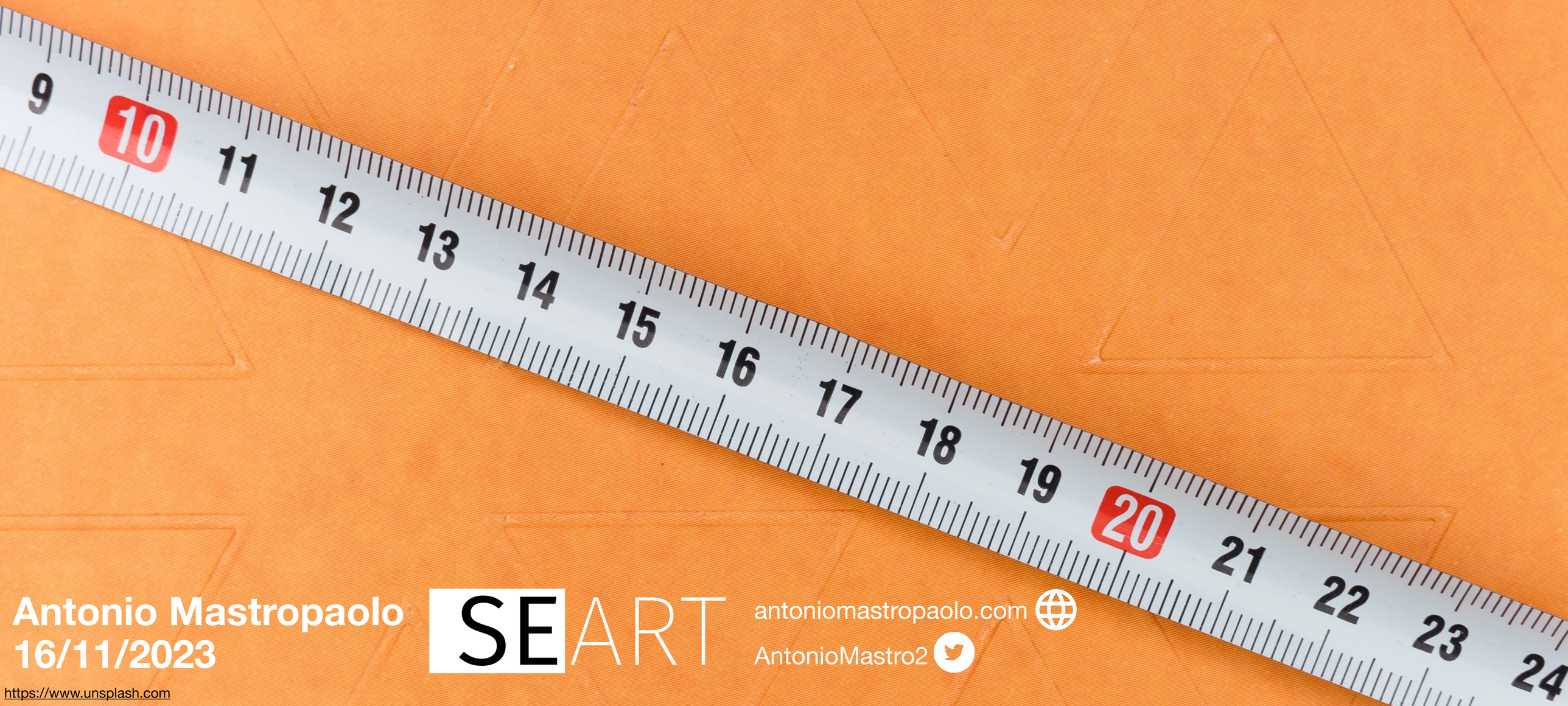
Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization



Antonio Mastropaolo
16/11/2023

SEART

antoniomastropaolo.com

AntonioMastro2

https://www.unsplash.com

https://www.unsplash.com

Documenting the code is extremely important.....

https://www.unsplash.com

Documenting the code is extremely important.....

SUC**

https://www.unsplash.com

Commented code is more understandable as compared to non-commented code

The Effect of Modularization and Comments on Program Comprehension

S. N. Woodfield

Computer Sciences Department
Arizona State University
Tempe, Arizona 85281

H. E. Dunsmore
V. Y. Shen

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

Abstract

An experiment was conducted to investigate how different types of modularization and comments are related to programmers' ability to understand programs. Forty-eight experienced programmers were given eight different versions of the same program and asked to answer a twenty question quiz used to measure comprehension. These eight different versions were the result of the program being constructed with four types of modularization (monolithic, functional, super, and abstract data type), each with and without comments. Those subjects whose programs contained comments were able to answer more questions than those without comments. Also, those subjects who were given the abstract data type version of the program were able to do significantly better than those with any other type of modularization.

Introduction

One software feature common to almost all large programs is the use of modularization. At first, modularization was motivated by the desire to conserve computer memory. A program was considered properly modularized if it was possible to execute the program using the available memory. Later, as memory became larger, code was modularized to make it easier for people to work with programs. Now, considering the human factors of programming, it has become more difficult to determine or describe proper modularization. Today almost everyone agrees that modularity is an important aspect of reducing the cost of developing software systems. Yet, there remain many questions such as "How large should a module should be?" or "How should modules interface with each other?".

This paper reports on an empirical study of several modularization methodologies to see how different approaches affect the ability of programmers to understand programs. The Yourdon and Constantine [1] methodology attempts to divide a program into functional units, one function per physical module. (A physical module is a section of code whose boundaries are defined by the syntax of the language such as a subroutine in Fortran or a procedure in Pascal). A routine implementing only one function should be easy to understand since all the code is related to that one function. Furthermore, the function (module) is constructed only once and is invoked in other parts of the program where that function is needed. Since only one function resides in a physical module, the identification of the function's boundaries is simple.

The abstract data type approach to modularization may include several functions in one physical module [2,3]. The module brings together all operations that manipulate a single data type. This abstract data type approach is a special case of physical modules containing several logical modules. (A logical module is a section of code which implements only one function. A logical module need not correspond one-to-one with physical modules).

This paper reports on research conducted to help determine how best to modularize programs to aid in comprehension. As described above, possibilities include physical modules containing exactly one logical module or more than one logical module. It is possible that more than one logical module in a physical module is acceptable if the logical modules are easily identified and easily understood.

To facilitate understanding one means of separating the logical modules in a physical module is to insert a brief comment at the beginning of each logical module. Comments used in this research were designed to describe a logical module's contents. If there were more than one logical module per physical module, the inserted comments were intended to help define the boundaries of the logical modules. Previous work has shown that comment statements in a program are not harmful, but may not be very useful either [4,5]. Such studies were conducted primarily to decide if the prolific use of comments aids in the general comprehension of a program. Our research attempts to determine whether comments used more judiciously (i.e., in the right locations) might be a better approach. Specifically, this research attempts to determine if short comments, inserted just before a logical module, can aid comprehension by briefly describing the function of the logical module and helping to define the logical module's boundaries.

We considered four types of modularization:

(1) monolithic - (i.e., non-modularization) a program with one physical module.

CH1627-9/81/0000/0215$00.75 © 1981 IEEE

Quality Analysis of Source Code Comments

Daniela Steidl Benjamin Hummel Elmar Juergens
CQSE GmbH, Garching b. München, Germany
{steidl,hummel,juergens}@cqse.eu

ICPC'13

***Abstract*—A significant amount of source code in software systems consists of comments, *i. e.*, parts of the code which are ignored by the compiler. Comments in code represent a main source for system documentation and are hence key for source code understanding with respect to development and maintenance. Although many software developers consider comments to be crucial for program understanding, existing approaches for software quality analysis ignore system commenting or make only quantitative claims. Hence, current quality analyzes do not take a significant part of the software into account. In this work, we present a first detailed approach for quality analysis and assessment of code comments. The approach provides a model for comment quality which is based on different comment categories. To categorize comments, we use machine learning on Java and C/C++ programs. The model comprises different quality aspects: by providing metrics tailored to suit specific categories, we show how quality aspects of the model can be assessed. The validity of the metrics is evaluated with a survey among 16 experienced software developers, a case study demonstrates the relevance of the metrics in practice.**

I. Introduction

A significant amount of source code in software systems consists of comments, which document the implementation and help developers to understand the code, *e. g.*, for later modification or reuse: Several researchers have conducted experiments showing that commented code is easier to understand than code without comments [1], [2]. Comments are the second most-used documentary artifact for code understanding, behind only the code itself [3]. In addition, source code documentation is also vital in maintenance and forms an important part of the general documentation of a system. In contrast to external documentation, comments in source code are a convenient way for developers to keep documentation and code consistently up to date. Developers widely agree that poor general documentation leads to misunderstandings [4], and studies have shown that poor documentation significantly lowers the maintainability of software [5]. Although developers commonly agree on the importance of software documentation [3], commenting code is often neglected due to release deadlines and other time pressure during development.

Previous approaches to analyzing software quality ignore comments, or make only quantitative claims: they evaluate the comment ratio of a system to measure documentation quality [6], [7]. However, this metric is not sufficient: Several comments (copyrights or commented out code) should be excluded as they do not enhance system understanding and quantitative measures cannot detect outdated/ useless comments.

Furthermore, a complete model of comment quality does not exist. Coding conventions, *e. g.*, marginally touch on the topic of commenting code but mostly lack depth and precision [8]. So far, (semi-) automatic methods for comment quality assessment have not been developed as comment analysis is a difficult task: Comments comprise natural language and have no mandatory format aside from syntactic delimiters. Hence, algorithmic solutions will be heuristic in nature.

Problem Statement. Current quality analysis approaches ignore system commenting or are restricted to the comment ratio metric only. Hence, a major part of source code documentation is ignored during software quality assessment.

Contribution. Based on comment classification, we provide a semi-automatic approach for quantitative and qualitative evaluation of comment quality.

We present a semi-automatic approach for comment quality analysis and assessment. First, we perform comment categorization both for Java and C/C++ programs based on machine learning to differentiate between different comment types. Comment categorization enables a detailed quantitative analysis of a system's comment ratio and a qualitative analysis tailored to suit each single category. Comment categorization is the underlying basis of our comprehensive quality model. The model comprises quality attributes for each comment category based on four criteria: consistency throughout the project, completeness of system documentation, coherence with source code, and usefulness to the reader. To assess quality attributes, we provide metrics detecting quality defects in comments of specific categories. We evaluate the metrics' validity and relevance separately: Validity is evaluated with a survey among experienced software developers. The survey shows that the metrics can additionally give refactoring recommendations. A case study of five open source projects evaluates the relevance of our approach showing that comment classification provides better insights of the system documentation quality than the simple comment ratio metric and that our metrics reveal quality defects in practice.

II. Related Work

We group the related work into categories of general comment analysis, information retrieval techniques, comment evolution studies, and code recognition approaches.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant EvoCon, 01IS12034A. The responsibility for this article lies with the authors.

978-1-4673-3092-3/13/$31.00 © 2013 IEEE

ICPC 2013, San Francisco, CA, USA

Steidl et al.

Developers do not always carefully comment code, or update existing comments in response to code changes

Do Code and Comments Co-Evolve?
On the Relation Between Source Code and Comment Changes

Beat Fluri, Michael Würsch, and Harald C. Gall
s.e.a.l. – software architecture and evolution lab
Department of Informatics
University of Zurich, Switzerland
{fluri,wuersch,gall}@ifi.uzh.ch

Abstract

Comments are valuable especially for program understanding and maintenance, but do developers comment their code? To which extent do they add comments or adapt them when they evolve the code? We examine the question whether source code and associated comments are really changed together along the evolutionary history of a software system. In this paper, we describe an approach to map code and comments to observe their co-evolution over multiple versions. We investigated three open source systems (i.e., ArgoUML, Azureus, and JDT Core) and describe how comments and code co-evolved over time. Some of our findings show that: 1) newly added code—despite its growth rate—barely gets commented; 2) class and method declarations are commented most frequently but far less, for example, method calls; and 3) that 97% of comment changes are done in the same revision as the associated source code change.

1. Introduction

"Comment your code!" The task of commenting one's source code is often neglected; even though everybody who is writing software knows the value of good comments [20]. Reading code is a fundamental task during software engineering [10]—and code is read more often than it is written. Even books covering best-practices in commenting exist, *e.g.*, *The Elements of Java Style* by Vermeulen *et al.* [21]. Comments allow one to understand the code faster and deeper and to improve its readability [18, 19]. Especially, they are crucial to sustain software maintainability and aid in reverse engineering, for example when applying the *Read All the Code in One Hour* reengineering pattern [5]. Elshoff and Marcotty already stated in the early eighties that comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs [6]. This finding was confirmed by the studies of Ted Tenny [19]. But as the example of Lakhotia shows, sometimes programmers do not care that someone else might want to understand the source code [13].

For maintenance and reverse engineering tasks both, the lack of comments as well as outdated comments are counter-productive. To understand whether the comments are a reason for decreasing maintainability in software projects we address the following research questions in this paper:

1. Does the ratio between comment and source code remain stable over the history of a software project (*i.e.*, is there a recognizable effort made to comment the source code)?
2. Which source code entities are most likely to be commented?
3. Are comments adapted when source code is changed (*i.e.*, are comments kept up-to-date) and when does the adaptation take place—while changing the source code or afterwards?

The contribution of this paper is an approach to map source code entities to comments in the code and a technique to extract comment changes over the history of a software project. We have conducted three experiments on three different open source software systems to answer our research questions: ArgoUML, Azureus, and JDT Core. We describe how comments and code co-evolved over time. Some of our findings show that: 1) newly added code—despite its growth rate—barely gets commented; 2) class and method declarations are commented most frequently but far less, for example, method calls; and 3) that 97% of comment changes are done in the same revision as the associated source code change.

The remainder of the paper is structured as follows. In Section 2 we present our approach to investigate the relation

A Study of the Documentation Essential to Software Maintenance

Sergio Cozzetti B. de Souza
Universidade Católica de Brasília
Brasília, DF, BRAZIL

Nicolas Anquetil
Universidade Católica de Brasília
Brasília, DF, BRAZIL
anquetil@ucb.br

Káthia M. de Oliveira
Universidade Católica de Brasília
Brasília, DF, BRAZIL
kathia@ucb.br

ABSTRACT

Software engineering has been striving for years to improve the practice of software development and maintenance. Documentation has long been prominent on the list of recommended practices to improve development and help maintenance. Recently however, agile methods started to shake this view, arguing that the goal of the game is to produce software and that documentation is only useful as long as it helps to reach this goal.

On the other hand, in the re-engineering field, people wish they could re-document useful legacy software so that they may continue maintain them or migrate them to new platform.

In these two case, a crucial question arises: "How much documentation is enough?" In this article, we present the results of a survey of software maintainers to try to establish what documentation artifacts are the most useful to them.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*

General Terms

software system documentation, empirical study, software maintenance, program understanding

1. INTRODUCTION

Among all the recommended practices in software engineering, software documentation has a special place. It is one of the oldest recommended practices and yet has been, and continue to be, renowned for its absence (e.g. [4]). There is no end to the stories of software systems (particularly legacy software) lacking documentation or with outdated documentation. For years, the importance of documentation has been stressed by educators, processes, quality models, etc. and despite of this we are still discussing why it is not generally created and maintained (e.g. [11]).

The topic gained renewed interest with two recent trends:

- Agile methods question the importance of documentation as a development aid;
- The growing gap between "traditional" (e.g. COBOL) and up-to-date technologies (e.g. OO or web-oriented) increased the pressure to re-document legacy software.

Both issues raise a similar question: What documentation would be most useful to software maintenance?

If they propose a renewed development paradigm, agile methods do not bring significant changes for software maintenance. They do claim that permanent re-factoring turns maintenance into a normal state of the methods. However, they do not explain how such methods would work over extended periods of time, when a development team is sure to disperse with the knowledge it has of the implementation details. Documentation is still a highly relevant artifact of software maintenance.

Legacy software re-documentation tries to remedy the deficiencies of the past in terms of up-to-date documentation. However, it is a costly activity, difficult to justify to users because it does not bring any visible change for them (at least in the short term).

In this paper we present a survey of software maintainers trying to establish the importance of various documentation artifacts for maintenance. The paper is divided as follows: In Section 2, we review some basic facts about software maintenance and its needs; in Section 3, we summarize the relevant literature on software documentation; in Section 4, we present the survey we conducted; and in Section 5, we comment the result of the survey.

2. SOFTWARE MAINTENANCE

Maintenance is traditionally defined as any modification made on a system after its delivery. Studies show that software maintenance is, by far, the predominant activity in software engineering (90% of the total cost of a typical software [17, 21]). It is needed to keep software systems up-to-date and useful: Any software system reflects the world within which it operates, when this world changes, the software needs to change accordingly. Lehman's first law of software evolution (law of continuing change, [13]) is that "a program that is used undergoes continual change or becomes progressively less useful". Maintenance is mandatory,

SIGDOC'05, September 21–23, 2005, Coventry, United Kingdom.

A Study of the Documentation Essential to Software Maintenance

Sergio Cozzetti B. de Souza
Universidade Católica de Brasília
Brasília, DF, BRAZIL

ABSTRACT

Software engineering has been ...
the practice of software deve...
umentation has long been ...
mended practices to im...
tenance. Recently how...
this view, arguing th...
software and that ...
helps to reach thi...
On the other h...
they could re...
may continu...
form.
In thes...
docume...
result...
wha...

Authorized licens... Restrictions apply.

Software Quality Journal, 17:367–394, 2009
DOI 10.1007/s11219-009-9075-x

Analyzing the co-evolution of comments and source code

Beat Fluri · Michael Würsch · Emanuel Giger · Harald C. Gall

Published online: 26 March 2009
© Springer Science+Business Media, LLC 2009

Abstract Source code comments are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Nevertheless, commenting source code and keeping comments up-to-date is often neglected for reasons of time or programmers obliviousness. In this paper, we investigate the question whether developers comment their code and to what extent they add comments or adapt them when they evolve the code. We present an approach to associate comments with source code entities to track their co-evolution over multiple versions. A set of heuristics are used to decide whether a comment is associated with its preceding or its succeeding source code entity. We analyzed the co-evolution of code and comments in eight different open source and closed source software systems. We found with statistical significance that (1) the relative amount of comments and source code grows at about the same rate; (2) the type of a source code entity, such as a method declaration or an if-statement, has a significant influence on whether or not it gets commented; (3) in six out of the eight systems, code and comments co-evolve in 90% of the cases; and (4) surprisingly, API changes and comments do not co-evolve but they are re-documented in a later revision. As a result, our approach enables a quantitative assessment of the commenting process in a software system. We can, therefore, leverage the results to provide feedback during development to increase the awareness of when to add comments or when to adapt comments because of source code changes.

Keywords Software evolution analysis · Software repositories · Source code changes · Comment changes · Comment quality · Software maintenance

1 Introduction

Documenting software is painful, especially when time is scarce and release deadlines are putting serious pressure on development teams, making it necessary to prioritize tasks.

B. Fluri (✉) · M. Würsch · E. Giger · H. C. Gall
Department of Informatics, University of Zurich, Zurich, Switzerland
e-mail: fluri@ifi.uzh.ch

2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)

A Large-Scale Empirical Study on Code-Comment Inconsistencies

Fengcai Wen, Csaba Nagy, Gabriele Bavota, Michele Lanza
Software Institute, Università della Svizzera italiana (USI), Switzerland
{*fengcai.wen, csaba.nagy, gabriele.bavota, michele.lanza*}*@usi.ch*

***Abstract*—Code comments are a primary means to document source code. Keeping comments up-to-date during code change activities requires substantial time and attention. For this reason, researchers have proposed methods to detect code-comment inconsistencies (*i.e.,* comments that are not kept in sync with the code they document) and studies have been conducted to investigate this phenomenon. However, these studies were performed at a small scale, relying on quantitative analysis, thus limiting the empirical knowledge about code-comment inconsistencies. We present the largest study at date investigating how code and comments co-evolve. The study has been performed by mining 1.3 Billion AST-level changes from the complete history of 1,500 systems. Moreover, we manually analyzed 500 commits to define a taxonomy of code-comment inconsistencies fixed by developers. Our analysis discloses the extent to which different types of code changes (*e.g.,* change of *selection* statements) trigger updates to the related comments, identifying cases in which code-comment inconsistencies are more likely to be introduced. The defined taxonomy categorizes the types of inconsistencies fixed by developers. Our results can guide the development of tools aimed at detecting and fixing code-comment inconsistencies.**

***Index Terms*—Software Evolution, Code Comments**

I. Introduction

Any code-related activity lays its foundations in program comprehension: before fixing a bug, refactoring a class, or writing new tests, developers first need to acquire knowledge about the involved code components. As recently shown by Xia *et al.* [1], this results in 58% of developers' time spent comprehending code. Besides the code itself, code comments are considered as the most important form of documentation for program comprehension [2]. Indeed, not surprisingly, studies showed that commented code is easier to comprehend than uncommented code [3], [4]. This empirical evidence also pushed researchers to consider code comments as a pivotal factor to study technical debt [5]–[7], or to assess code quality [8], [9].

While the importance of code comments is undisputed, developers do not always have the chance to carefully comment new code and/or to update comments as consequence of code changes [10]. This latter scenario might result in the introduction of code-comment inconsistencies, manifesting when the source code does not co-evolve with the related comments. For example, if a method comment is not updated after major changes to the method's application logic, the comment might provide misleading information to developers comprehending the method, hindering program comprehension rather than fostering it.

Given the potential harmfulness of code-comment inconsistencies, several researchers studied the co-evolution of code and comments [11]–[14], while others proposed techniques and tools able to detect code-comment inconsistencies automatically [15]–[18]. These techniques are able to identify specific types of code-comment inconsistencies. For example, @TCOMMENT [17] detects inconsistencies between Javadoc comments related to `null` values and exceptions with the behavior implemented in the related method's body, while Fraco [18] focuses on inconsistencies introduced as result of rename refactoring operations. Still, more research is needed in this area to increase the types of code-comment inconsistencies that can be automatically identified. Also, the empirical evidence provided by studies that pioneered the investigation of code-comment evolution [11]–[14] is limited to the analysis of the change history of a few software systems (less than 10).

To raise the knowledge about the co-evolution of code and comments and the introduction/fixing of code-comment inconsistencies, we present a large-scale empirical study quantitatively and qualitatively analyzing these phenomena. We mine the complete change history of 1,500 Java projects hosted on GitHub for a total of 3,323,198 analyzed commits. For each commit, we use GumTreeDiff [19] to extract AST operations performed on the files modified in it. In this way, we captured fine-grained changes performed in code (*e.g.,* change of a *selection* statement) as well as update, delete, and insert operations performed in the related comments. Overall, this process resulted in a database of ∼476 GB containing ∼1.3 Billion AST-level operations impacting code or comments. Using this data, we study the extent to which code changes impacting different code constructs (*e.g., literals*, *iteration statements*) trigger the update of the related code comments (*e.g.,* the developer adds a try statement and updates the method comment to "document" the changed code behavior).

Then, we manually analyze 500 commits identified, via a keywords-matching mechanism, as likely related to the fixing of code-comment inconsistencies. The output of this analysis is a taxonomy of code comment-related changes implemented by developers, from which we present relevant cases related to code-comment inconsistencies, and discuss implications for researchers and practitioners.

As a contribution to the research community, we make the database of fine-grained code changes publicly available. This enables the replication of this work, making also other types of investigations possible.

DOI 10.1109/ICPC.2019.00019

Retrieval-based Neural Source Code Summarization

2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)

Zhang*
g University,

Xu Wang*†
SKLSDE Lab, Beihang University, China
wangxu@act.buaa.edu.cn

Hongyu Zhang
The University of Newcastle, Australia
hongyu.zhang@newcastle.edu.au

Sun*
University,

Xudong Liu*
SKLSDE Lab, Beihang University, China
liuxd@act.buaa.edu.cn

ACM Reference Format:
n Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *42nd International Conference on Software Engineering (ICSE '20), May23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380383

pt

Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks

Antonio Mastropaolo*, Simone Scalabrino†, Nathan Cooper‡, David Nader Palacio‡, Denys Poshyvanyk‡, Rocco Oliveto†, Gabriele Bavota*
*SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland
†SEMERU @ Computer Science Department, William and Mary, USA
‡University of Molise, Italy

arning (DL) techniques are gaining more and
software engineering community. They have
eral code-related tasks, such as automatic
ments generation. Recent studies in the
ing (NLP) field have shown that the
former (T5) architecture can achieve
a variety of NLP tasks. The basic
n a model on a large and generic
sk (e.g., filling masked words in
e-trained. It is fine-tuned on
nce related to a specific
the T5 model performs
port code-related tasks,
composed of natural
n, we fine-tune such
evious works that
ode mutants. (ii)
comments. We
ith the results
sed solutions
exploiting
base, can
rning

For example, in the work by Wat
string representing a test method wit
and the output is an appropriate as
given test. In the approach by Haque
is composed of strings representing a subro
while the output is a natural language summ
the subroutine.
Recent years have seen the raise of transfer l
field of natural language processing. The basic id
pre-train a model on a large and generic dataset by us
supervised task, *e.g.*, masking tokens in strings and as
model to guess the masked tokens. Then, the trained mo
fine-tuned on smaller and specialized datasets, each one a
at supporting a specific task. In this context, Raffel *et al.* [
proposed the T5 (Text-To-Text Transfer Transformer) model
pre-trained on a large natural language corpus and fine-tuned
to achieve state-of-the-art performance on many tasks, all
characterized by text-to-text transformations. The goal
The goal of this work is to empirically investigate the
potential of a T5 model when pre-trained and fine-tuned to
support many of the previously listed code-related tasks also

An Empirical Study on Code Comment Completion

Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, Gabriele Bavota
SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland

***Abstract*—Code comments play a prominent role in program comprehension activities. However, source code is not always documented and code and comments not always co-evolve. To deal with these issues, researchers have proposed techniques to automatically generate comments documenting a given code at hand. The most recent works in the area applied deep learning (DL) techniques to support such a task. Despite the achieved advances, the empirical evaluations of these approaches show that they are still far from a performance level that would make them valuable for developers. We tackle a simpler and related problem: Code comment completion. Instead of generating a comment for a given code from scratch, we investigate the extent to which state-of-the-art techniques can help developers in writing comments faster. We present a large-scale study in which we empirically assess how a simple n-gram model and the recently proposed Text-To-Text Transfer Transformer (T5) architecture can perform in autocompleting a code comment the developer is typing. The achieved results show the superiority of the T5 model, despite the n-gram model being a competitive solution.**

***Index Terms*—Empirical Study, Code Comments**

I. Introduction

For example, in the recent work by Mastropaolo *et*
the authors show that state-of-the-art tech
generate comments equivalen
in only ∼10% of cases. Fo
completely relying on "*machi*
of today, a far-fetched goal th
is unlikely to result in major br
In this work, we tackle the simp
completion, in which the "machin
a comment that the developer start
done for code tokens by code compl

The code comment completion
tackled by Ciurumelea *et al.* [24]
code: They study whether a deep l
dict the next word that a developer
commenting code. This is, to the best
only work done in this area. Stemmin
present in this paper a large-scale study a
a simple n-gram model and of the recent
Text Transfer Transformer (T5) architectur
code comment completion for Java progra
the work by Ciurumelea *et al.* [24], besi
different context (*i.e.,* Python *vs* Java) we:
actual advantages brought by a DL-based m
simpler n-gram model that can be trained in
time required by T5; (ii) do not limit our stu
the single next word the developer is likel
evaluate how the investigated techniques perfor
to predict longer word sequences (*e.g.,* the ne
providing a more advanced completion support t
We also study the complementarity of the two tec
report qualitative examples of correct and wrong
to understand their strengths and limitations.

Our study has been run on a dataset composed b
Java methods with their related comments. The
results can be summarized as follows. First, the T5 mo
performs the n-gram model, achieving superior perfo
in all the comment completion scenario we tested. S
despite being more performant, the T5 model exploits as
not only the first part of the comment already writte
the developer (also used by the n-gram model), but a
context representing the relevant code for the commen

Automated Documentation of Android Apps

Emad Aghajani, *Student Member, IEEE*, Gabriele Bavota, *Member, IEEE*,
Mario Linares-Vásquez, *Member, IEEE*, and Michele Lanza, *Senior Member, IEEE*

Abstract—Developers do not always have the knowledge needed to understand source code and must refer to different resources (e.g., teammates, documentation, the web). This non-trivial process, called program comprehension, is very time-consuming. While many approaches support the comprehension of a given code at hand, they are mostly focused on defining extractive summaries from the code (i.e., on selecting from a given piece of code the most important statements). However, if the information needed to comprehend the code is not there, their usefulness is limited. We present ADANA, an approach to automatically comments describing a given piece of Android code. ADANA reuses the descriptions of similar and well-documented code snippets retrieved from various online resources. Our evaluation has shown that ADANA is able to aid the program comprehension

s—Program comprehension, documentation, Android

do not always possess the knowl-
rehend the source code they are
true when code lacks documen-
to make up for the lacking
er to teammates and other
the Internet [5], such as
However, what develo-
ieces of information,
ways help to answer
f source code is
utomated sum-
[11], [12], [13],
ummaries [14].
lements is

We evaluated ADANA in three studies. Results sh
that (i) ADANA can, on average, automatically do
with code comments one third of the code
mobile app, (ii) the ASIA clone detector
comments injected by ADANA help
both in terms of time needed and

2 ADANA

ADANA is
Android

Automatic Source Code Summarization with Extended Tree-LSTM

suke Shido*, Yasuaki Kobayashi*, Akihiro Yamamoto*, Atsu
*Graduate School of Informatics, Ky
{shido, kobayashi}@iip.ist.i.kyoto-u.ac.jp,
†Center for Exploratory Research, Hi
{atsushi.miyamoto.eu, tadayuki.matsumura

ct—**Neural machine translation models are used to**
cally generate a document from given source code since
be regarded as a machine translation task. Source
mmarization is one of the components for automatic
t generation, which generates a summary in natural
from given source code. This suggests that techniques
eural machine translation, such as Long Short-Term
can be used for source code summarization.
a considerable difference between source code
uage: Source code is essentially *structured*,
conditional branching, etc. Therefore, there
apply known machine translation mod

trees (ASTs) cap
portant

A Convolutional Neural Network for Language-Agnostic Source Code Summarization

Jessica Moore[1], Ben Gelman[1], and David Slater[1]
[1]*Machine Learning Group, Two Six Labs, Arlington, Virginia, United States*
{*jessica.moore, ben.gelman, david.slater*}*@twosixlabs.com*

Keywords: Natural Language Processing, Deep Learning, Source Code, Crowdsourcing, Automatic Documen

Descriptive comments play a crucial role in the software engineering process. They decre
time, enable better bug detection, and facilitate the reuse of previously written code. Ho
are commonly the last of a software developer's priorities and are thus either insufficient
Automatic source code summarization may therefore have the ability to significantly i
development process. We introduce a novel encoder-decoder model that summarizes so
writing a comment to describe the code's functionality. We make two primary innov
source code summarization models. First, our encoder is fully language-agnostic
input preprocessing. Second, our decoder has an open vocabulary, enabling it to pr
not seen in training. We demonstrate results comparable to state-of-the-art method
set and provide the first results on a data set consisting of multiple programming

1 Introduction

udies of software development patterns suggest that
engineers spend a significant amount of their
on program comprehension tasks,
entation or trying to understand
2017; Minelli et al.,
developer inter-
all confirm

the potential to signific
velopment life-cycle
descriptive commen
selves did not do
whether neural m
used to automat
source code.
Current s
tion models
ing lexeme
trees (AS
al., 20

Ensemble Models for Neural Source Code Summarization of Subroutines

Alexander LeClair, Aakash Bansal, and Collin McMillan
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{aleclair, abansal1, cmc}@nd.edu

***Abstract*—A source code summary of a subroutine is a brief description of that subroutine. Summaries underpin a majority of documentation consumed by programmers, such as the method summaries in JavaDocs. Source code summarization is the task of writing these summaries. At present, most state-of-the-art approaches for code summarization are neural network-based solutions akin to seq2seq, graph2seq, and other encoder-decoder architectures. The input to the encoder is source code, while the decoder helps predict the natural language summary. While these models tend to be similar in structure, evidence is emerging that different models make different contributions to prediction quality – differences in model performance are orthogonal and complementary rather than uniform over the entire dataset. In this paper, we explore the orthogonal nature of different neural code summarization approaches and propose ensemble models to exploit this orthogonality for better overall performance. We demonstrate that a simple ensemble strategy boosts performance by up to 14.8%, and provide an explanation for this boost. The takeaway from this work is that a relatively small change to the inference procedure in most neural code summarization techniques leads to outsized improvements in prediction quality.**

***Index Terms*—source code summarization, automatic documentation generation, neural networks**

I. Introduction

A source code "summary" of a subroutine is a natural
language description of that subroutine. Summaries are the
foundation of many documentation systems such as JavaDocs,
where the summary is used to help programmers quickly gain
an understanding of the purpose of the subroutine, without
reading its source code [1]. The task of writing
come to be known as source code
an active research area for
mmer behavior

a source code summary
code summarization are
decoder model, in whi
representation of source
natural language summ
proaches have almost
as sentence templates
Current strategies
broadly classified b
modeling in the
code as text, foc
natural language
approaches that
source file [13].
code with an
such as the A
scrutiny of G
the AST or C
showing pro
Recent w
ing on pro
tions of co
better pre
[20]. Th
such as
Howeve
lations
than a
by th
to th
out
pre

Code Generation as a Dual Task of Code Summarization

Bolin Wei[†], Ge Li[†*], Xin Xia[‡], Zhiyi Fu[†], Zhi Jin[†*]
[†]Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, China; Software Institute, Peking University, China
[‡]Faculty of Information Technology, Monash University, Australia
[†]bolin.wbl@gmail.com [†]{lige,ypfzy,zhijin}@pku.edu.cn
[‡]xin.xia@monash.edu

Abstract

Code summarization (CS) and code generation (CG) are two crucial tasks in the field of automatic software development. Various neural network-based approaches are proposed to solve these two tasks separately. However, there exists a specific intuitive correlation between CS and CG, which has not been exploited in previous work. In this paper, we apply the relations between two tasks to improve the performance of both tasks. In other words, exploiting the duality between the two tasks, we propose a dual training framework to train the two tasks simultaneously. In this framework, we consider the dualities on probability and attention weights, and design corresponding regularization terms to constrain the duality. We evaluate

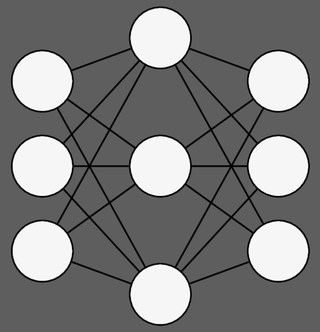
Code Summarizer

INPUT

```
public ConnectionConsumer createConnectionConsumer
    (final Destination destination)
       throws JMSException{

   if(LOGGER.isTraceEnabled())
   {
       ActiveMQRALogger.LOGGER.
         trace("Create connectionConsumer");
   }
   else {
       throw new IllegalStateException(ISE);
   }
}
```

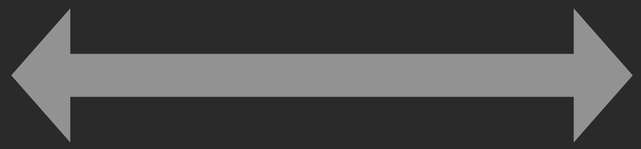
DL-Model



PREDICTION

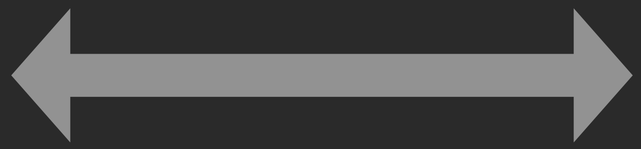
Create a connection to the Consumer.

MODEL EVALUATION

GROUND TRUTH		*MODEL'S PREDICTION*
`Create a connection to the consumer.`		`Create a connection to the consumer.`

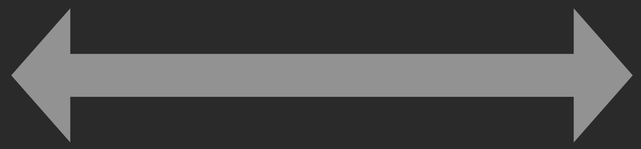
MODEL EVALUATION

GROUND TRUTH	*MODEL'S PREDICTION*
`Create a connection to the consumer.`	`Create a connection to the consumer.`



MODEL EVALUATION

GROUND TRUTH	*MODEL'S PREDICTION*
`Create a connection to the consumer.`	`Create a logger .`



NLP (Natural Language Processing) Related Metrics

BLEU SCORE[1]

ROUGE SCORE[2]

BERT SCORE[3]

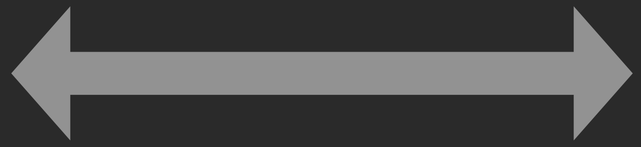
...

1. BLEU: A Method for Automatic Evaluation of Machine Translation —*Papinei et al.*—
2. Rouge: A package for automatic evaluation of summaries —*Lin.*—
3. Bertscore: Evaluating text generation with bert —*Zhang et al.*—

MODEL EVALUATION

GROUND TRUTH

```
Create a connection
to the consumer.
```



MODEL'S PREDICTION

```
Create a logger .
```

BLEU SCORE: 0.26

Reads the contents of this source as a string. PR

Get the textual information from this source and represent it as a string. GT

```
public String read() throws IOException {
  Closer closer = Closer.create();
  try {
     Reader reader = closer.register(openStream());
     return CharStreams.toString(reader);
  } catch ( Throwable e ) {
     throw closer.rethrow (e);
  } finally { closer.close(); }
}
```

Reads the contents of this source as a string.	PR

Get the textual information from this source and represent it as a string.	GT

```
public String read() throws IOException {
  Closer closer = Closer.create();
  try {
     Reader reader = closer.register(openStream());
     return CharStreams.toString(reader);
  } catch ( Throwable e ) {
     throw closer.rethrow (e);
  } finally { closer.close(); }
}
```

SEMANTICALLY EQUIVALENT CODE SUMMARIES

Reads the contents of this source as a string.	PR

Get the textual information from this source and represent it as a string.	GT

```
public String read() throws IOException {
  Closer closer = Closer.create();
  try {
     Reader reader = closer.register(openStream());
     return CharStreams.toString(reader);
  } catch ( Throwable e ) {
     throw closer.rethrow (e);
  } finally { closer.close(); }
}
```

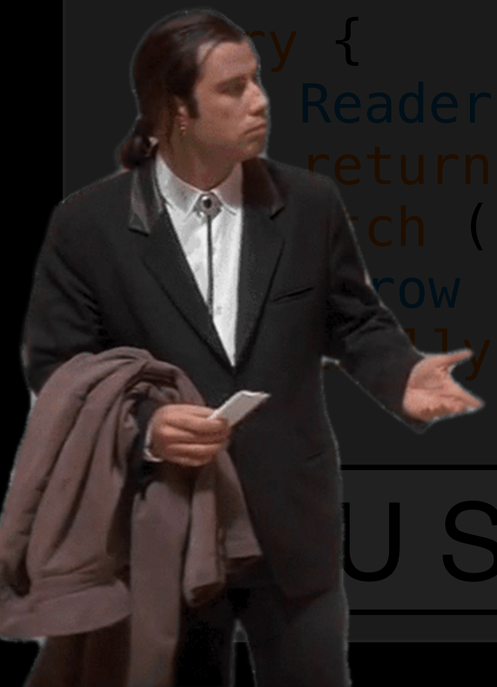
BLEU SCORE: 0.21

SEMANTICALLY EQUIVALENT CODE SUMMARIES

Reads the contents of this source as a string. | PR

Get the textual information from this source and represent it as a string. | GT

```
public String read() throws IOException {
  Closer closer = Closer.create();
  try {
    Reader reader = closer.register(openStream());
    return CharStreams.toString(reader);
  } catch ( Throwable e ) {
    throw closer.rethrow (e);
  } finally { closer.close(); }
```



U SCORE: 0.21

SEMANTICALLY EQUIVALENT CODE SUMMARIES

https://www.unsplash.com

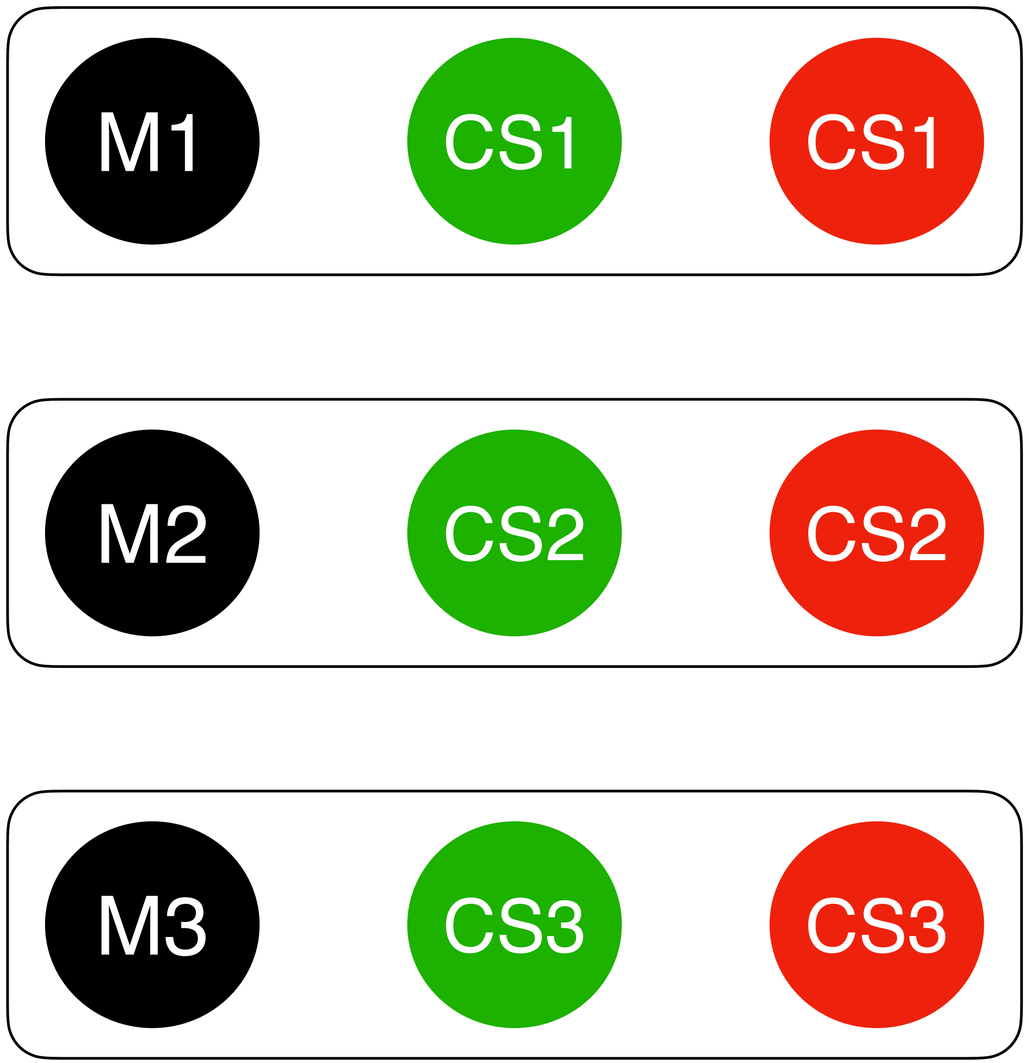
SIDE (Summary alIgnment to coDe sEmantic)

Get the textual information from this source and represent it as a string. ✓

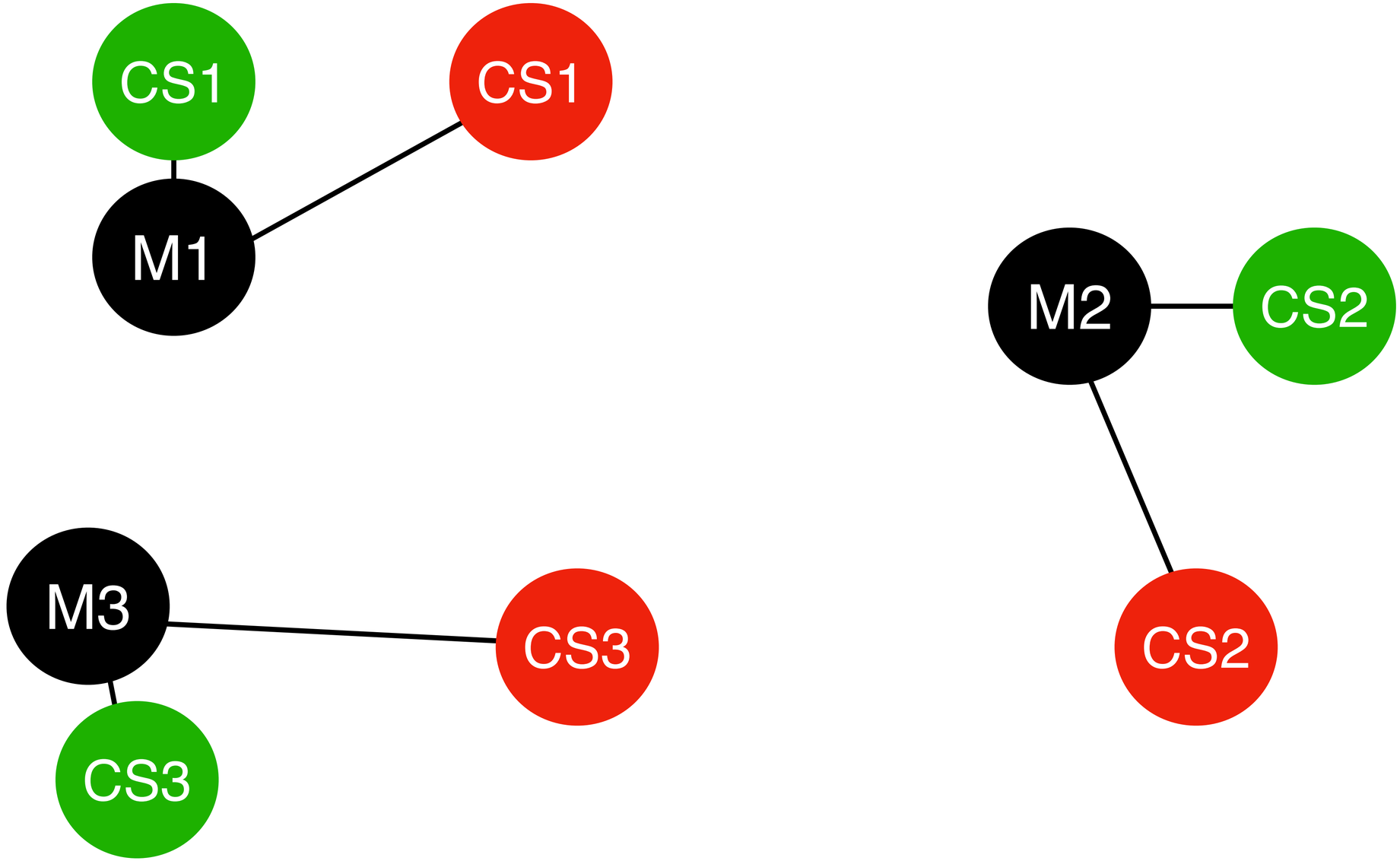
Connect to the server and return the status ✗

```
public String read() throws IOException {
  Closer closer = Closer.create();
  try {
     Reader reader = closer.register(openStream());
     return CharStreams.toString(reader);
  } catch ( Throwable e ) {
     throw closer.rethrow (e);
  } finally { closer.close(); }
}
```

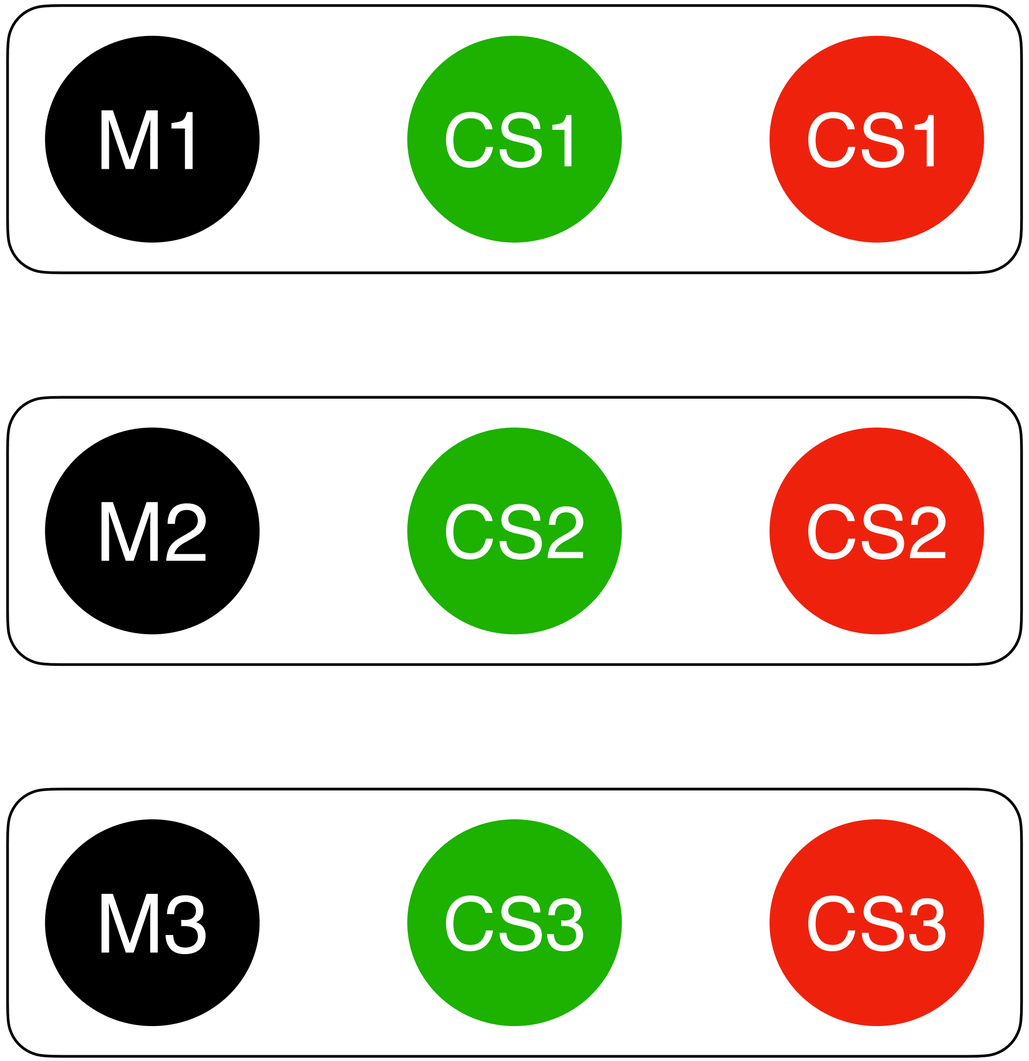
A Contrastive Learning-based Method



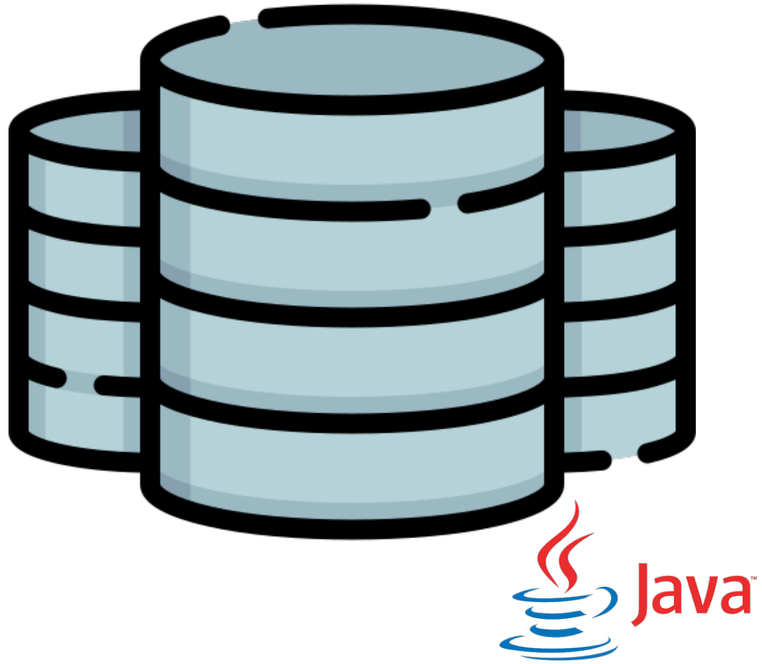
A Contrastive Learning-based Method



A Contrastive Learning-based Method

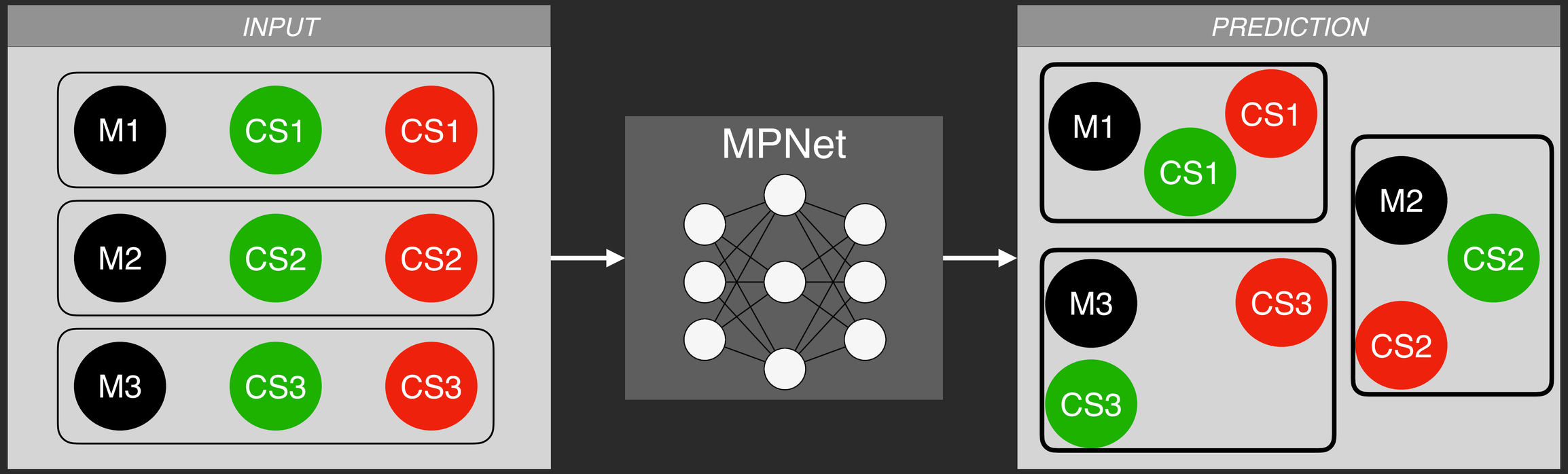


~180K



CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation —*Lu et al.*—

DL-Model Training



MPNet: Masked and permuted pre-training for language understanding

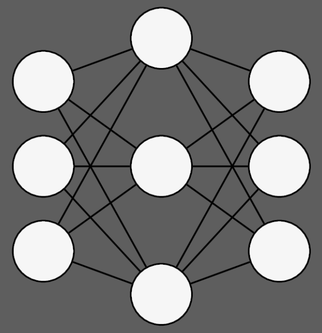
SIDE

INPUT

Create a connection to the consumer.

```
public ConnectionConsumer
        createConnectionConsumer
        …{
      if(LOGGER.isTraceEnabled())
      {
         ActiveMQRALogger.LOGGER.
            trace("Create
               connectionConsumer");
      }
      else {
         …
     }
}
```

MPNet



PREDICTION

0.81

MPNet: Masked and permuted pre-training for language understanding

Evaluation of SIDE

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fail

Evaluation of SIDE

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fail

Whether SIDE allows for a more effective representations of developers’ evaluations regarding the quality of automatically generated summaries

Evaluation of SIDE

Reassessing Automatic Evaluation Metrics for Code Summarization Tasks

Devjeet Roy
devjeet.roy@wsu.edu
Washington State University
Pullman, WA, USA

Sarah Fakhoury
sarah.fakhoury@wsu.edu
Washington State University
Pullman, WA, USA

Venera Arnaoudova
venera.arnaoudova@wsu.edu
Washington State University
Pullman, WA, USA

ABSTRACT

In recent years, research in the domain of source code summarization has adopted data-driven techniques pioneered in machine translation (MT). Automatic evaluation metrics such as BLEU, METEOR, and ROUGE, are fundamental to the evaluation of MT systems and have been adopted as proxies of human evaluation in the code summarization domain. However, the extent to which automatic metrics agree with the gold standard of human evaluation has not been evaluated on code summarization tasks. Despite this, marginal improvements in metric scores are often used to discriminate between the performance of competing summarization models.

In this paper, we present a critical exploration of the applicability and interpretation of automatic metrics as evaluation techniques for code summarization tasks. We conduct an empirical study with 226 human annotators to assess the degree to which automatic metrics reflect human evaluation. Results indicate that metric improvements of less than 2 points do not guarantee systematic improvements in summarization quality, and are unreliable as proxies of human evaluation. When the difference between metric scores for two summarization approaches increases but remains within 5 points, some metrics such as METEOR and chrF become highly reliable proxies, whereas others, such as corpus BLEU, remain unreliable. Based on these findings, we make several recommendations for the use of automatic metrics to discriminate model performance in code summarization.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **General and reference → Metrics**; **Evaluation**.

KEYWORDS

automatic evaluation metrics, code summarization, machine translation

ACM Reference Format:
Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3468264.3468588

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468588

1 INTRODUCTION

Useful source code comments play a vital role in program comprehension and other software maintenance activities [45, 58]. However, proper documentation comes at a cost and producing well-written comments requires a substantial amount of effort on the part of software developers. This is one of the motivating factors behind why source code summarization is a rapidly growing research area—at least 18 papers proposing or evaluating automated summarization approaches are published in 2020 [1, 2, 10, 16, 18, 20, 22, 24, 27, 32, 47, 52, 55, 56, 60, 61, 63, 66].

The earliest code summarization approaches are based on strong syntactic theories of comment structure, information retrieval, and textual templates. These approaches typically evaluate the quality of a generated summary using human annotators who rate summaries on metrics such as as: content adequacy [36], conciseness [23, 37], and fluency [37, 38, 50].

In the last 5 years, the solution space for code summarization has significantly shifted towards the widespread adoption of techniques and evaluation metrics from the Machine Translation (MT) domain. Prior to 2015, virtually all summarization approaches involved template or information retrieval based techniques. In 2015, the first paper using an MT approach was published at an SE conference [39], and, to the best of our knowledge, there are 35 papers thus far that propose an approach, an evaluation, or a critique of MT summarization approaches [2–5, 8–10, 12, 16–21, 23–25, 27, 28, 31, 35, 39, 47, 48, 53–57, 59–62, 65, 66]. Note that 7 of those are published in 2019 and 13 in 2020 alone.

One of the primary reasons for this shift is the idea that translating source code to its natural language equivalent holds many parallels with the concept of translating one natural language to another [23]. A side effect of this shift is a substantial change in the amount of data needed to evaluate code summarization approaches; training generative models requires a large amount of data. Naturally, the evaluation techniques used must also scale, and automated metrics provide an efficient way to evaluate the quality of generated summaries en mass.

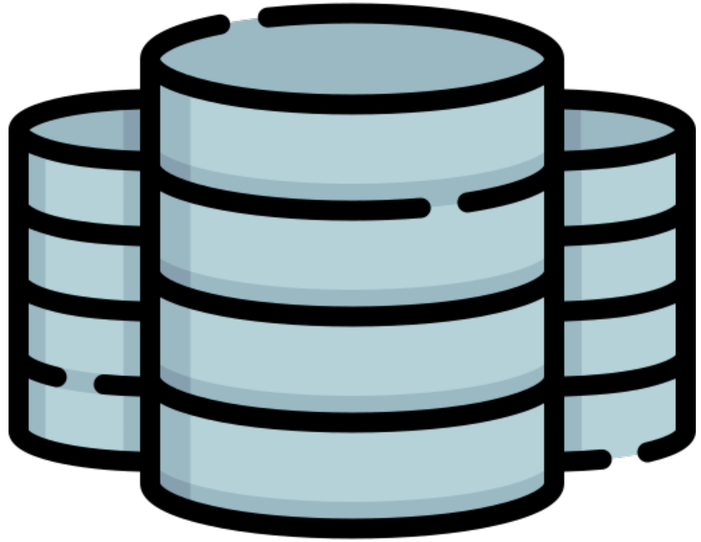
Automated metrics designed to evaluate natural language translation approaches, such as BLEU [40], METEOR [6], and ROUGE [30], have been adopted by code summarization researchers where a generated summary is compared to a 'gold standard' or 'reference' summary. In the context of leading comment summaries, the reference summary is often the original accompanying comment written by a developer.

In the MT domain, BLEU has long been accepted as a de-facto best-practice metric. Recently, however, prominent machine translation researchers have raised concern over the use of BLEU [34, 42, 43], warning the MT community that the way BLEU is used

FSE'2021

Roy et al.

~5K



Evaluation of SIDE

Reassessing Automatic Evaluation Metrics for Code Summarization Tasks

Devjeet Roy
devjeet.roy@wsu.edu
Washington State University
Pullman, WA, USA

Sarah Fakhoury
sarah.fakhoury@wsu.edu
Washington State University
Pullman, WA, USA

Venera Arnaoudova
venera.arnaoudova@wsu.edu
Washington State University
Pullman, WA, USA

ABSTRACT

In recent years, research in the domain of source code summarization has adopted data-driven techniques pioneered in machine translation (MT). Automatic evaluation metrics such as BLEU, METEOR, and ROUGE, are fundamental to the evaluation of MT systems and have been adopted as proxies of human evaluation in the code summarization domain. However, the extent to which automatic metrics agree with the gold standard of human evaluation has not been evaluated on code summarization tasks. Despite this, marginal improvements in metric scores are often used to discriminate between the performance of competing summarization models.

In this paper, we present a critical exploration of the applicability and interpretation of automatic metrics as evaluation techniques for code summarization tasks. We conduct an empirical study with 226 human annotators to assess the degree to which automatic metrics reflect human evaluation. Results indicate that metric improvements of less than 2 points do not guarantee systematic improvements in summarization quality, and are unreliable as proxies of human evaluation. When the difference between metric scores for two summarization approaches increases but remains within 5 points, some metrics such as METEOR and chrF become highly reliable proxies, whereas others, such as corpus BLEU, remain unreliable. Based on these findings, we make several recommendations for the use of automatic metrics to discriminate model performance in code summarization.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **General and reference → Metrics**; **Evaluation**.

KEYWORDS

automatic evaluation metrics, code summarization, machine translation

ACM Reference Format:
Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3468264.3468588

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468588

1 INTRODUCTION

Useful source code comments play a vital role in program comprehension and other software maintenance activities [45, 58]. However, proper documentation comes at a cost and producing well-written comments requires a substantial amount of effort on the part of software developers. This is one of the motivating factors behind why source code summarization is a rapidly growing research area—at least 18 papers proposing or evaluating automated summarization approaches are published in 2020 [1, 2, 10, 16, 18, 20, 22, 24, 27, 32, 47, 52, 55, 56, 60, 61, 63, 66].

The earliest code summarization approaches are based on strong syntactic theories of comment structure, information retrieval, and textual templates. These approaches typically evaluate the quality of a generated summary using human annotators who rate summaries on metrics such as as: content adequacy [36], conciseness [23, 37], and fluency [37, 38, 50].

In the last 5 years, the solution space for code summarization has significantly shifted towards the widespread adoption of techniques and evaluation metrics from the Machine Translation (MT) domain. Prior to 2015, virtually all summarization approaches involved template or information retrieval based techniques. In 2015, the first paper using an MT approach was published at an SE conference [39], and, to the best of our knowledge, there are 35 papers thus far that propose an approach, an evaluation, or a critique of MT summarization approaches [2–5, 8–10, 12, 16–21, 23–25, 27, 28, 31, 35, 39, 47, 48, 53–57, 59–62, 65, 66]. Note that 7 of those are published in 2019 and 13 in 2020 alone.

One of the primary reasons for this shift is the idea that translating source code to its natural language equivalent holds many parallels with the concept of translating one natural language to another [23]. A side effect of this shift is a substantial change in the amount of data needed to evaluate code summarization approaches; training generative models requires a large amount of data. Naturally, the evaluation techniques used must also scale, and automated metrics provide an efficient way to evaluate the quality of generated summaries en mass.

Automated metrics designed to evaluate natural language translation approaches, such as BLEU [40], METEOR [6], and ROUGE [30], have been adopted by code summarization researchers where a generated summary is compared to a 'gold standard' or 'reference' summary. In the context of leading comment summaries, the reference summary is often the original accompanying comment written by a developer.

In the MT domain, BLEU has long been accepted as a de-facto best-practice metric. Recently, however, prominent machine translation researchers have raised concern over the use of BLEU [34, 42, 43], warning the MT community that the way BLEU is used

FSE'2021

Roy et al.

Conciseness: Assesses the degree to which the summary contains unnecessary information

Evaluation of SIDE

Reassessing Automatic Evaluation Metrics for Code Summarization Tasks

Devjeet Roy
devjeet.roy@wsu.edu
Washington State University
Pullman, WA, USA

Sarah Fakhoury
sarah.fakhoury@wsu.edu
Washington State University
Pullman, WA, USA

Venera Arnaoudova
venera.arnaoudova@wsu.edu
Washington State University
Pullman, WA, USA

ABSTRACT

In recent years, research in the domain of source code summarization has adopted data-driven techniques pioneered in machine translation (MT). Automatic evaluation metrics such as BLEU, METEOR, and ROUGE, are fundamental to the evaluation of MT systems and have been adopted as proxies of human evaluation in the code summarization domain. However, the extent to which automatic metrics agree with the gold standard of human evaluation has not been evaluated on code summarization tasks. Despite this, marginal improvements in metric scores are often used to discriminate between the performance of competing summarization models.

In this paper, we present a critical exploration of the applicability and interpretation of automatic metrics as evaluation techniques for code summarization tasks. We conduct an empirical study with 226 human annotators to assess the degree to which automatic metrics reflect human evaluation. Results indicate that metric improvements of less than 2 points do not guarantee systematic improvements in summarization quality, and are unreliable as proxies of human evaluation. When the difference between metric scores for two summarization approaches increases but remains within 5 points, some metrics such as METEOR and chrF become highly reliable proxies, whereas others, such as corpus BLEU, remain unreliable. Based on these findings, we make several recommendations for the use of automatic metrics to discriminate model performance in code summarization.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **General and reference → Metrics**; **Evaluation**.

KEYWORDS

automatic evaluation metrics, code summarization, machine translation

ACM Reference Format:
Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3468264.3468588

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468588

1 INTRODUCTION

Useful source code comments play a vital role in program comprehension and other software maintenance activities [45, 58]. However, proper documentation comes at a cost and producing well-written comments requires a substantial amount of effort on the part of software developers. This is one of the motivating factors behind why source code summarization is a rapidly growing research area—at least 18 papers proposing or evaluating automated summarization approaches are published in 2020 [1, 2, 10, 16, 18, 20, 22, 24, 27, 32, 47, 52, 55, 56, 60, 61, 63, 66].

The earliest code summarization approaches are based on strong syntactic theories of comment structure, information retrieval, and textual templates. These approaches typically evaluate the quality of a generated summary using human annotators who rate summaries on metrics such as as: content adequacy [36], conciseness [23, 37], and fluency [37, 38, 50].

In the last 5 years, the solution space for code summarization has significantly shifted towards the widespread adoption of techniques and evaluation metrics from the Machine Translation (MT) domain. Prior to 2015, virtually all summarization approaches involved template or information retrieval based techniques. In 2015, the first paper using an MT approach was published at an SE conference [39], and, to the best of our knowledge, there are 35 papers thus far that propose an approach, an evaluation, or a critique of MT summarization approaches [2–5, 8–10, 12, 16–21, 23–25, 27, 28, 31, 35, 39, 47, 48, 53–57, 59–62, 65, 66]. Note that 7 of those are published in 2019 and 13 in 2020 alone.

One of the primary reasons for this shift is the idea that translating source code to its natural language equivalent holds many parallels with the concept of translating one natural language to another [23]. A side effect of this shift is a substantial change in the amount of data needed to evaluate code summarization approaches; training generative models requires a large amount of data. Naturally, the evaluation techniques used must also scale, and automated metrics provide an efficient way to evaluate the quality of generated summaries en mass.

Automated metrics designed to evaluate natural language translation approaches, such as BLEU [40], METEOR [6], and ROUGE [30], have been adopted by code summarization researchers where a generated summary is compared to a 'gold standard' or 'reference' summary. In the context of leading comment summaries, the reference summary is often the original accompanying comment written by a developer.

In the MT domain, BLEU has long been accepted as a de-facto best-practice metric. Recently, however, prominent machine translation researchers have raised concern over the use of BLEU [34, 42, 43], warning the MT community that the way BLEU is used

FSE'2021

Roy et al.

Conciseness: Assesses the degree to which the summary contains unnecessary information

Fluency: Evaluates the "smoothness" rate in the generated summary

Evaluation of SIDE

Reassessing Automatic Evaluation Metrics for Code Summarization Tasks

Devjeet Roy
devjeet.roy@wsu.edu
Washington State University
Pullman, WA, USA

Sarah Fakhoury
sarah.fakhoury@wsu.edu
Washington State University
Pullman, WA, USA

Venera Arnaoudova
venera.arnaoudova@wsu.edu
Washington State University
Pullman, WA, USA

ABSTRACT

In recent years, research in the domain of source code summarization has adopted data-driven techniques pioneered in machine translation (MT). Automatic evaluation metrics such as BLEU, METEOR, and ROUGE, are fundamental to the evaluation of MT systems and have been adopted as proxies of human evaluation in the code summarization domain. However, the extent to which automatic metrics agree with the gold standard of human evaluation has not been evaluated on code summarization tasks. Despite this, marginal improvements in metric scores are often used to discriminate between the performance of competing summarization models.

In this paper, we present a critical exploration of the applicability and interpretation of automatic metrics as evaluation techniques for code summarization tasks. We conduct an empirical study with 226 human annotators to assess the degree to which automatic metrics reflect human evaluation. Results indicate that metric improvements of less than 2 points do not guarantee systematic improvements in summarization quality, and are unreliable as proxies of human evaluation. When the difference between metric scores for two summarization approaches increases but remains within 5 points, some metrics such as METEOR and chrF become highly reliable proxies, whereas others, such as corpus BLEU, remain unreliable. Based on these findings, we make several recommendations for the use of automatic metrics to discriminate model performance in code summarization.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; • **General and reference → Metrics**; **Evaluation**.

KEYWORDS

automatic evaluation metrics, code summarization, machine translation

ACM Reference Format:

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468588

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468588

1 INTRODUCTION

Useful source code comments play a vital role in program comprehension and other software maintenance activities [45, 58]. However, proper documentation comes at a cost and producing well-written comments requires a substantial amount of effort on the part of software developers. This is one of the motivating factors behind why source code summarization is a rapidly growing research area—at least 18 papers proposing or evaluating automated summarization approaches are published in 2020 [1, 2, 10, 16, 18, 20, 22, 24, 27, 32, 47, 52, 55, 56, 60, 61, 63, 66].

The earliest code summarization approaches are based on strong syntactic theories of comment structure, information retrieval, and textual templates. These approaches typically evaluate the quality of a generated summary using human annotators who rate summaries on metrics such as as: content adequacy [36], conciseness [23, 37], and fluency [37, 38, 50].

In the last 5 years, the solution space for code summarization has significantly shifted towards the widespread adoption of techniques and evaluation metrics from the Machine Translation (MT) domain. Prior to 2015, virtually all summarization approaches involved template or information retrieval based techniques. In 2015, the first paper using an MT approach was published at an SE conference [39], and, to the best of our knowledge, there are 35 papers thus far that propose an approach, an evaluation, or a critique of MT summarization approaches [2–5, 8–10, 12, 16–21, 23–25, 27, 28, 31, 35, 39, 47, 48, 53–57, 59–62, 65, 66]. Note that 7 of those are published in 2019 and 13 in 2020 alone.

One of the primary reasons for this shift is the idea that translating source code to its natural language equivalent holds many parallels with the concept of translating one natural language to another [23]. A side effect of this shift is a substantial change in the amount of data needed to evaluate code summarization approaches; training generative models requires a large amount of data. Naturally, the evaluation techniques used must also scale, and automated metrics provide an efficient way to evaluate the quality of generated summaries en mass.

Automated metrics designed to evaluate natural language translation approaches, such as BLEU [40], METEOR [6], and ROUGE [30], have been adopted by code summarization researchers where a generated summary is compared to a 'gold standard' or 'reference' summary. In the context of leading comment summaries, the reference summary is often the original accompanying comment written by a developer.

In the MT domain, BLEU has long been accepted as a de-facto best-practice metric. Recently, however, prominent machine translation researchers have raised concern over the use of BLEU [34, 42, 43], warning the MT community that the way BLEU is used

FSE'2021

Roy et al.

Conciseness: Assesses the degree to which the summary contains unnecessary information

Fluency: Evaluates the "smoothness" rate in the generated summary

Content Adequacy: Assess the extent to which the summary lacks information needed to understand the code

Evaluation of SIDE

Reassessing Automatic Evaluation Metrics for Code Summarization Tasks

Devjeet Roy
devjeet.roy@wsu.edu
Washington State University
Pullman, WA, USA

Sarah Fakhoury
sarah.fakhoury@wsu.edu
Washington State University
Pullman, WA, USA

Venera Arnaoudova
venera.arnaoudova@wsu.edu
Washington State University
Pullman, WA, USA

ABSTRACT

In recent years, research in the domain of source code summarization has adopted data-driven techniques pioneered in machine translation (MT). Automatic evaluation metrics such as BLEU, METEOR, and ROUGE, are fundamental to the evaluation of MT systems and have been adopted as proxies of human evaluation in the code summarization domain. However, the extent to which automatic metrics agree with the gold standard of human evaluation has not been evaluated on code summarization tasks. Despite this, marginal improvements in metric scores are often used to discriminate between the performance of competing summarization models.

In this paper, we present a critical exploration of the applicability and interpretation of automatic metrics as evaluation techniques for code summarization tasks. We conduct an empirical study with 226 human annotators to assess the degree to which automatic metrics reflect human evaluation. Results indicate that metric improvements of less than 2 points do not guarantee systematic improvements in summarization quality, and are unreliable as proxies of human evaluation. When the difference between metric scores for two summarization approaches increases but remains within 5 points, some metrics such as METEOR and chrF become highly reliable proxies, whereas others, such as corpus BLEU, remain unreliable. Based on these findings, we make several recommendations for the use of automatic metrics to discriminate model performance in code summarization.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; • **General and reference → Metrics**; **Evaluation**.

KEYWORDS

automatic evaluation metrics, code summarization, machine translation

ACM Reference Format:

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468588

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468588

1 INTRODUCTION

Useful source code comments play a vital role in program comprehension and other software maintenance activities [45, 58]. However, proper documentation comes at a cost and producing well-written comments requires a substantial amount of effort on the part of software developers. This is one of the motivating factors behind why source code summarization is a rapidly growing research area—at least 18 papers proposing or evaluating automated summarization approaches are published in 2020 [1, 2, 10, 16, 18, 20, 22, 24, 27, 32, 47, 52, 55, 56, 60, 61, 63, 66].

The earliest code summarization approaches are based on strong syntactic theories of comment structure, information retrieval, and textual templates. These approaches typically evaluate the quality of a generated summary using human annotators who rate summaries on metrics such as as: content adequacy [36], conciseness [23, 37], and fluency [37, 38, 50].

In the last 5 years, the solution space for code summarization has significantly shifted towards the widespread adoption of techniques and evaluation metrics from the Machine Translation (MT) domain. Prior to 2015, virtually all summarization approaches involved template or information retrieval based techniques. In 2015, the first paper using an MT approach was published at an SE conference [39], and, to the best of our knowledge, there are 35 papers thus far that propose an approach, an evaluation, or a critique of MT summarization approaches [2–5, 8–10, 12, 16–21, 23–25, 27, 28, 31, 35, 39, 47, 48, 53–57, 59–62, 65, 66]. Note that 7 of those are published in 2019 and 13 in 2020 alone.

One of the primary reasons for this shift is the idea that translating source code to its natural language equivalent holds many parallels with the concept of translating one natural language to another [23]. A side effect of this shift is a substantial change in the amount of data needed to evaluate code summarization approaches; training generative models requires a large amount of data. Naturally, the evaluation techniques used must also scale, and automated metrics provide an efficient way to evaluate the quality of generated summaries en mass.

Automated metrics designed to evaluate natural language translation approaches, such as BLEU [40], METEOR [6], and ROUGE [30], have been adopted by code summarization researchers where a generated summary is compared to a 'gold standard' or 'reference' summary. In the context of leading comment summaries, the reference summary is often the original accompanying comment written by a developer.

In the MT domain, BLEU has long been accepted as a de-facto best-practice metric. Recently, however, prominent machine translation researchers have raised concern over the use of BLEU [34, 42, 43], warning the MT community that the way BLEU is used

FSE'2021

Roy et al.

Conciseness: Assesses the degree to which the summary contains unnecessary information

Fluency: Evaluates the "smoothness" rate in the generated summary

Content Adequacy: Assess the extent to which the summary lacks information needed to understand the code

{0…5}

https://www.unsplash.com

Dependent Variables

Concisene

Fluenc

Content

https://www.unsplash.com

Independent Variables

SIDE

Word/Overlap-based

Embedding-based

https://www.unsplash.com

Independent Variables

SIDE

Word/Overlap-based

Embedding-based

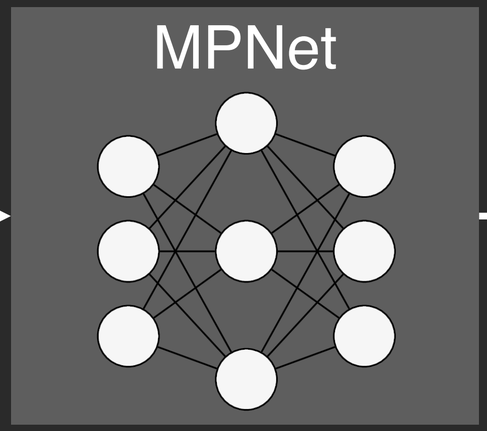

1
insert an arbitrary number of days
add n days to this date
P
G
BLEU SCORE
ROUGE SCORE
BERT SCORE
.....
2
add a new tab with a canonical …
add a new tab with a default …
P
G
BLEU SCORE
ROUGE SCORE
BERT SCORE
.....
3
create a new result set supporting …
implement a new result set …
P
G
BLEU SCORE
ROUGE SCORE
BERT SCORE
.....
4
constructs an instance from…
constructs an instance from…
P
G
BLEU SCORE
ROUGE SCORE
BERT SCORE
.....
~5K

SIDE

INPUT

```
Create a connection to the consumer.

public ConnectionConsumer
        createConnectionConsumer
        …{
      if(LOGGER.isTraceEnabled())
      {
         ActiveMQRALogger.LOGGER.
            trace("Create
               connectionConsumer");
      }
      else {
         …
   }
}
```



PREDICTION

0.81

MPNet: Masked and permuted pre-training for language understanding

~5K

1

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fails

1

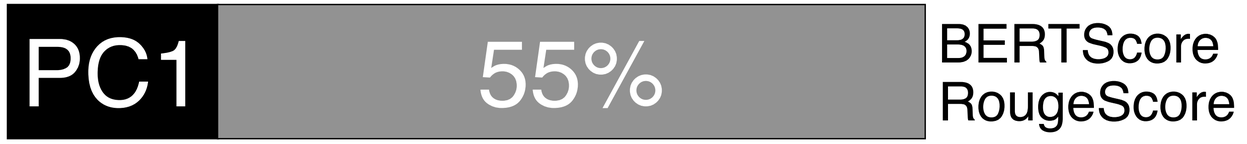
RINCIPAL COMPONENT ANALYSIS (PCA

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fails

1

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fails

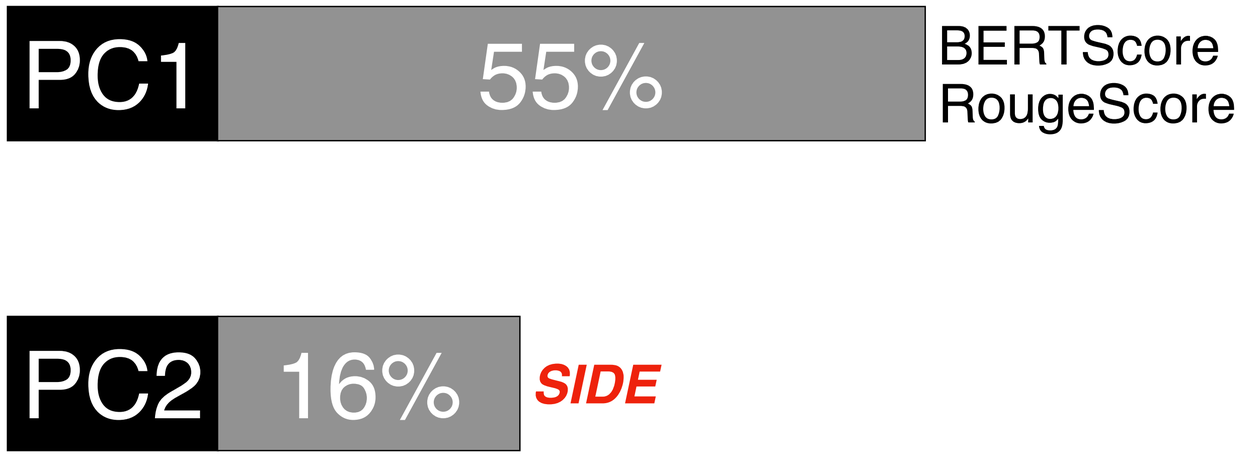
RINCIPAL COMPONENT ANALYSIS (PCA



1

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fails

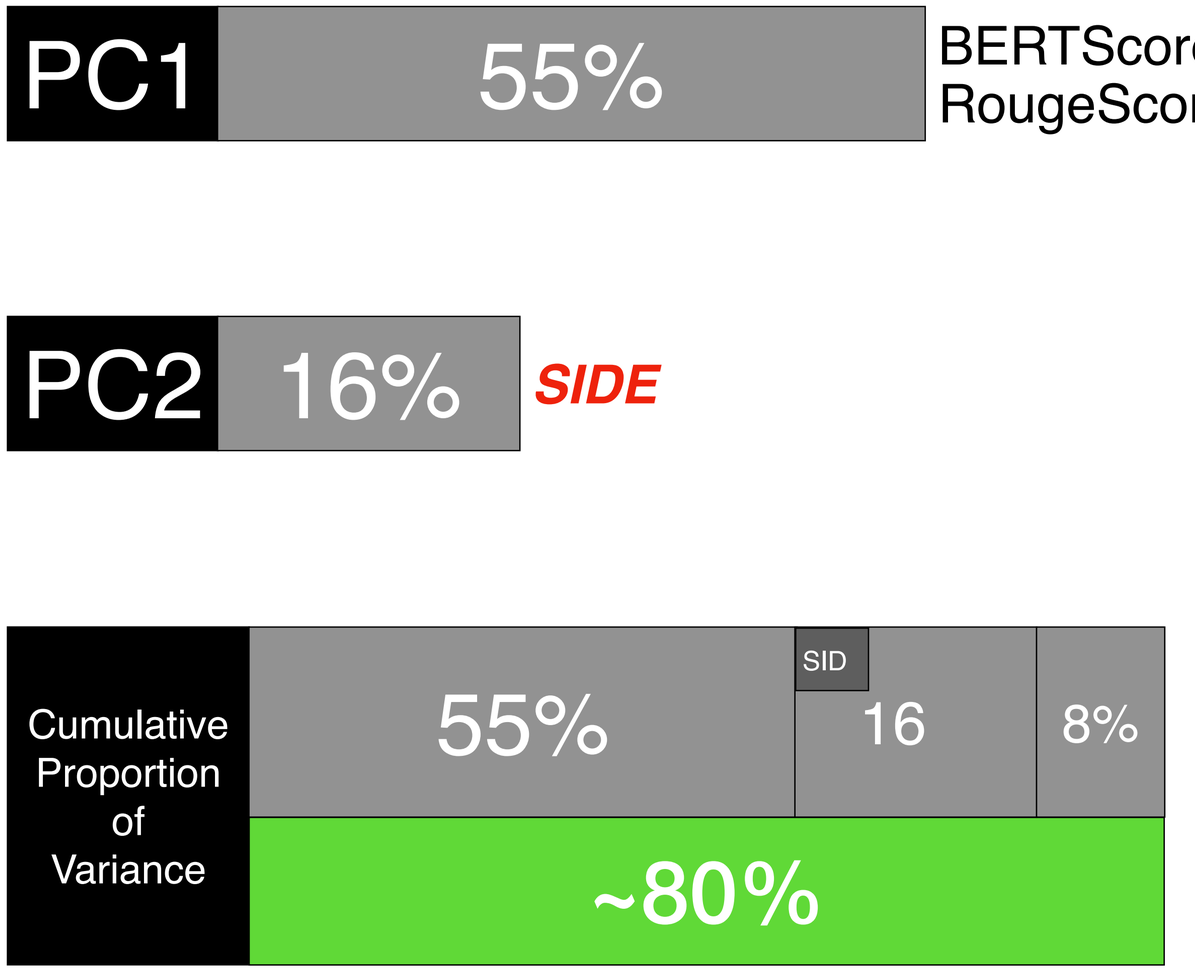
RINCIPAL COMPONENT ANALYSIS (PCA



1

Whether SIDE can be effectively used to capture new “dimensions” where state-of-the-art metrics fails

RINCIPAL COMPONENT ANALYSIS (PCA



2

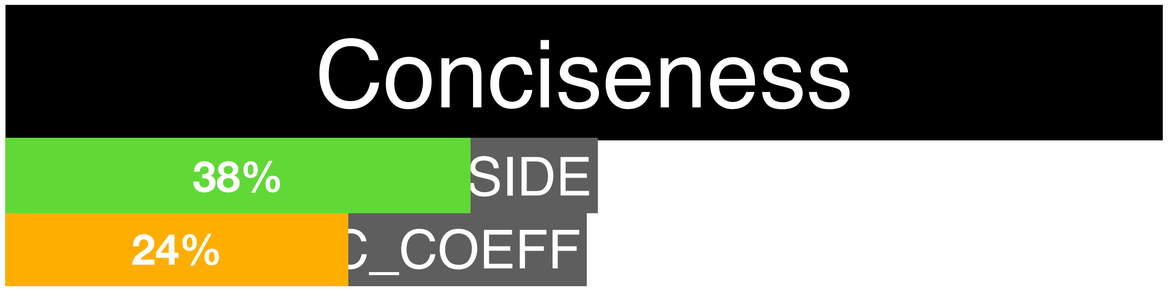
RDERED LOGISTIC REGRESSION MOD

Whether SIDE allows for a more effective representations of developers' evaluations regarding the quality of automatically generated summaries

2

Whether SIDE allows for a more effective representations of developers' evaluations regarding the quality of automatically generated summaries

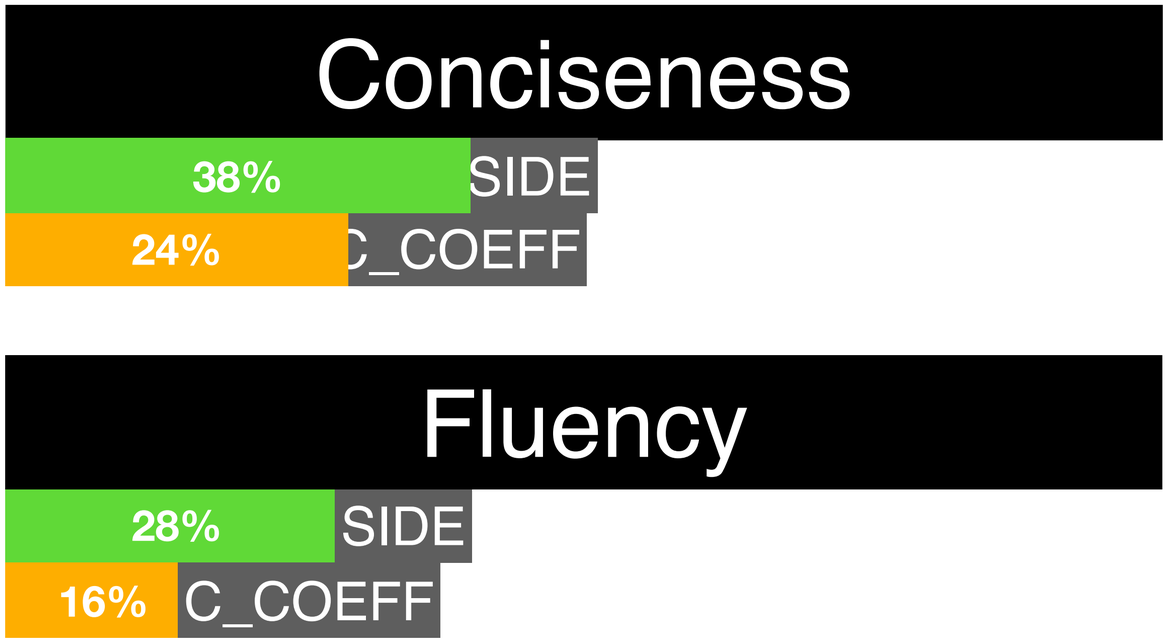
RDERED LOGISTIC REGRESSION MOD



2

Whether SIDE allows for a more effective representations of developers' evaluations regarding the quality of automatically generated summaries

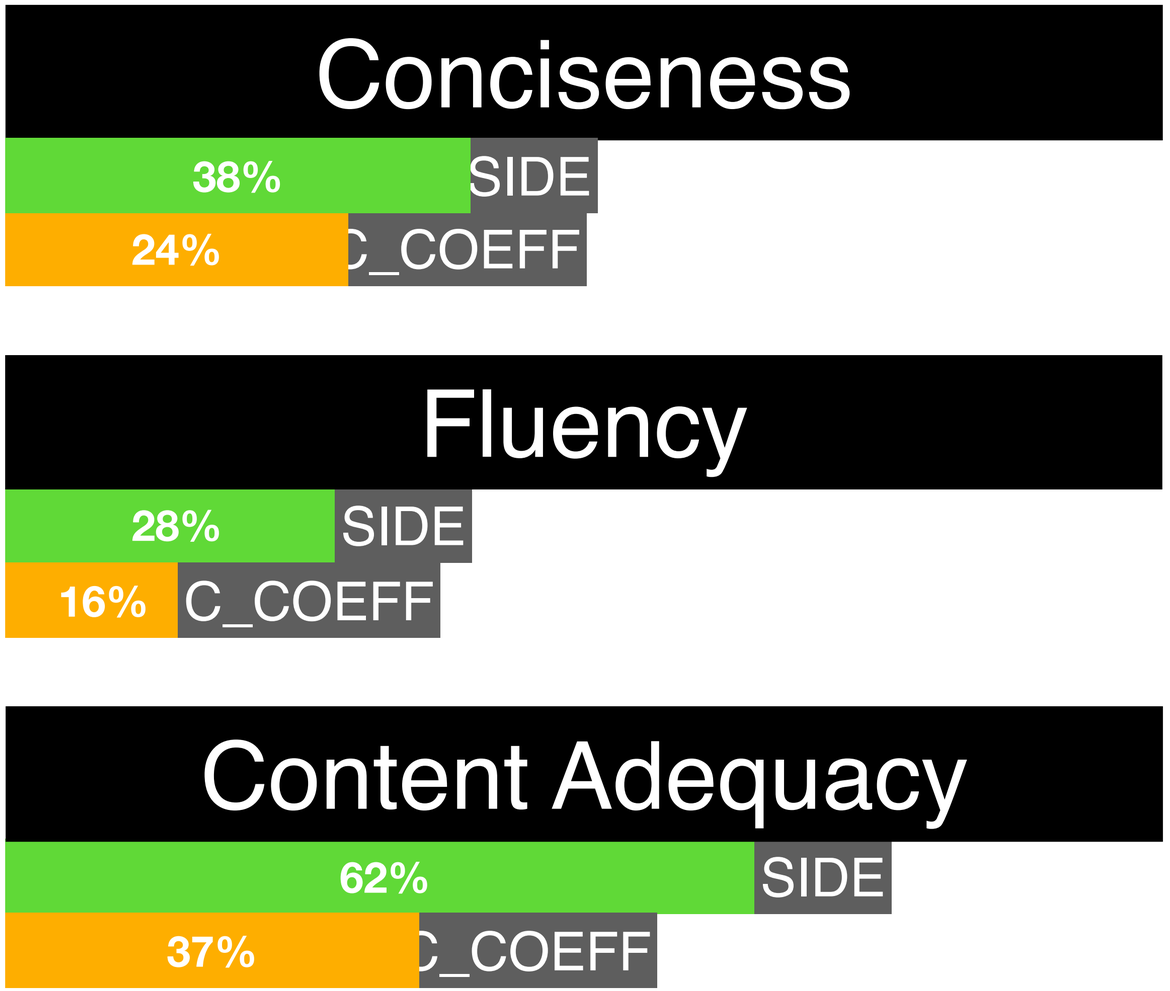
RDERED LOGISTIC REGRESSION MOD



2

Whether SIDE allows for a more effective representations of developers' evaluations regarding the quality of automatically generated summaries

RDERED LOGISTIC REGRESSION MOD



METHOD

1

```
public Element asElement() {
    return this.component.createCopy();
}
```

REFERENCE SUMMARY

not yet documented

GENERATED SUMMARY

returns a copy of this component as an element

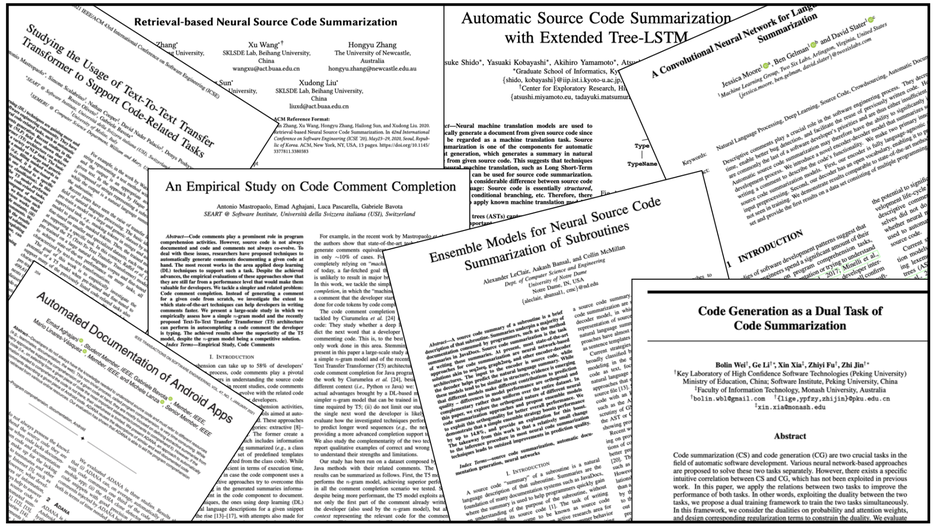
SCORES

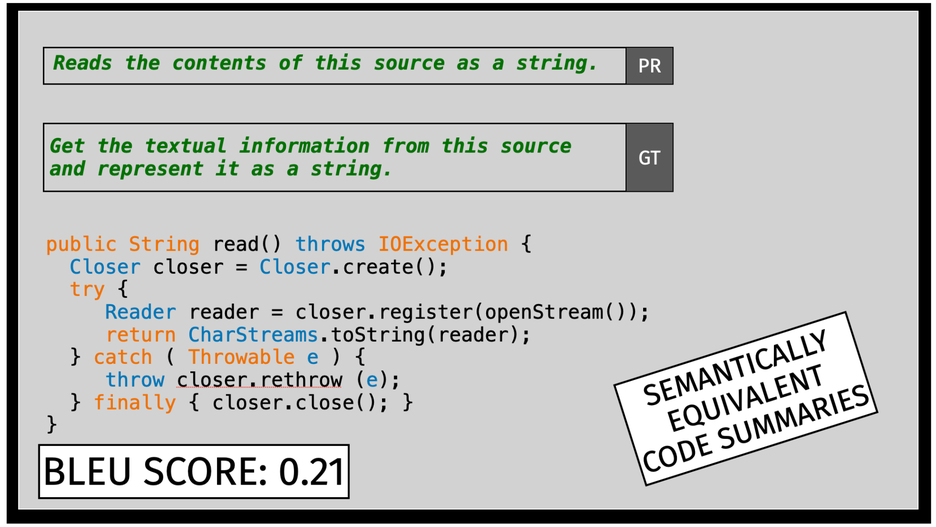
			SOTA ↓				OUR ↑		HUMAN ↑		
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.34	*0.00*	*0.08*	*0.43*	*0.00*	*0.00*	*0.00*	*0.91*	*88*	*4*	*4*	*5*

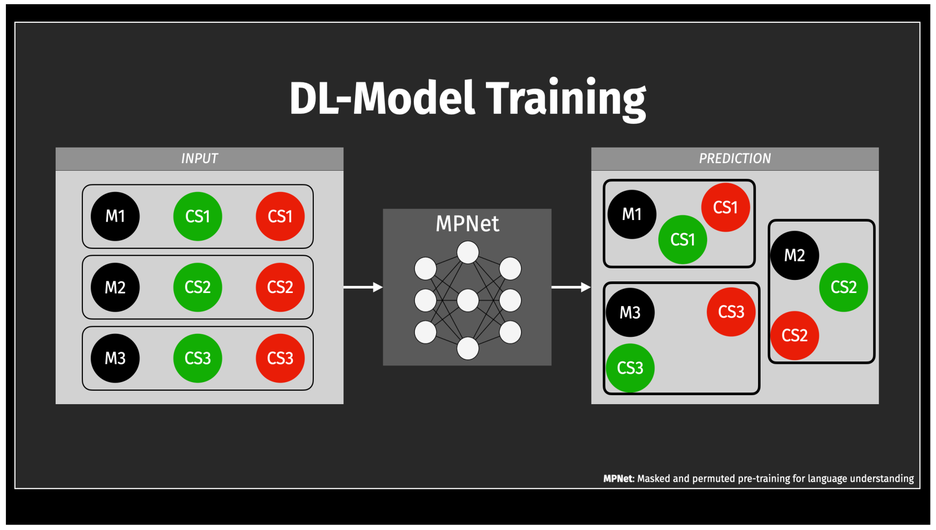
State-of-the-art metrics are not sufficient to describe humans' assessment of automatically generated code summaries

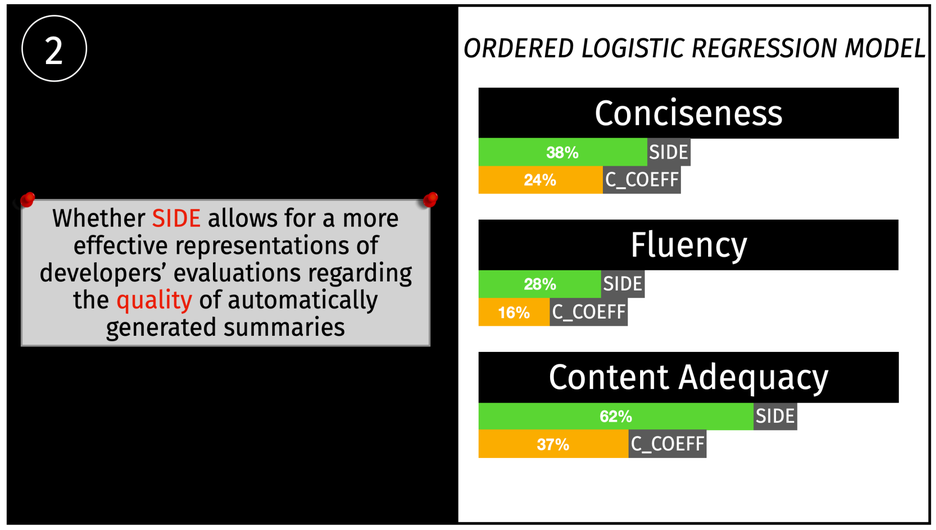
State-of-the-art metrics are not sufficient to describe humans' assessment of automatically generated code summaries

A more comprehensive evaluation framework which takes into account more traditional metrics and *SIDE*-like metrics are needed to capture more dimensions









https://github.com/side-metric/summarization

SEART @ USI — Università della Svizzera italiana — Software Institute

Software Institute Seminars @USI 2023-2024

Stay in touch!

antonio.mastropaolo@usi.ch

antoniomastropaolo.com

AntonioMastro2