (knowledge = 1, naturalness = 1);
2. *expert–unnatural* (knowledge = 1, naturalness = 0);
3. *non-expert–natural* (knowledge = 0, naturalness = 1); and
4. *non-expert–unnatural* (knowledge = 0, naturalness = 0).

We rely on the Wilcoxon rank-sum test [79] to check differences in *AU* and *CD* among the groups. The null hypothesis is that there is no significant difference in terms of *AU* and *CD* between pairs of groups. We do not compare groups containing evaluations with different values of both knowledge and naturalness (*e. g.*, *expert–natural* with *non-expert–unnatural*), since this would not allow us to understand the cause of the difference. We reject the null hypothesis if the p-value is lower than 0.05. We adjust the p-values using the Holm

method for multiple comparisons [30]. This method ranks n p-values in increasing order of value and multiplies the first one by n, the second by n-1, and so on. We also compute the effect size to quantify the magnitude of the significant differences we find. To this end, we report Cliff's delta [13], since it is suitable for non-parametric data. Cliff's d ranges in the interval [-1,1] and is *negligible* for $|d| < 0.148$, *small* for $0.148 \leqslant |d| < 0.33$, *medium* for $0.33 \leqslant |d| < 0.474$, and *large* for $|d| \geqslant 0.474$.

3.5 EMPIRICAL STUDY RESULTS

Before focusing on the results, we report demographic information about the study participants, *i.e.*, occupation, programming experience (years), and Java experience (years). Note that we only report such information for those who participated in both *P1* and *P2*. All the participants are students: 22 (62.86%) are Bachelor students, 11 (31.43%) are Master students and 2 (6.71%) are Ph.D. students. On average, the participants have ~3.6 years of programming experience and ~2.5 years of Java programming experience. 14 of them (40%) have more than 4 years of programming experience, and none of them has less than 3 years of programming experience.

As a first analysis, we perform a permutation test aimed at determining the effect of naturalness (developer-centric and global) on the understandability, and its interaction with the developers' topic knowledge. The results are reported in Table 8. Specifically, the table reports the results of the permutation test, *i.e.*, *p*-values for the effect of naturalness, of topic knowledge, and their interaction on *AU* and *CD*. The results are computed considering both the developer-centric naturalness (top) and global naturalness (bottom).

On the one hand, results show that, as expected, the topic knowledge of a developer has a statistically significant effect on the actual understandability (*AU*). It is, indeed, reasonable to think that the higher the level of knowledge on a topic, the higher the actual level of understanding that a developer can achieve. On the other hand, the permutation test shows that the deceptiveness (*CD*) is not affected by the topic knowledge. That is, such knowledge does not help developers in realizing which snippets they did not actually understand, and therefore to say it explicitly.

Comparing results in terms of the two dependent variables (*i.e.*, developer-centric vs. global naturalness), results indicate that, while the global naturalness does not have a significant effect on *AU* nor on *CD*, the developer-centric naturalness has a significant effect on the *CD*. For both the analyses related to global and developer-centric naturalness, there is no significant interaction with the topic knowledge.

Table 9 reports the comparison of the mean *AU* and *CD* between natural and unnatural snippets according to the global naturalness

Table 8: Effect of global and developer-centric naturalness, of topic knowledge, and of their interaction using two-way permutation test.

Topic knowledge and developer-centric naturalness.

Metric	Variable	D.f.	R Sum Sq.	R Mean Sq.	Iter.	p-value
CD	Developer-centric naturalness	1	0.455	0.455	500,000	**0.035**
	Topic knowledge	1	0.005	0.005	274,939	0.784
	Interaction	1	0.031	0.031	500,000	0.574
AU	Developer-centric naturalness	1	0.144	0.144	500,000	0.306
	Topic knowledge	1	1.383	1.383	500,000	**0.002**
	Interaction	1	0.054	0.054	500,000	0.528

Topic knowledge and global naturalness.

Metric	Variable	D.f.	R Sum Sq.	R Mean Sq.	Iter.	p-value
CD	Global naturalness	1	0.046	0.046	51	1.000
	Topic knowledge	1	0.005	0.005	51	1.000
	Interaction	1	0.176	0.176	500,000	0.185
AU	Global naturalness	1	0.140	0.140	500,000	0.314
	Topic knowledge	1	1.522	1.522	500,000	**0.001**
	Interaction	1	0.036	0.036	500,000	0.601

Table 9: Actual understandability and deceptiveness for global and developer-centric naturalness: a comparison using the Wilcoxon Rank Sum test.

Developer-centric Naturalness										
	Natural				**Unnatural**				**p-value**	**Cliff's d**
Measure	**Q1**	**Median**	**Mean**	**Q3**	**Q1**	**Median**	**Mean**	**Q3**		
AU	0.00%	0.00%	32.15%	0.67%	0.00%	0.00%	28.62%	0.67%	0.508	0.035 (negligible)
CD	0.00%	0.00%	18.35%	0.33%	0.00%	0.00%	25.42%	0.58%	**0.028**	−0.111 (negligible)
Global Naturalness										
	Natural				**Unnatural**				**p-value**	**Cliff's d**
Measure	**Q1**	**Median**	**Mean**	**Q3**	**Q1**	**Median**	**Mean**	**Q3**		
AU	0.00%	0.00%	29.71%	0.67%	0.00%	0.00%	30.97%	0.67%	0.713	−0.020 (negligible)
CD	0.00%	0.00%	20.29%	0.33%	0.00%	0.00%	23.27%	0.33%	0.390	−0.044 (negligible)

Table 10: Comparison on the subset with Wilcoxon rank-sum test for developer-centric naturalness.

	High knowledge vs. Low knowledge			
	Unnatural code for the evaluator		**Natural code for the evaluator**	
Measure	***p*-value**	**Cliff's *d***	***p*-value**	**Cliff's *d***
AU	**0.003**	**−0.221 (small)**	0.111	−0.121 (negligible)
CD	0.487	−0.052 (negligible)	0.568	0.040 (negligible)
	Natural *vs.* Unnatural			
	High knowledge evaluators		**Low knowledge evaluators**	
Measure	***p*-value**	**Cliff's *d***	***p*-value**	**Cliff's *d***
AU	0.897	−0.009 (negligible)	0.241	−0.091 (negligible)
CD	**0.019**	**0.159 (small)**	0.455	0.057 (negligible)

and the developer-centric naturalness respectively, along with the p-values of the Wilcoxon Rank Sum test and the Cliff's delta effect size. The test confirms that, as previously found, the difference of *AU* between the groups is not statistically significant, and the effect size negligible. Instead, we observed that, when using the developer-centric model to compare natural and unnatural code, the latter has a significantly higher deceptiveness (p-value $\simeq$ 0.028). When developers evaluate unnatural code, they are more prone to say that they understood the code when this was not true. On average, the difference in deceptiveness is ~7.1%. However, the magnitude of such a difference is negligible, even if the Cliff's delta is close to the 0.148 threshold for being "small". Finally, the Wilcoxon Rank Sum test also confirms that, when considering the global naturalness, the deceptiveness does not exhibit statistically significant differences between natural and unnatural source code.

The fact that unnatural code deceives developers could seem counter-intuitive: conceptually when developers read a snippet that is unfamiliar to them, they should be more prone to admit that they did not understand it. In such a case, the deceptiveness would be zero.

To deeper investigate such a phenomenon, we perform the same analysis keeping into account *knowledge* and developer-centric *naturalness* together. We report such a comparison in Table 10. The difference in terms of *AU* between *knowledgeable* and *non-knowledgeable* is large, especially for unnatural code (Cliff's d $\simeq$ -0.22). At the same time, the difference in terms of *CD* is particularly large between *natural* and *unnatural* code for developers with high topic knowledge (Cliff's d $\simeq$ 0.16).

RQ summary: Developer-centric naturalness significantly affects the deceptiveness of a snippet, above all for developers with high knowledge about the main topic of the snippet.

3.5.1 *Discussion and Implications*

The results show that naturalness (either global or developer-centric) does not directly affect the ability of developers to correctly answer questions about the code snippets. As expected, knowledge is a much more important factor to this end: a developer who knows well JDBC will more likely understand code that uses the JDBC APIs. Developer-centric naturalness, instead, significantly affects the deceptiveness of a snippet: developers are more prone to think that they understood a snippet whey they actually did not if the snippet is unnatural. We do not observe the same effect if we take into account global naturalness. This indicates that, differently from global naturalness, developer-centric naturalness can help to capture deceptiveness, which is an important understandability aspect. We also observed that this result is particularly evident for developers with high knowledge about the topic of the snippet. This result provides a possible explanation to the findings by Ray et al. [58]: unnatural code is likely to contain more defects because developers are deceived by it. In other words, developers may read unnatural code, decide that they understood it and start modifying it. If, however, they do not properly understood it as they believe, the consequence can be the introduction of bugs.

Figure 6 shows an example of a code snippet about servlets (some parts were removed for space and formatting reasons). An evaluator with high knowledge about Java servlets (0.8) said that she understood such a snippet. However, she achieved a AU of 0 (*i. e.*, she answered incorrectly to all the questions).

The obtained results, and above all the introduction of the developer-centric naturalness and its observed effects, can have noticeable implications for software engineering researchers, especially those developing recommending systems for software developers and, in general, tool support for software development.

For example, tools for code reviewer assignment or for issue triaging could take into account the developer-centric naturalness for the code a developer should review or modify and take this into account when performing the assignment of a reviewing or change task. Also, code reviewers or developers could be warned by a recommender to better pay attention in their task when an automated tool recognizes that the code could be highly deceptive for her. Moreover, the capability to estimate deceptiveness could open the road to developing better just-in-time defect prediction approaches [32, 42, 53], because high deceptiveness (on developer-centric unnatural code) could increase the likelihood to introduce a bug.

Last, but not least, the results of our study open the road for deeper analyses of the origin of deceptiveness when understanding source code. In other words, it is worthwhile to investigate what the intrinsic characteristics of a source code snippet (*e. g.*, the combination of

Figure 6: Example of deceptive code.

```
public RequestDispatcher getRequestDispatcher(String path) {
  final String newRequestUri = path;
  [...]
  for (final ServletDefinition servletDefinition :
    servletDefinitions) {
    if (servletDefinition.shouldServe(path)) {
      return new RequestDispatcher() {
        @Override
        public void forward(ServletRequest servletRequest,
    ServletResponse servletResponse) throws ServletException,
    IOException {
          Preconditions.checkState(
            !servletResponse.isCommitted(),
            "Response has been committed--you can"
            + "only call forward before committing"
            + "the response");

          // clear buffer before forwarding
          servletResponse.resetBuffer();

          ServletRequest requestToProcess;
          if (servletRequest instanceof HttpServletRequest)
          {
            requestToProcess = wrapRequest(
              (HttpServletRequest) servletRequest,
              newRequestUri
            );
          } else {
            // This should never happen, but instead of
            // throwing an exception we will allow a happy
            // case pass thru for maximum tolerance to
            // legacy (/internal) code.
            requestToProcess = servletRequest;
          }
        }
        [...]
      };
    }
  }

  //otherwise, can't process
  return null;
}
```

source code patterns/idioms, choice of identifiers, API usages) that provoke high deceptiveness are.

Ultimately, once the phenomenon has been better understood, this will also impact educators, which could better teach students (i) how to write source code in such a way to minimize deceptiveness for somebody else who has to review or change it; and (ii) how to better review source code when its characteristics exhibit, for a given developer, symptoms of high deceptiveness.

3.6 THREATS TO VALIDITY

Threats to construct validity concern the relation between theory and observation. In this case, they are mainly due to the process adopted to build our developer-centric model. A better approach would have been to use the source code previously written by the participants as a training set. Unfortunately, we did not have access to such code for all the developers. We approximated such a code-base by training the model using code snippets from open-source projects, as detailed in Section 3.4.

Another possible threat is due to the questions we asked to evaluate the developers' understanding — *i. e.*, they could be too easy or difficult. To mitigate this problem, in the first phase (*P1*), we defined the questions based on the Oracle tutorials. Specifically, we defined questions that covered all the subtopics of each topic. For example, concerning the topic collections, we considered questions on all data structure (*e. g.*, `ArrayList` and `HashMap`). As for the second phase (*P2*), three of the authors wrote the questions (each question was written by one of them). Each question was double-checked by all the other authors: they could give feedback about the clarity of the question (*e. g.*, when the found an ambiguous question), the soundness of the possible answers (*e. g.*, more than a correct answers) and the difficulty (*e. g.*, the question is too easy or one of the answers is tricky).

Threats to internal validity concern factors internal to our study that we ignored and that could have influenced the investigated relations. We observed that developer-centric naturalness has an effect on understandability. However, it is possible that other factors, such as the experience, was an indirect cause of such a result. To mitigate this threat, we considered the topic knowledge in our model (Section 3.5), and we observed no significant effect of such a factor alone on code understandability. Also, another possible threat could be that the participants previously worked on the code they had to understand. To mitigate this threat, we checked that none of the participants was a contributor to such projects.

Threats to conclusion validity concern the relationship between theory and outcome and are essentially due to the appropriate usage of statistical procedures in order to take our conclusions. As also ex-

plained in Section 3.4, due to the nature of the analyzed data (not following a normal distribution), we rely on non-parametric statistics (Wilcoxon Rank Sum test, Cliff's delta effect size, Permutation test, and Kendall's τ correlation). Moreover, when multiple comparisons are performed, we properly adjust *p*-values using Holm's correction.

Threats to external validity concern the generalizability of our results. Our study is focused on the Java programming language. Our results could be not generalizable to other languages. Besides, we involved in our study a total of 35 developers using convenience sampling: the results may be limited to such a sample. Replications of such a study may be necessary to ensure the generalizability of the results. To foster the replicability of our study, we publicly release a replication package with our scripts and data.

3.7 CONCLUSION

Code understandability is important for maintenance-related activities. However, no state of the art metric is able to predict such an aspect [65]. On the other hand, previous work showed that unnatural code is more defect-prone [58]: this is an indication of the fact that there could be a relationship between understandability and naturalness. In this chapter, we investigated this relationship. We first conducted a study in which we show that a global naturalness model is not able to predict code understandability. Then, we defined a novel developer-centric model, which is based only on code about topics on which the developer has experience.

To check the existence of such a relationship, we conducted an empirical study with 52 Java developers, in which we asked them to understand both natural and unnatural snippets. Our results show that developer-centric naturalness is significantly associated with a higher deceptiveness of the source code: unnatural code makes developers believe that they understood the snippet (*perceived understandability*, *i.e.*, the one declared by them), while they do not (*actual understandability*, *i.e.*, the number of correct answers to the questions). The Dunning-Kruger effect [21] may play a role in this: developers with small expertise in a topic may misjudge their ability to understand code regarding such a topic.

Our results allow us to provide a possible explanation for the introduction of some bugs: when developers are faced with unnatural code, they are wrongly overconfident (which is shown by the results of our study); as a result, there is a higher risk that they introduce bugs (which was shown by Ray et al. [58]). This opens the road towards further studies aimed at better understand the reasons for deceptiveness during code comprehension, as well as to the application of the developer-centric naturalness in recommender systems aimed

at supporting a number of software engineering tasks, such as code review, change request triaging, or defect prediction.

4

ON THE RELATIONSHIP BETWEEN API QUALITY AND THE SOFTWARE FAILURE PRONENESS

4.1 INTRODUCTION

For a developer, the understanding of Application Programming Interfaces (APIs) is essential for their proper use. The best way to learn how it works and how to use an API is going through the documentation. The documentation of an API can be seen as a technical manual that contains all the information needed to work with the API. In order to be understood by API users, the documentation must be easy to read, it must be clear and it must contain the right balance between background information, useful to understand the context, and that which is tightly needed. Documentation with a shallow description of the function's behavior, lacking an exhaustive explanation of the parameters and/or the return values, or with no practical code examples of API usage can negatively affect developers, discouraging them from integrating the API into their code.

However, API developers do not always dispose of the necessary time or resources to invest in documentation, which is often inadequate or even missing. In this case the risk is that the API is used incorrectly by the developer, who, not having full knowledge of how to use the API, is more likely to introduce bugs into the code. Our assumption is that the introduction of bugs by developers may be related to the use of low quality APIs and that this impact is higher when the API is *first used*.

To investigate the quality of the API, we analyzed 805 Java methods, paying particular attention to the information content in the documentation. We used a web platform which was implemented to simplify the documentation analysis process and to automate the selection of APIs assigned to evaluators. For each method we identified the overall quality of the documentation and the quality of the individual parts (paragraphs). This process of analysis involved 6 researchers and took about 2 months. The analysis allowed us to create a taxonomy of the information contained in the API documentation, but most importantly to investigate the relationship between the quality of the documentation and the introduction of bugs when the API was first used.

Our assumption is that APIs with low quality documentation may lead the developer to introduce bugs in the software code, in particular when the API is first used. We used the SZZ algorithm [68] to analyze the history of 201 GitHub projects, which use at least one

of the analyzed methods, in order to investigate the relationship between the first use of an API and the presence of bugs in the source code.

The results confirm our initial hypothesis, in fact we showed that APIs with lower documentation are more likely to introduce bugs when used for the first time within a software project.

4.2 EMPIRICAL STUDY DESIGN

The *goal* of this study is to investigate the relationship between the quality of API documentation and the proneness to introduce bugs into the code, in particular to what extent this happens when the API is used for the first time within a software project. The study *context* consists of documentation for 805 Java methods that were manually analyzed and assessed for information content and quality, and 201 GitHub projects using at least one of the investigated methods.

4.2.1 *Research Questions*

The first purpose of the study is to understand what information is contained within the API documentation. For this reason we formulate the following research question:

> **RQ_1:** *What are the elements composing the documentation of Java APIs?*

The RQ_1 answer produced the creation of a taxonomy of knowledge types detailed in Section 4.3.1. At the same time as analyzing the type of information contained in an API, we analyzed the quality of the components and the overall quality of the method documentation, and we managed to answer the second research question:

> **RQ_2:** *What is the quality of API documentation, and of its elements?*

After answering the first two research questions closely related to the characteristics of the documentation, we tried to understand to what extent the use of APIs with different quality ratings relates to the introduction of bugs in the code. We then pose the third search question:

> **RQ_3:** *Does a low API documentation quality relate with higher defect-proneness?*

To answer the RQ_3 we considered the history of 201 Java projects on GitHub employing at least one of the methods previously analyzed and we used the SZZ algorithm to look for a link between the introduction of bugs and the use of low quality APIs.

4.2.2 *Analysis Methodology*

The first step for data collection was the selection of Java libraries from Maven Central Repository[1]. We randomly chose two libraries for each category in the repository, for a total of 290 libraries and then we implemented a tool to extract all public methods with their javadoc. The Table 11 shows the categories covered by the analysis and reports the number of analyzed methods documentation and the total number of downloaded methods for each category.

Table 11: Number of analyzed methods (#M) and total number of methods (#T) for Maven categories.

MVN category	#M/#T	MVN category	#M/#T
Aspect Oriented	4/20	Language Runtime	6/20
Assertion Libraries	12/40	Math Libraries	5/20
Bytecode Libraries	6/20	Message Queue Clients	5/20
CSS, LESS, SASS	11/20	Microbenchmarks	7/19
Cache Clients	5/20	MySQL Drivers	10/20
Cache Implementations	11/18	Native Access Tools	4/19
Cassandra Clients	15/38	OSGI Utilities	5/20
Chart Libraries	2/20	Object Serialization	5/19
Collections	4/15	Off-Heap Libraries	4/18
Compression Libraries	6/20	PostgreSQL Drivers	8/20
Console Utilities	11/20	RDF Libraries	4/20
Core Utilities	4/20	REST Framework	5/20
DB Migration Tools	8/20	Rule Engines	9/38
Distributed Computing	8/20	SSH Libraries	5/20
Distributed Coordination	8/20	Search Engines	10/20
Embedded SQL Databases	13/40	Security Frameworks	3/20
Expression Languages	9/20	Social Network Clients	9/40
Functional Programming	11/20	Swing Layouts	5/20
Graph Databases	9/19	Swing Libraries	11/40
HBase Clients	8/20	Testing Frameworks	8/20
HTML Parsers	9/40	Web Frameworks	14/40
I18N Libraries	7/20	Web Servers	14/40
JDBC Extensions	6/20	Web Testing	7/20
JPA Implementations	4/20	WebSocket Clients	5/18
Java Compilers/Parsers	7/20	XMPP Integration Libraries	8/20
JavaScript Processors	2/20		

The selected Java methods were manually analyzed with the goal of evaluating both the overall documentation quality of a method and the quality of the individual parts (paragraph level). The overall quality and the quality at paragraph level was assigned by a scale

1 https://mvnrepository.com

of values ranging from 1 (very low) to 5 (very high). Moreover, starting from the taxonomy of knowledge types proposed by Maalej and Robillard [38], for each paragraph we assigned one or more types of knowledge and we identified any information patterns (*e.g.*, `if [condition] returns [something]`).

The labeling process was supported by a web application implemented *ad hoc*. The web application displayed the documentation and source code of the methods, which were randomly selected before being submitted to two evaluators. After highlighting individual paragraphs, the evaluators had to assign a quality assessment, identify the type of information contained in the paragraph, choose between existing labels or create new ones, and assign an assessment to the quality of the paragraph. Finally, after analyzing all paragraphs, the evaluators made an assessment of the overall quality of the method documentation. The platform not only randomly assigned the APIs to be evaluated, but also ensured that each method was evaluated by at least two evaluators.

At the end of the labelling process, data from 805 methods were collected, each of which was analyzed by two random evaluators. At this point we proceeded with the conflict resolution phase. The web application highlighted cases where the two evaluators had a conflicting opinion about the type of information contained in the paragraphs. These cases were submitted to a third evaluator who could either confirm the decision of one of the two evaluators or reject both of them and re-label the paragraph. In addition, the two evaluators resolved conflicts in quality assessment by manually analyzing all cases where the quality assessment differed by at least two score points, while in other cases the conflict resolution process was automated by averaging the scores assigned to the documentation. Finally two evaluators came together to refine the new taxonomy of knowledge types identified in the individual paragraphs, merging similar categories, splitting those that were too general or specifying others.

After collecting information and analyzing the quality of 805 Java APIs methods, the purpose was to identify open source projects on GitHub that use those methods within their code. To achieve our goal we used the GitHub search APIs to search for files containing the API and method name. A representative example of our search was: `"import org.springframework.core.TypeDescriptor" "getType"`.

The resulting files were sorted by the best match criteria so that at the top of the GitHub search APIs output list were the files containing both search keys. We took the first 1020 projects resulting from the query and then we eliminated the false positives. In this way we collected more than 239k API uses belonging to more than 51k repositories, which covers 662 methods out of 805. Finally, we checked the matching between versions of the libraries, identifying 2098 APIs uses (out of 239k) belonging to 920 repositories and covering 79 of

the 805 methods analyzed during the previous phase. From the list of 920 repositories we eliminated all those with less than 100 commits and less than 3 contributors, obtaining the final number of 201 repositories approved for analysis.

In order to identify the commits that introduced a bug, we cloned the 201 repositories from GitHub and for each project we produced a log file in the following format:

```
FileName,GitID,TimeStamp,Author,Change-Type,CommitMessage
```

Since we were unable to know if the 201 projects used an issue tracking system and we did not know what it was, we decided to adopt a very simple approach to identify potential fix commits. We exploited the content of the `CommitMessage` attribute to establish whether a commit was a fix or not. More specifically, we considered fix commits all those commits that contained the keyword *fix* within the commit message. We then used the SZZ algorithm, implemented based on Bavota et al. [8], to identify the commits that introduced the bugs. The SZZ implementation first identifies the lines changed by the fix (excluding cosmetic changes and changes to commented lines). Then, starting from the file version before the fix, and considering only the fixed lines, we use `"git blame -w -p"` to identify the last change before the fix to these lines, along with the file name, and line number mapping. In summary, for each non-comment changed line of fixed files, the algorithm outputs a candidate introduction location (commit, file name and line number).

4.3 RESULTS

In this section we present and discuss results aimed at addressing the research questions formulated in Section 4.2.

4.3.1 *What are the elements composing the documentation of Java APIs?*

Figure 7 shows the first contribution of this work: the creation of a knowledge types taxonomy of API documentation. For the taxonomy definition we started from the knowledge types taxonomy proposed by Maalej and Robillard [38]. Examining the elements that compose the documentation of the 805 methods employed in our study, we realized that the taxonomy categories defined by Maalej and Robillard [38] were not enough. Out of this observation emerged the necessity to define 10 new categories but, above all, the need to make them more specific in order to delineate the various API documentation elements. The proposed taxonomy can be read from left to right. On the left we find the 21 main categories (colored in blue) to which are linked, to the right, the 67 subcategories (colored in blue, green and yellow).

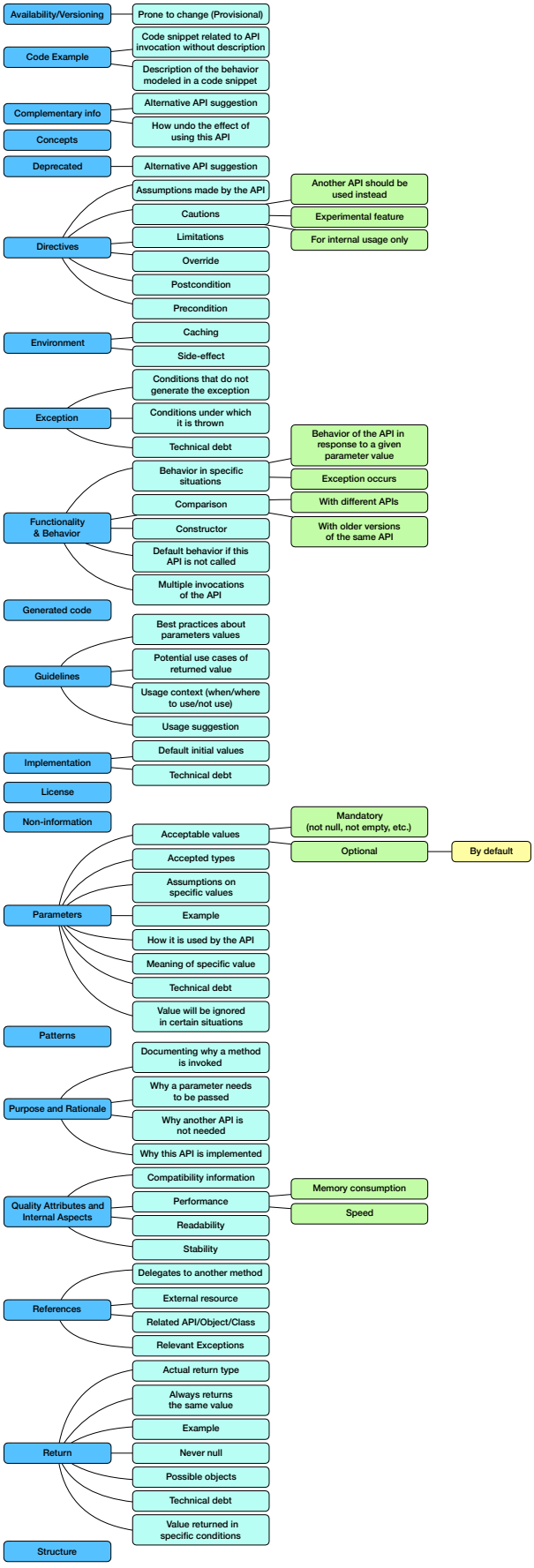

Figure 7: Taxonomy of documentation knowledge types.

For clarity we provide a complete list of the 21 main categories with their description, divided into categories which perfectly fit those proposed by Maalej and Robillard [38], existing categories in [38] modified to adapt them to our taxonomy, and new categories arising from our empirical study.

4.3.1.1 *Categories which perfectly fit the Maalej and Robillard taxonomy*

Concepts. Explains the meaning of terms used to name or describe an API element, or describes design or domain concepts used or implemented by the API.

Non-information. A section of documentation containing any complete sentence or self-contained fragment of text that provides only uninformative boilerplate text.

Patterns. Describes how to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how the behavior of an element can be customized, etc.

Structure. Describes the internal organization of a compound element (e.g. important classes, fields, or methods), information about type hierarchies, or how elements are related to each other.

4.3.1.2 *Evolution of Maalej and Robillard taxonomy categories*

Code Example. Provides code examples of how to use and combine elements to implement certain functionality or design outcomes. We specialized the category in two subcategories: code snippet related to API invocation without description, and description of the behavior modeled in a code snippet.

Directives. Specifies what users are allowed/not allowed to do with the API element. Directives are clear contracts. In defining our taxonomy, we focused on six aspects relating to directives: assumptions made by the API, cautions, limitations, directives in case of method override, preconditions, and postconditions.

Environment. Describes aspects related to the environment in which the API is used. The difference from the taxonomy baseline is the lack of compatibility issues, differences between versions, and licensing information, which are placed in specially created categories such as *Availability/Versioning*, and *License*, or in subcategories such as *Comparison with older version of the same API* and *Compatibility information*.

Functionality & Behavior. Describes what the API does (or does not do) in terms of functionality or features. Describes what happens when the API is used. Our category differs from the baseline taxonomy in that elements describing API calls to other methods do not fit into this category, but are labeled as *Implementation*.

Purpose and Rationale. Explains the purpose of providing an element or the rationale of a certain design decision. This is information that answers a "why" question. Specifically, we include in this category the documentation elements that answer the questions: why a method is invoked?, why a parameter needs to be passed?, why another API is not needed?, and why this API is implemented?

Quality Attributes and Internal Aspects. Contains all the elements that discuss API performance, readability, stability, and compatibility information.

References. Includes any pointer to other sources of information. It does not only refer to external resources, but includes cases where the API delegates execution to another method, adds a reference to other APIs, objects, classes, or exceptions.

4.3.1.3 *New categories*

Availability/Versioning. Information about availability of an API or information about API changes from the last version.

Complementary information. Any additional information which is not directly related to the API, but is useful for who use the API.

Deprecated. Notify that the API is deprecated or describes the reason why API is deprecated.

Exception. Describes the possible conditions under which the exception is thrown by the API and/or the conditions that do not generate the exception.

Generated code. Refers to portions of automatically generated code.

Guidelines. Explains how to use the API in specific situations, in which contexts it is suggested to use/not to use the API and best practices.

Implementation. Information about how the API is implemented and what methods are called within the API.

License. Add information related to license(s).

Parameters. Describes about parameter(s), how the API uses them, the meaning of specific values, examples of possible parameter values, etc.

Return. Add information about what is returned by the method.

4.3.2 *What is the quality of API documentation, and of its elements?*

Table 12 shows the overall quality distribution of the analyzed documentation on a scale from 1 (low quality) to 5 (high quality), in half-point increments. The results reveal that in 192 cases the quality of the documentation obtains the average value of 3. Although in only

Table 12: Number of API documentation (#N) for different overall documentation rates (API rate).

API rate	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	**Tot**
#N	28	31	66	116	192	177	145	40	10	805

10 cases the quality reaches the maximum value, considering all the occurrences in which the score exceeds the median value, it results that in 46% of the cases the observed quality is medium-high, while in 30% of cases the API documentation quality is medium-low.

Table 13 lists the documentation element types, showing the number of times an evaluator identified an element belonging to that category and a set of statistical data related to quality, such as minimum, maximum, average and median. The first data that stands out concerns the massive presence of elements regarding the description of parameters, 3328 out of 8497 total elements. It is also possible to notice that the 3 most frequent categories, *Functionality & Behavior*, *Return*, and *Parameters* cover 75% of the elements identified. This data proves that developers tend to prioritize these three pieces of information when documenting an API, But there is another reason that should not be overlooked: the automatic creation of javadoc tags (e.g. `@param`). In fact, looking more closely at the table we notice that 4 of the 5 most populated categories are those containing the most common block tags, such as `@param`, `@return`, `@throws`, and `@see`.

From the quality point of view it has been observed that *Code Example*, *Guidelines*, and *Purpose & Rationale* get the highest scores, averaging 3.86, 3.82, and 3.81, respectively. The result means that developers take special care in describing these elements, contributing to the overall improvement of API quality.

4.3.3 *Does a low API documentation quality relate with higher defect-proneness?*

Before answering the RQ_3 we did the Shapiro-Wilk test on our data, in particular we tested `quality_overall` and `quality_average` for the purpose of studying its distribution. The obtained results are shown below:

```
Shapiro-Wilk normality test

data:  quality_overall
W = 0.86507, p-value < 2.2e-16

data:  quality_average
W = 0.81588, p-value < 2.2e-16
```

Table 13: Documentation Element Types with number of occurrences (#N) and quality statistics.

Documentation Element Type	#N	Quality			
		min	max	avg	med
Availability/Versioning	126	2	5	3.10	3
Code Example	79	2	5	3.86	4
Complementary Information	183	1	5	3.63	4
Concepts	20	3	5	3.60	4
Deprecated	46	1	4	3.28	3
Directives	116	1	5	3.49	4
Environment	11	3	4	3.64	4
Exception	636	1	5	3.23	3
Functionality & Behavior	1848	1	5	3.31	3
Generated code	1	3	3	3.00	3
Guidelines	141	2	5	3.82	4
Implementation	136	2	5	3.64	4
License	2	3	4	3.50	3.5
Non-information	43	1	3	2.33	3
Parameters	3328	1	5	3.07	3
Patterns	2	4	4	4.00	4
Purpose & Rationale	57	2	5	3.81	4
Quality Attr. & Internal Aspects	51	2	5	3.63	4
References	454	1	5	3.12	3
Return	1216	1	5	3.16	3
Structure	1	2	2	2.00	2

Figure 8 is the Quantile-Quantile plot produced by the functions `qqnorm()` and `qqline()` available in *R*. Both analyses reveal that the distribution of the data is non-normal.

For this reason we decided to use a non-parametric test for comparing independent samples. First of all we decided to investigate the quality aspects of API documentation by dividing them into APIs that contributed to the introduction of bugs coincided with their first use and all the others. This analysis was carried out both from the point of view of overall documentation quality and considering the documentation components average scores. As shown in the left frame of the Figure 9, the overall quality of API documentation that is not part of fix-inducing changes is averaged at 2.91, on the contrary, the overall quality of APIs considered responsible for causing the introduction of bugs is 2.30. Even more interesting are the values of the first and third quartile and the median, in fact while the first data val-

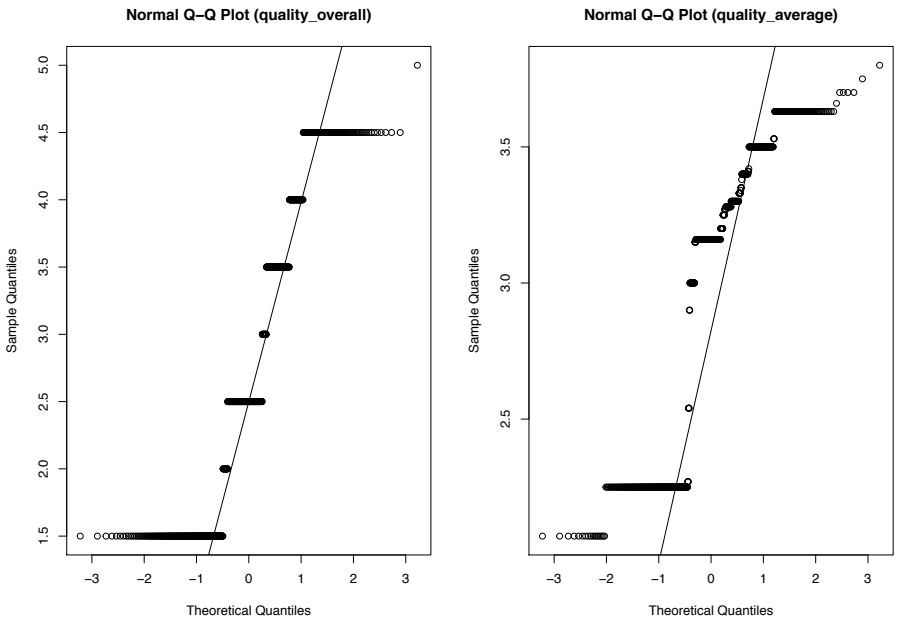

Figure 8: Normal Q-Q Plots of overall API quality and average API elements quality.

ues are between 2.5 and 3.5 and have a median of 2.5, the second are between 1.5 and 2.5 with a median of 1.5. Later we used the Wilcoxon Rank Sum test [79] and Cliff's delta (d) effect size [25] to compare the data distribution and to verify if there was a statistically significant difference between the distributions. This hypothesis was confirmed by the test, as shown by the following results:

```
--- Wilcoxon rank sum test with continuity correction ---
data:  quality_overall by is_bug
W = 84974, p-value = 1.752e-15
alternative hypothesis:
true location shift is greater than 0

--- Cliff's Delta ---
delta estimate: 0.3501438 (medium)
95 percent confidence interval:
      inf       sup
0.2582226 0.4357820
```

We also obtained similar results with the quality data of the elements that make up the documentation of an API. In this case, the API quality average is the average of the quality scores assigned to its component elements. The two boxplots in the right frame of the Figure 9 show the quality comparison between APIs not responsible (`is_buggy_commit = 0`) and responsible (`is_buggy_commit = 1`) for introducing bugs during their first use. In the first case the first and

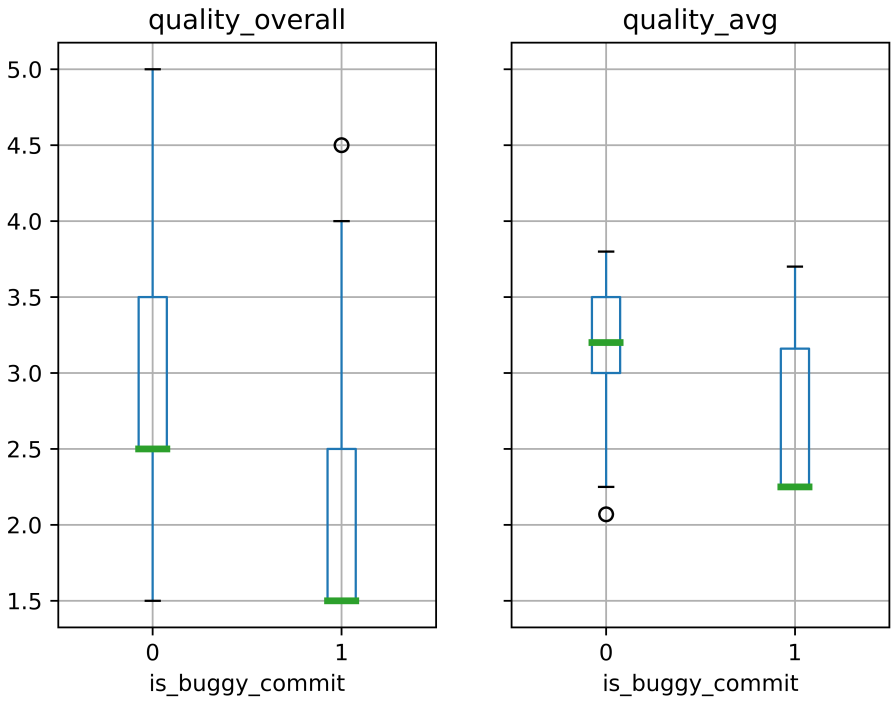

Figure 9: Boxplot of API quality.

third quartile data vary between 3.00 and 3.50, while in the second case they vary between 2.25 and 3.07. Regarding the median, in the first case we have a value of 3.20, while in the second we have a value of 2.25. Again, there is a significant statistical difference according to the Wilcoxon Rank Sum test and the effect size is medium.

```
--- Wilcoxon rank sum test with continuity correction ---
data:  quality_average by is_bug
W = 84159, p-value = 2.815e-14
alternative hypothesis:
true location shift is greater than 0

--- Cliff's Delta ---
delta estimate: 0.3371943 (medium)
95 percent confidence interval:
      inf       sup
0.2469371 0.4216513
```

In conclusion, findings suggest that commits containing the introduction of APIs having low quality documentation induce more bugs than commits where the introduced APIs have higher quality documentation.

4.4 THREATS TO VALIDITY

Construct validity. The main threat concerns the data obtained as a result of the SZZ algorithm. As explained by Da Costa et al. [19] the SZZ algorithm can be imprecise for several reasons, e.g., bulk changes, or considering as fix-inducing changes the first commit in the program history. We mitigated this threat by, at least, excluding the first commit in the history of each source code file, considering, instead, bulk changes as still valid, potential responsible of fix inducing changes. A second threat to construct validity is the subjectivity in the evaluation of API documentation. As explained in Section 4.2.2, to minimize the level of subjectivity we used two evaluators for the analysis of each API, and in cases where the evaluators disagreed, a third evaluator was in charge of resolving the conflicts.

Internal validity. We are aware that there is no cause-effect relationship between the low quality API documentation and the bug introduction. The introduction of defects in the code may depend on a plethora of other factors such as software complexity, partial understanding or misinterpretation of software requirements, delivery time pressure, lack of efficient testing tools or "simply" developer distraction. For this reason, we don't make any claims about the cause-effect relationship in our work.

External validity. We are aware that our work, to be generalized, must extend (i) the number of analyzed API documentation and (ii) the number of GitHub projects that use these APIs.

4.5 RELATED WORK

The importance of APIs documentation and its impact on software life has inspired a lot of research. There are many studies on whether and to what extent the information on the web could enrich and improve the quality of APIs documentation. Robillard and Chhetri [61] developed a tool called *Krec* that could detect and recommend API documentation fragments relevant to the developer. In particular the tool is able to highlight indispensable and valid information. Parnin and Treude [51] studied the extent to which methods of a particular API are documented on the Web, measuring the effectiveness and completeness of crowdsourced documentation. Their analysis showed the importance of social media information about API documentation. Treude and Robillard [72] use the information on the web, in particular the meta data available on Stack Overflow, to improve the official APIs documentation. Petrosyan, Robillard, and De Mori [54] proposed an approach based on textual classification techniques able to identify the sections of the tutorial where it is explained how to use a given API type. They analyzed five tutorials for Java APIs achieving a precision between 0.74 and 0.94. Parnin et al. [52]

made an empirical study on coverage, dynamics, and quality of Stack Overflow crowd documentation and proposed an approach that uses API discussions to automatically generate API documentation. Robillard et al. [62] proposed a new way of producing documentation, the On-Demand Developer Documentation (OD3). According to their vision, the OD3 should answer the developer's queries by generating high-quality documentation.

4.6 CONCLUSION

The quality of API documentation plays a key role in understanding what the API does and, most importantly, in integrating it into the code without introducing bugs. However, factors such as lack of time or developer negligence affect the quality of the API documentation. In this chapter we analyze the documentation of 805 methods belonging to Java APIs. The analysis was performed thanks to the implementation of a web application responsible for checking that each document was analyzed by at least two evaluators. The evaluators identified the different knowledge types, assigned them an assessment of the quality and evaluated the overall quality of the documentation.

The first contribution of the work was the creation of a taxonomy of knowledge types within the API documentation. We then analyzed the quality of the documentation and its elements, observing that in 46% of the cases the overall quality of the documentation was medium-high, while the elements with the highest scores were *Code Example*, *Guidelines*, and *Purpose & Rationale*. Finally, we used the SZZ algorithm to check whether the first use of APIs with low quality documentation would affect the introduction of bugs. During this analysis, we investigated the history of 201 GitHub projects. The results showed a relationship between the low quality API documentation and the introduction of bugs.

Although manual analysis of the quality of API documentation takes a lot of effort and time, to be sure of the validity of our results, we should extend the study to a larger number of APIs and projects that exploit them. In addition, in the future we plan to include in the analysis not only the API documentation, but also external information, such as information contained within Stack Overflow discussions.

5 CONCLUSION

The purpose of this thesis is to summarize the research works carried out over the three years of my PhD, during which I had the opportunity to engage in various research activities, facing different issues and problems, from defects prediction to the analysis of API documentation. Research works presented in this thesis are those that embrace a common thread, that of crowdsourced documentation and code snippets. In fact, in Chapter 2 we analyze the characteristics of Stack Overflow answers containing code snippets that are exploited within GitHub projects, in Chapter 3 we propose an approach for the creation of naturalness models based on developers' knowledge in order to predict the understandability of code snippets, finally, in Chapter 4 we investigate the relationship between the quality of Java API documentation and the introduction of bugs in the code.

Specifically, we posed six research questions. First, we investigated the characteristics of Stack Overflow answers containing one or more code snippets that are leveraged within the GitHub project code. Then, the first research question is:

> *Which are the characteristics of SO answers that have been leveraged by developers?*

We studied 22 factors, grouped into three categories: community factors, code quality factors, and text readability factors. We address this research question by statistically comparing the value distributions of these factors between leveraged and non-leveraged answers. Results highlight that community factors play a role, in particular the answer score, the number of comments and the creation date exhibit significant and large/medium differences between leveraged and non-leveraged posts. Another interesting result shows that developers tend to leverage longer and more complex code snippets. Based on the results, we asked ourselves whether it would be useful to use recommender systems to identify posts that are likely to be leveraged by developers. Therefore, our second research question:

> *Which is the performance of a recommender system in identifying posts that are likely to be leveraged by developers?*

We developed 12 classifiers, combining four machine learning techniques with three data balancing techniques. The classifiers used a subset of the factors analyzed in the RQ_1 as independent variables and the categorical variable leveraged/non-leveraged as dependent variable. Our results showed that the classifier that achieved the best

results was the Random Forest without data balance, but the most interesting finding was the evidence of how the recency of data affected the accuracy of the classifiers.

Staying in the field of code snippets, another aspect that motivated our research was the understandability of code. Specifically we investigated the relationship between developer-centric naturalness and source code understandability, by making the following research question:

> *How does the understandability vary for source code snippets exhibiting different levels of developer-centric naturalness?*

To answer the RQ_3 we assessed the experience of 52 developers on eight Java topics, and we used this information to build developer-centric naturalness models. We employed these models to select a list of *naturalunnatural* code snippets and submitted them to the developers to verify their actual understanding of the code. The result of the analysis was astounding, in fact when developers are faced with unnatural code snippets more often they perceived they perceive to understand the code while, in reality, they did not.

Regarding crowdsourced documentation, we investigated the documentation quality of 805 Java methods in order to: (i) identify the elements that make up the documentation, (ii) analyze the quality of the elements and the overall quality of the documentation, and (iii) investigate the relationship between the first use of low quality APIs and the introduction of bugs in the code. Our last research question can be summarized with the following one:

> *What is the quality of the Java API documentation and how does it relate with the introduction of defects in the code?*

Our empirical study showed that in 46% of cases the documentation quality is between 3.5 and 5, while in 30% of cases it is between 1 and 2.5. The most important result, however, relates to the introduction of defects. In fact, the results show that APIs with lower quality documentation are the ones most involved in the introduction of bugs.

BIBLIOGRAPHY

[1] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. "On code reuse from stackoverflow: An exploratory study on android apps". In: *Information and Software Technology* 88 (2017), pp. 148–158.

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. "You get where you're looking for: The impact of information sources on code security". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 289–305.

[3] Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, and Michele Lanza. "Automated Documentation of Android Apps". In: *IEEE Transactions on Software Engineering* (2019).

[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. "Learning natural coding conventions". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 281–293.

[5] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. "Stack overflow: a code laundering platform?" In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2017, pp. 283–293.

[6] Sebastian Baltes and Stephan Diehl. "Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects". In: *arXiv preprint arXiv:1802.02938* (2018).

[7] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. "SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts". In: *arXiv preprint arXiv:1803.07311* (2018).

[8] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. "When does a refactoring induce bugs? an empirical study". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2012, pp. 104–113.

[9] Yoav Benjamini and Yosef Hochberg. "Controlling the false discovery rate: a practical and powerful approach to multiple testing". In: *Journal of the Royal statistical society: series B (Methodological)* 57.1 (1995), pp. 289–300.

[10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. "Example-centric programming: integrating web search into the development environment". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2010, pp. 513–522.

[11] Raymond PL Buse and Westley R Weimer. "Learning a metric for code readability". In: *IEEE Transactions on Software Engineering* 36.4 (2009), pp. 546–558.

[12] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. "Evolution of understandability in OSS projects". In: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE. 2004, pp. 58–66.

[13] Norman Cliff. "Dominance statistics: Ordinal analyses to answer ordinal questions." In: *Psychological bulletin* 114.3 (1993), p. 494.

[14] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[15] Meri Coleman and Ta Lin Liau. "A computer readability formula designed for machine scoring." In: *Journal of Applied Psychology* 60.2 (1975), p. 283.

[16] Michael L Collard, Huzefa H Kagdi, and Jonathan I Maletic. "An XML-based lightweight C++ fact extractor". In: *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE. 2003, pp. 134–143.

[17] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. "Context-based recommendation to support problem solving in software development". In: *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE. 2012, pp. 85–89.

[18] Denzil Correa and Ashish Sureka. "Chaff from the wheat: characterization and modeling of deleted questions on stack overflow". In: *Proceedings of the 23rd international conference on World wide web*. ACM. 2014, pp. 631–642.

[19] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. "A framework for evaluating the results of the szz approach for identifying bug-introducing changes". In: *IEEE Transactions on Software Engineering* 43.7 (2016), pp. 641–657.

[20] Masayuki Doi, Yoshiki Higo, Ryo Arima, Kento Shimonaka, and Shinji Kusumoto. "On the naturalness of auto-generated code: can we identify auto-generated code automatically?" In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, pp. 340–343.

[21] David Dunning. "The Dunning–Kruger effect: On being ignorant of one's own ignorance". In: *Advances in experimental social psychology*. Vol. 44. Elsevier, 2011, pp. 247–296.

[22] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. "Stack overflow considered harmful? the impact of copy&paste on android application security". In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 121–136.

[23] Rudolph Flesch. "A new readability yardstick." In: *Journal of applied psychology* 32.3 (1948), p. 221.

[24] Mark Gabel and Zhendong Su. "A study of the uniqueness of source code". In: *Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2010, pp. 147–156.

[25] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[26] R. Gunning. *The technique of clear writing*. McGraw-Hill, 1968. URL: https://books.google.ch/books?id=vJZpAAAAMAAJ.

[27] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[28] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. "Will they like this? evaluating code contributions with language models". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 157–167.

[29] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. "On the naturalness of software". In: *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE. 2012, pp. 837–847.

[30] Sture Holm. "A simple sequentially rejective multiple test procedure". In: *Scandinavian journal of statistics* (1979), pp. 65–70.

[31] Matthieu Jimenez, Thiery Titcheu Checkam, Maxime Cordy, Mike Papadakis, Marinos Kintis, and Mark Traon Yves Le and Harman. "Are mutants really natural? a study on how "naturalness" helps mutant selection". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2018, pp. 1–10.

[32] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. "Studying just-in-time defect prediction using cross-project models". In: *Empirical Software Engineering* 21.5 (2016), pp. 2072–2106.

[33] Maurice G Kendall. "A new measure of rank correlation". In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[34] J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. "Derivation of new readability formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for navy enlisted personnel". In: (1975).

[35] Bin Lin, Luca Ponzanelli, Andrea Mocci, Gabriele Bavota, and Michele Lanza. "On the uniqueness of code redundancies". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE. 2017, pp. 121–131.

[36] Jin-Cherng Lin and Kuo-Chiang Wu. "A model for measuring software understandability". In: *The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*. IEEE. 2006, pp. 192–192.

[37] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. "Understanding variable importances in forests of randomized trees". In: *Advances in neural information processing systems*. 2013, pp. 431–439.

[38] Walid Maalej and Martin P Robillard. "Patterns of knowledge in API reference documentation". In: *IEEE Transactions on Software Engineering* 39.9 (2013), pp. 1264–1282.

[39] Brian W Matthews. "Comparison of the predicted and observed secondary structure of T4 phage lysozyme". In: *Biochimica et Biophysica Acta (BBA)-Protein Structure* (1975).

[40] G Harry Mc Laughlin. "SMOG grading-a new readability formula". In: *Journal of reading* 12.8 (1969), pp. 639–646.

[41] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[42] Shane McIntosh and Yasutaka Kamei. "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction". In: *IEEE Transactions on Software Engineering* 44.5 (2017), pp. 412–428.

[43] Roberto Minelli, Andrea Mocci, and Michele Lanza. "I know what you did last summer - An investigation of how developers spend their time". In: *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE. 2015, pp. 25–35.

[44] Sanjay Misra and Ibrahim Akman. "Comparative study of cognitive complexity measures". In: *Computer and Information Sciences, 2008. ISCIS'08. 23rd International Symposium on*. IEEE. 2008, pp. 1–4.

[45] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects". In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, pp. 362–373.

[46] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. "Divide-and-conquer approach for multi-phase statistical migration for source code (t)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 585–596.

[47] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. "Learning API usages from bytecode: a statistical approach". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 416–427.

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

[49] Anders Odén, Hans Wedel, et al. "Arguments for Fisher's permutation test". In: *The Annals of Statistics* 3.2 (1975), pp. 518–520.

[50] Abraham Naftali Oppenheim. *Questionnaire design, interviewing and attitude measurement*. Bloomsbury Publishing, 2000.

[51] Chris Parnin and Christoph Treude. "Measuring API documentation on the web". In: *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*. 2011, pp. 25–30.

[52] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow". In: *Georgia Institute of Technology, Tech. Rep* 11 (2012).

[53] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. "Fine-grained just-in-time defect prediction". In: *Journal of Systems and Software* 150 (2019), pp. 22–36.

[54] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. "Discovering information explaining API types using text classification". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 869–879.

[55] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. "StORMeD: Stack Overflow ready made data". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 474–477.

[56] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. "Mining StackOverflow to turn the IDE into a self-confident programming prompter". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 102–111.

[57] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. "Understanding and classifying the quality of technical forum questions". In: *2014 14th International Conference on Quality Software*. IEEE. 2014, pp. 343–352.

[58] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. "On the "naturalness" of buggy code". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 428–439.

[59] Veselin Raychev, Martin Vechev, and Eran Yahav. "Code completion with statistical language models". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pp. 419–428.

[60] Peter C Rigby and Martin P Robillard. "Discovering essential code elements in informal documentation". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 832–841.

[61] Martin P Robillard and Yam B Chhetri. "Recommending reference API documentation". In: *Empirical Software Engineering* 20.6 (2015), pp. 1558–1586.

[62] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. "On-demand developer documentation". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 479–483.

[63] Taniya Saini and Sachin Tripathi. "Predicting tags for Stack Overflow questions using different classifiers". In: *2018 4th International Conference on Recent Advances in Information Technology (RAIT)*. IEEE. 2018, pp. 1–5.

[64] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. "Improving code readability models with textual features". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE. 2016, pp. 1–10.

[65] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. "Automatically Assessing Code Understandability: How Far Are We?" In: *Proceedings of the 32nd IEEE/ACM International*

Conference on Automated Software Engineering. IEEE Press. 2017, pp. 417–427.

[66] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. "Automatically Assessing Code Understandability". In: *IEEE Transactions on Software Engineering* (2019).

[67] RJ Senter and Edgar A Smith. *Automated readability index*. Tech. rep. CINCINNATI UNIV OH, 1967.

[68] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *ACM sigsoft software engineering notes* 30.4 (2005), pp. 1–5.

[69] Manuel Sojer and Joachim Henkel. "License risks from ad hoc reuse of code from the internet". In: *Communications of the ACM* 54.12 (2011), pp. 74–81.

[70] D Srinivasulu, Adepu Sridhar, and Durga Prasad Mohapatra. "Evaluation of Software Understandability Using Rough Sets". In: *Intelligent Computing, Networking, and Informatics*. Springer, 2014, pp. 939–946.

[71] Mathupayas Thongmak and Pornsiri Muenchaisri. "Measuring understandability of aspect-oriented code". In: *International Conference on Digital Information and Communication Technology and Its Applications*. Springer. 2011, pp. 43–54.

[72] Christoph Treude and Martin P Robillard. "Augmenting api documentation with insights from stack overflow". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 392–403.

[73] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. "On the localness of software". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 269–280.

[74] Medha Umarji, Susan Elliott Sim, and Crista Lopes. "Archetypal internet-scale source code searching". In: *IFIP International Conference on Open Source Systems*. Springer. 2008, pp. 257–263.

[75] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. "Recovering clear, natural identifiers from obfuscated JS names". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 683–693.

[76] Christopher Vendome, Daniel German, Massimiliano Di Penta, Gabriele Bavota, Mario Linares-Vásquez, and Denys Poshyvanyk. "To Distribute or Not to Distribute? Why Licensing Bugs Matter". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 268–279.

[77] Shaowei Wang, David Lo, Bogdan Vasilescu, and Alexander Serebrenik. "EnTagRec++: An enhanced tag recommendation system for software information sites". In: *Empirical Software Engineering* 23.2 (2018), pp. 800–832.

[78] Elaine J. Weyuker. "Evaluating software complexity measures". In: *IEEE Transactions on Software Engineering (TSE)* 14.9 (1988), pp. 1357–1365.

[79] Frank Wilcoxon. "Individual comparisons by ranking methods". In: *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[80] Edmund Wong, Jinqiu Yang, and Lin Tan. "Autocomment: Mining question and answer sites for automatic comment generation". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE. 2013, pp. 562–567.

[81] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. "Tag recommendation in software information sites". In: *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE. 2013, pp. 287–296.

[82] Xin Xia, David Lo, Denzil Correa, Ashish Sureka, and Emad Shihab. "It takes two to tango: Deleted stack overflow question prediction with text and meta features". In: *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*. Vol. 1. IEEE. 2016, pp. 73–82.

[83] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. "What do developers search for on the web?" In: *Empirical Software Engineering* 22.6 (2017), pp. 3149–3185.

[84] Di Yang, Aftab Hussain, and Cristina Videira Lopes. "From query to usable code: an analysis of Stack Overflow code snippets". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM. 2016, pp. 391–402.

[85] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. "Stack Overflow in Github: any snippets there?" In: *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE. 2017, pp. 280–290.

[86] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. "Example overflow: Using social media for code recommendation". In: *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. IEEE Press. 2012, pp. 38–42.