On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot

***Abstract*—Software engineering research has always being concerned with the improvement of code completion approaches, which suggest the next tokens a developer will likely type while coding. The recent release of GitHub Copilot constitutes a big step forward, also because of its unprecedented ability to automatically generate even entire functions from their natural language description. While the usefulness of Copilot is evident, it is still unclear to what extent it is robust. Specifically, we do not now the extent to which semantic-preserving changes in the natural language description provided to the model have an effect on the generated code function. In this paper we present an empirical study in which we aim at understanding whether *different but semantically equivalent natural language descriptions result in the same recommended function*. A negative answer would pose questions on the robustness of deep learning (DL)-based code generators since it would imply that developers using different wordings to describe the same code would obtain different recommendations. We asked Copilot to automatically generate 892 Java methods starting from their original Javadoc description. Then, we generated different semantically equivalent descriptions for each method both manually and automatically, and we analyzed the extent to which predictions generated by Copilot changed. Our results show that modifying the description results in different code recommendations in $\sim$46% of cases. Also, differences in the semantically equivalent descriptions might significantly impact the correctness of the generated code ($\pm$28%).**

***Index Terms*—Empirical Study, Recommender Systems**

I. Introduction

One of the long lasting dreams in software engineering research is the automated generation of source code. Towards this goal, several approaches have been proposed. The first attempts targeted the relatively simpler problem of code completion, that has been tackled exploiting historical information [48], coding patterns mined from software repositories [20], [40], [54], [9], [39], [43], [18] and, more recently, Deep Learning (DL) models [60], [25], [27], [7], [51], [14].

The recent release of *GitHub Copilot* [13] pushed the capabilities of these tools to whole new levels. The large-scale training performed on the OpenAI's Codex model (159Gb of data from public GitHub repositories) allows Copilot to not limit its recommendations to few code tokens/statements the developer is likely to write: Copilot is able to automatically synthesize entire functions just starting from their signature and natural language descriptions.

This new generation of source code recommender systems has the potential to change the way in which developers write code [17] and comes with a number of open questions concerning how to effectively exploit them to maximize developers' productivity.

Intuitively, the ability of the developer to provide "proper" inputs to the model will become central to boost the effectiveness of its recommendations. In the concrete example of GitHub Copilot, the natural language description provided to the model to automatically generate a code function could substantially influence the model output. This means that two developers providing different natural language descriptions for the same function they would like to automatically generate could receive two different recommendations. While this would be fine in case the two descriptions are actually different in the semantics of what they describe, receiving different recommendations for *semantically equivalent natural language descriptions* would pose questions on the robustness and usability of DL-based code recommenders.

This is the main research question we investigate in this paper: We study the extent to which different semantically equivalent natural language descriptions of a function result in different recommendations (*i.e.,* different synthesized functions) by *GitHub Copilot* . The latter is selected as representative of DL-based code recommenders since it is the *de facto* state-of-the-art tool when it comes to code generation.

We collected from an initial set of 1,401 open source projects a set of 892 Java methods that are (i) accompanied by a Doc Comment for the Javadoc tool, and (ii) exercised by a test suite written by the project's contributors. Then, as done in the literature [21], [30], we considered the first sentence of the Doc Comments as a "natural language description" of the method. We refer to the extracted sentence as the "*original*" description.

We preliminarily checked whether existing automated paraphrasing techniques are suitable for robustness testing, *i.e.,* if they can be used to create semantically equivalent descriptions of the methods to generate. We validated two state-of-the-art approaches in this scenario: PEGASUS [64], a DL-based paraphrasing tool, and Translation Pivoting (TP), a heuristic-based approach. We used both techniques to generate a paraphrase for each *original* description in our dataset. Then, we manually inspected the obtained paraphrases and classified them as semantically equivalent or not. We obtained positive results for both the approaches, with TP being the best performing one with 77% of valid paraphrases.

Then, to answer our main research question, we generated different paraphrases for each *original* description: We used the two previously described automated approaches, *i.e.,* PEGASUS and TP, and we also manually generated paraphrases by distributing the original descriptions among four of the authors, each of which was in charge of paraphrasing a subset of them.

Therefore, for each *original* description, we obtained a set of semantically equivalent *paraphrased* descriptions. We provided both the *original* and the *paraphrased* descriptions as input to *Copilot*, asking it to generate the corresponding method body. We analyze the percentage of cases in which the *paraphrased* descriptions result in a different code prediction as compared to the *original* one, with a particular focus on the impact on the prediction quality, *e.g.,* cases in which the *original* description resulted in the recommendation of a method passing its associated test cases while switching to a *paraphrased* description made *Copilot* recommending a method failing its related tests.

Our results show that paraphrasing a description results in a change in the code recommendation in ∼46% of cases. The resulting changes also cause substantial variations in the percentage of correct predictions. Such findings indicate the central role played by the model's input in the code recommendation and the need for testing and improving the robustness of DL-based code generators.

Data and code used in our study are publicly available [5].

II. Study Design

The *goal* of our study is to understand how robust is a state-of-the-art DL-based code completion approach (*i.e., GitHub Copilot*). We aim at answering the following research questions (RQs):

RQ_0: To what extent can automated paraphrasing techniques be used to test the robustness of DL-based code generators? Not always natural language processing techniques can be used out of the box on software-related text [33]. Therefore, with this preliminary RQ, we want to understand whether existing automated techniques for generating natural language paraphrases are suitable for SE task at hand (*i.e.,* paraphrasing a function description).

RQ_1: To what extent is the output of GitHub Copilot influenced by the code description provided as input by the developer? This RQ aims at understanding whether *Copilot*, as a representative of DL-based code generators, is likely to generate different recommendations for different semantically equivalent natural language descriptions provided as input.

In the following we detail the context for our study (Section II-A) and how we collected (Section II-B) and analyzed (Section II-C) the data needed to answer our RQs.

A. Context Selection

The context of our study is represented by 892 Java methods collected through the following process. We selected all GitHub Java repositories having at least 300 commits, 50 contributors, and 25 stars. These filters have been used in an attempt to exclude personal/toy projects. We also excluded forked projects to avoid duplicates. The decision to focus on a single programming language aimed instead at simplifying the non-trivial toolchain needed to run our study. The whole repositories selection process has been performed using the GitHub search tool by Dabic *et al.* [16]. At this stage, we obtained 1,401 repositories.

In our experimental design, we use the passing/failing tests as a proxy to assess the correctness of the predictions generated by *Copilot*. Thus, we need the projects to use a testing framework and to be compilable. We selected all projects that used Maven as build automation tool and for which the build of their latest release succeeded. We obtained 214 repository. By parsing the POM (Project Object Model) file[1] we only considered projects having as dependencies both jUnit [3] — a well-known unit testing framework — and Jacoco [1] — a code coverage library. We analyzed the Jacoco reports and selected as methods subject of our experiment those having at least 75% of statement coverage. This gives us confidence that the related test cases exercise an acceptable number of behaviors and, therefore, could allow to spot cases in which different generated functions for semantically-equivalent descriptions actually behave differently. We are aware that passing tests does not imply correctness. We discuss this aspect further in Section IV.

Given our goal to use the method's description as input for *Copilot*, we also exclude methods not having any associated Doc Comment for the Javadoc tool. Then, we process the Doc Comment of each method in our dataset to extract from it the first sentence (*i.e.,* from the beginning to the first "."). This is the same approach used in the literature when building datasets aimed at training DL-based techniques for Java code summarization (see *e.g.,* [21], [30]), with the training set composed by pairs <`method, code_description`>, with the latter being the first sentence of the Doc Comment. To ensure that the extracted sentence contains enough wording for the code description, we exclude all methods having less than 10 tokens in the extracted first sentence, since their description may not be sufficient for synthesizing the method.

TABLE I
Our dataset of 892 methods from 33 repositories

	Avg	Median	St. Dev.
# Tokens	154.3	92.0	218.2
# Parameters	1.6	1.0	1.2
# Cyclomatic Complexity	5.3	3.0	7.6
% Coverage	96.1	100.0	6.7

The above-described process resulted in the collection of 892 Java methods. Table I shows descriptive statistics about their characteristics in terms of number of tokens, parameters and cyclomatic complexity. These three together provide an idea about the complexity of the task *Copilot* was asked to perform (*i.e.,* the complexity of the methods it had to generate). Statistics about the coverage show, instead, the by-design high statement coverage we ensure for the included methods.

[1]POM files are used in Maven to declare dependencies towards libraries.

Full Context

```
public class Hook implements Resultsable {

    // Start: attributes from JSON file report
    private final Result result = null;
    private final Match match = null;

    @JsonDeserialize(using = OutputsDeserializer.class)
    @JsonProperty("output")
    private final Output[] outputs = new Output[0];

    // foe Ruby reports
    private final Embedding[] embeddings = new Embedding[0];
    // End: attributes from JSON file report

    @Override
    public Result getResult() {
        return result;
    }

    /** Return the embedding vector */
    public Embedding[] getEmbeddings() {
        |
    }                                          Method to be predicted

    /** Checks if the hook has content meaning as it has at least
      * attachment or result with error
      * message.
    */
    public boolean hasContent() {
        if (embeddings.length > 0) {
            return true;
        }
        if (StringUtils.isNotBlank(result.getErrorMessage())) {
            return true;
        }
        // TODO: hook with 'output' should be treated
        /  as empty or not?
        return false;
    }
}
```

Non Full Context

```
public class Hook implements Resultsable {

    // Start: attributes from JSON file report
    private final Result result = null;
    private final Match match = null;

    @JsonDeserialize(using = OutputsDeserializer.class)
    @JsonProperty("output")
    private final Output[] outputs = new Output[0];

    // foe Ruby reports
    private final Embedding[] embeddings = new Embedding[0];
    // End: attributes from JSON file report

    @Override
    public Result getResult() {
        return result;
    }

    /** Return the embedding vector */
    public Embedding[] getEmbeddings() {
        |
    }                                          Method to be predicted
}
```

Fig. 1. *GitHub Copilot's* input for both code context representations

B. Data Collection

To address $\mathbf{RQ}_0$, we experiment with two state-of-the-art paraphrasing techniques. The first is named PEGASUS [64], and it is a sequence-to-sequence DL model pre-trained using self-supervised objectives specifically tailored for abstractive text summarization and fine-tuned for the task of paraphrasing [4]. As for the second technique, we opted for Translation Pivoting (TP). Such a technique relies on natural language translation services to translates the *original* description o from English into a foreign language (*i.e.,* French), obtaining $oE \rightarrow F$. Then, $oE \rightarrow F$ is translated back in the original language ($o_{E \rightarrow F \rightarrow E}$) obtaining a paraphrase.

We provide each technique with the *original* description as input. TP failed to generate a valid paraphrase (*i.e.,* a sentence different from the original one) in 100 cases (out of 892), while this only happened once with PEGASUS. We manually analyzed whether the valid paraphrases we obtained were actually semantically equivalent to the *original* description. For such a process, each of the 1,683 paraphrases (892 for each of the two tools minus the 101 invalid ones) has been independently inspected by two authors who classified it as semantically equivalent or not. Conflicts, that arisen in 11.9% (PEGASUS) and 16.54% (TP) of cases, have been solved by a third author not involved in the first place.

Concerning $\mathbf{RQ}_1$, we start from the *original* description and we generate semantically equivalent descriptions by (i) using the two automated tools, *i.e.,* PEGASUS [4] and TP, and (ii) manually generating paraphrases.

For the manual paraphrasing, we split the 892 methods together with their *original* description into four sets and assigned each of them to one author. Each author was in charge of writing a semantically equivalent but different description of the method by looking at its code and *original* description. This resulted in a dataset (available in [5]) in which, for each subject method, we have its *original* and *paraphrased* description. In the end, for each *original* sentence, we had between one and three paraphrases: $paraphrased_{PEGASUS}$, $paraphrased_{TP}$, and $paraphrased_{manual}$. While $paraphrased_{manual}$ is available for all the methods, $paraphrased_{PEGASUS}$ and $paraphrased_{TP}$ are not. Indeed, we exclude the cases in which each of such tools failed to generate paraphrases (1 and 100, respectively) and the ones that were not considered as semantically equivalent in our manual check (based on the results of RQ_0). The maximum number of semantically equivalent paraphrases is 2,575 (up to 891 with PEGASUS, up to 792 with TP, and 892 manually).

The paraphrases, as well as the *original* description, have been used as input to *Copilot*, simulating developers asking it to synthesize the same Java method by using different natural language descriptions. At the time of our study, *Copilot* does not provide open APIs to access its services. The only way to use it is through a plugin for one of the supported IDEs. Manually invoking *Copilot* for the thousands of times needed (up to 6,934, as we will explain later) was clearly not an option. For this reason, we developed a toolchain able to automatically invoke *Copilot* on the subject instances: We exploit the AppleScript language to automate this task on a MacBook Pro, simulating the developer's interaction with Visual Studio Code (*vscode*).

For each method m_i in our dataset, we created up to four different versions of the Java file containing it (one for each of the experimented descriptions). In all such versions, we (i) emptied m_i's body, just leaving the opening and closing curly bracket delimiting it; and (ii) removed the Doc Comment, replacing it with one of the four code descriptions we prepared (*i.e., original* or one of the *paraphrased* descriptions). Starting from these files, the automation script we implemented (available in our replication package [5]) performs the following steps on each file F_i.

First, it opens F_i in *vscode* and moves the cursor within the curly brackets of the method m_i of interest. Then, it presses "`return`" to invoke *Copilot*, waiting up to 20 seconds for its recommendation. Finally, it stores the received recommendation, that could possibly be empty (*i.e.,* no recommendation received). To better understand this process, the top part of Fig. 1 depicts how the invocation of *Copilot* is performed. The gray box represents the whole Java file (*i.e.,* the context used by *Copilot* for the prediction). The emptied method (*i.e.,*`getEmbeddings`) is framed with a black border, with the cursor indicating the position in which *Copilot* is invoked. The green comment on top of the method represents one of the descriptions we created. As it can be seen, Fig. 1 includes for the same Java file two different scenarios, named *Full context* and *Non-full context*. In the *Full context* scenario (top part of Fig. 1) we provide *Copilot* with the code **preceding and following** the emptied method, simulating a developer adding a new method in an already existing Java file. In the *Non-full context* scenario, instead, we only provide as context the code preceding the emptied method (bottom part of Fig. 1), simulating a developer writing a Java file sequentially and implementing a new method.

The basic idea behind these two scenarios is that the contextual information provided to *Copilot* can play a role in its ability to predict the emptied method. Overall, the maximum number of *Copilot* invocations needed for our study is 6,934 (892 *original* descriptions plus up to 2,575 paraphrases, each of which for 2 context scenarios). After having collected *Copilot*'s recommendations, we found out that sometimes they did not only include the method we asked to generate, but also additional code (*e.g.,* other methods). To simplify the data analysis and to make sure we only consider one recommended method, we wrote a simple parsing tool to only extract from the generated recommendation the first valid method (if any).

C. Data Analysis

Concerning $\mathbf{RQ}_0$, we report the number and the percentage of 892 methods for which automatically generated paraphrases (*i.e.,* those generated by PEGASUS and by TP) have been classified as semantically equivalent to the *original* description. This will provide an idea of how reliable these tools are when used for testing the robustness of DL-based code generators. Also, this analysis allows to exclude from RQ_1 automatically generated paraphrases that are not considered semantically equivalent.

To answer $\mathbf{RQ}_1$, we preliminarily assess how far the paraphrased descriptions are from the original ones (*i.e.,* the percentage of words that has been changed). This is done by computing the normalized token-level Levenshtein distance [29] (NTLev) between the *original* (d_o) and any *paraphrased* description (d_p), as:

$$NTLev(d_o, d_p) = \frac{TLev(d_o, d_p)}{\max(\{|d_o|, |d_p|\})}$$

with $TLev$ representing the token-level Levenshtein distance between the two descriptions.

While the original Levenshtein distance works at character-level, it can be easily generalized at token-level (each unique token is represented as a specific character). In this case, a token is a word in the text. The normalized token-level Levenshtein distance provides an indication of the percentage of words that must be changed in the *original* description to obtain a *paraphrased* one.

Then, we analyze the percentage of methods for which the *paraphrased* descriptions result in a different method prediction as compared to the *original* one. When they are different, we also assess how far the methods obtained by using a given *paraphrased* description is from the method recommended when providing the *original* description as input. Also in this case we use the token-level Levenshtein distance as metric. The latter is computed with the same formula previously reported for the natural text descriptions; in this case, however, the tokens are not the words but the Java syntactic tokens. Thus, NTLev indicates in this case the percentage of code tokens that must be changed to convert the method obtained through the *original* description into the one recommended with one of the paraphrases.

Finally, we study the "quality" of the recommendations obtained using the different descriptions both in the *Full context* and *Non-full context* scenarios. Given the sets of methods generated from the *original* description and each of the paraphrasing approach considered, we present the percentages of methods for which *Copilot*: (i) synthesized a method passing all the related test cases (*PASS*); (ii) synthesized a method that does not pass at least one of the test cases (*FAIL*); (iii) generated an invalid method (*i.e.,* with syntactic errors) (*ERROR*); (iv) did not generate any method (*EMPTY*). Syntactic errors have been identified as recommendations for which *Java Parser* [2] did not manage to identify a valid recommended method (*i.e.,* cases in which *Java Parser* fails to identify a method node in the AST generated for the obtained recommendation). On top of the passing/failing methods, we also compute the token-level Levenshtein distance and the CodeBLEU [47] between each synthesized method and the target one (*i.e.,* the one originally implemented by the developers). CodeBLEU measures how similar two methods are. Differently from the BLEU score [44], CodeBLEU evaluates the predicted code considering not only the overlapping n-grams but also syntactic and semantic match of the two pieces of code (predicted and reference) [47].

D. Replication Package

The code and data used in our study are publicly available [5]. In particular, we provide (i) the dataset of manually defined and automatically generated paraphrases; (ii) the AppleScript code used to automate the *Copilot* triggering; (iii) the code used to compute the CodeBLEU and the Levenshtein distance; (iv) the dataset of 892 methods and related tests used in our study; (v) the scripts used to automatically generate the paraphrased descriptions using PEGASUS and TP; and (vi) all raw data output of our experiments.

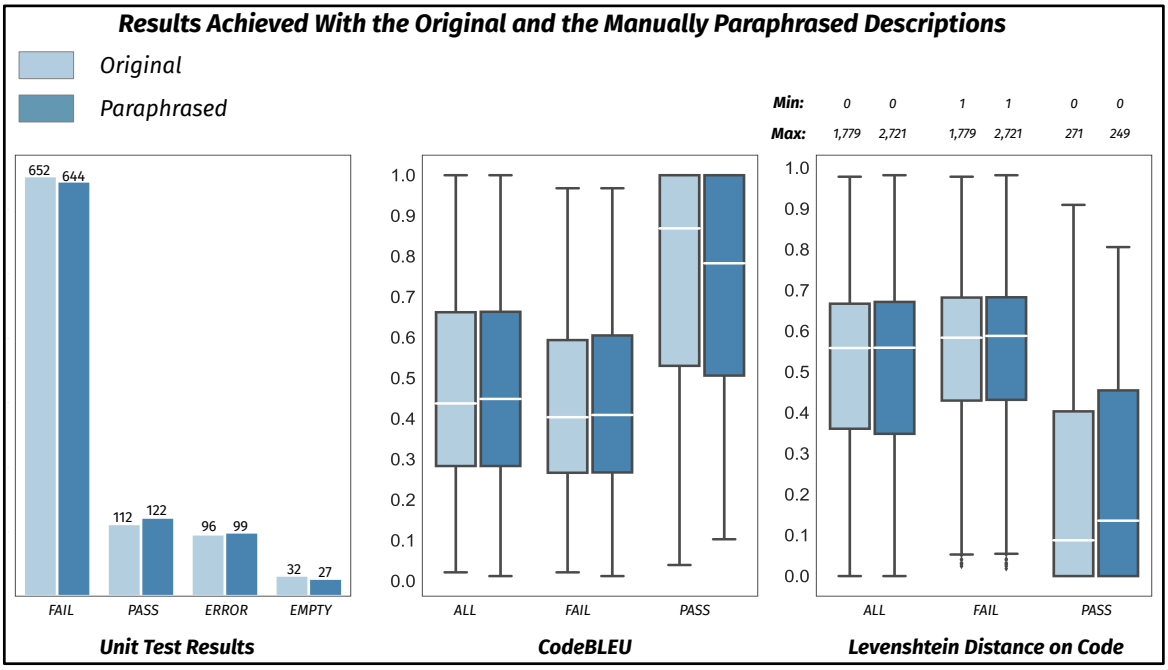

Fig. 2. Results achieved by Copilot when considering the *Full context* code representation on $paraphrases_{manual}$.

III. Results Discussion

As previously explained, in RQ_1 we conducted our experiments both in the *Full context* and in the *Non-full context* scenario. Since the obtained findings are similar, due to space limitations we only discuss in the paper the results achieved in the *Full context* scenario (*i.e.,* the case in which we provide *Copilot* with all code preceding and following the method object of the prediction). The results achieved in the *Non-full context* scenario are available in our replication package [5].

A. RQ_0: Evaluation of Automated Praphrase Generators

TABLE II
Number of semantically equivalent or nonequivalent paraphrased descriptions obtained using PEGASUS and TP.

	Equivalent	Nonequivalent	Invalid
PEGASUS	666 (74.7%)	225 (25.2%)	1 (0.1%)
TP	688 (77.1%)	104 (11.7%)	100 (11.2%)

Table II reports the number of semantically equivalent and nonequivalent descriptions obtained using the two state-of-the-art paraphrasing techniques, namely PEGASUS and Translation Pivoting (TP), together with the number of invalid paraphrases generated. Out of the 892 *original* descriptions on which they have been run, PEGASUS generated 666 (75%) semantically equivalent descriptions, while TP went up to 688 (77%). If we do not consider the invalid paraphrases, *i.e.,* the cases for which the techniques do not actually provide any paraphrase, the latter obtains ∼87% of correctly generated paraphrases.

These findings suggest that the two paraphrasing techniques can be adopted as testing tools to assess the robustness of DL-based code recommenders. In particular, once established a reference description (*e.g.,* the *original* description in our study), these tools can be applied to paraphrase it and verify whether, using the reference and the paraphrased descriptions, the code recommenders generate different predictions.

Answer to RQ_0. State-of-the-art paraphrasing techniques can be used as starting point to test the robustness of DL-based code recommenders, since they are able to generate semantically equivalent descriptions of a reference text in up to 77% of cases.

B. RQ_1: Robustness of GitHub Copilot

Performance of Copilot when using the original and the paraphrased description as input. Fig. 2 summarizes the performance achieved by *Copilot* when using the *original* description (light blue) and the manually generated *paraphrased* description (dark blue) as input. Similarly, we report in Fig. 3 the performance obtained when considering the paraphrases generated with the two automated techniques, *i.e.,* PEGASUS and TP (top and bottom of Fig. 3, respectively). It is worth noticing that, in the latter, we only considered in the analysis the paraphrases manually considered as equivalent in RQ_0, *i.e.,* 666 for PEGASUS and 688 for TP.

A first interesting result is that, as it can be noticed from Fig. 2 and Fig. 3, the results obtained with the three methodologies are very similar. For this reason, to avoid repetitions, in the following, we will mainly focus on the results obtained with the manually generated paraphrases.

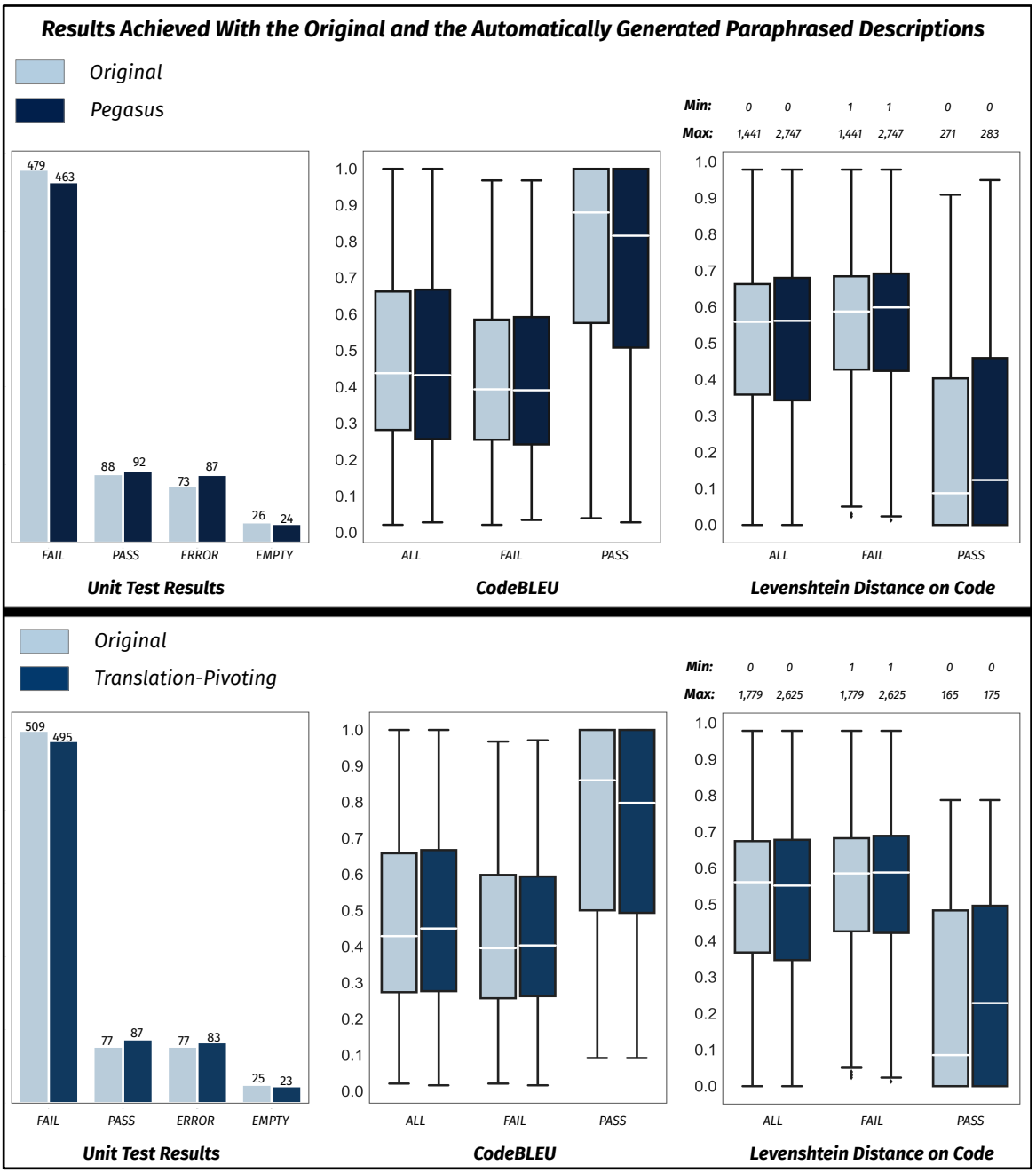

Fig. 3. Results achieved by Copilot when considering the *Full context* code representation on $paraphrases_{PEGASUS}$ and $paraphrases_{TP}$.

Also, as we will discuss, the quality of *Copilot*'s recommendations is very similar when using the *original* and the *paraphrased* descriptions.

In Fig. 2, the bar chart in the left side reports the number of methods recommended by *Copilot* (out of 892) that resulted in failing tests, passing tests, syntactic errors, and no (*i.e.,* empty) recommendation. Looking at such a chart, the first thing that leaps to the eyes is the high percentage of Java methods (∼73% for the *original* and ∼72% for the *paraphrased* description) for which *Copilot* was not able to synthesize a method passing the related unit tests.

Only ∼13% of instances (112 and 122 depending on the used description) resulted in test-passing methods. While such a result seems to indicate limited performance of *Copilot*, it must be considered the difficulty of the code generation tasks involved in our study. Indeed, we did not ask *Copilot* to generate simple methods possibly implementing quite popular routines (*e.g.,* a method to generate an MD5 hash from a string) but rather randomly selected methods that, as shown in Table I, are composed, on average, by more than 150 tokens (median = 92) and have an average cyclomatic complexity of 5.3 (median = 3.0).

Target Method

```
public void removeListener(IChemObjectListener col){
    if (chemObjectListeners == null) {
        return;
    }
    List<IChemObjectListener> listeners = lazyChemObjectListeners();
    if (listeners.contains(col)) {
        listeners.remove(col);
    }
}
```

Recommended method starting from the original description

PASS

```
public void removeListener(IChemObjectListener col){
     if (chemObjectListeners == null) {
        return;
     }
     lazyChemObjectListeners().remove(col);
}
```

CodeBLEU: 0.45

Fig. 4. Example of recommended method that passes the unit tests but reports a low CodeBLEU score compared to the oracle (*i.e.,* target method).

Thus, we consider the successful generation of more than 110 of these methods a quite impressive result for a code recommender. The remaining ∼15% of instances resulted either in a parsing error (∼100 methods) or in an empty recommendation (∼30 methods).

The box plot in the middle part of Fig. 2 depicts the results achieved in terms of CodeBLEU [47] computed between the recommended methods and the target one (*i.e.,* the one implemented by the original developers). Higher values indicate higher similarity between the compared methods. Instead, in the right box plot, we show the normalized Levenshtein distance, for which lower values indicate higher similarity.

For both metrics, we depict the distributions when considering all generated predictions, the ones failing tests, and the ones passing tests. As expected, higher (lower) values of CodeBLEU (Levenshtein distance) are associated with test-passing methods. Indeed, for the latter, the median CodeBLEU is ∼0.80 (Levenshtein = ∼0.10) as compared to the ∼0.40 (Levenshtein = ∼0.58) of test-failing methods. Despite such an expected finding, it is interesting to notice that 25% of test-passing methods have a rather low CodeBLEU <0.50.

Fig. 4 shows an example of recommended method having a CodeBLEU with the target method of 0.45 and passing the related tests. The recommended method, while substantially different from the target, captures the basic logic implemented in it. The target method first checks if the object `chemObjectListeners` is `null` and, if not, it proceeds removing from the `listeners` list the element matching the one provided as parameter (*i.e.,*`col`). The method synthesized by *Copilot* avoids the second `if` statement by directly performing the remove operation after the `null` check.

Note that there the two implementations are equivalent: The `remove` method of `java.util.List` preliminarily checks whether the passed element is contained in the list before removing it. While the check in the original method has no functional role, together with the introduction of the `listeners` variable, it might have been introduced to make the method more readable and self-explanatory.

Target Method

```
public static void translateAllPositive(IAtomContainer atomCon){
    double minX = Double.MAX_VALUE;
    double minY = Double.MAX_VALUE;
    Iterator<IAtom> atoms = atomCon.atoms().iterator();
    while (atoms.hasNext()) {
        IAtom atom = (IAtom) atoms.next();
        if (atom.getPoint2d() != null) {
            if (atom.getPoint2d().x < minX) {
                minX = atom.getPoint2d().x;
            }
            if (atom.getPoint2d().y < minY) {
                minY = atom.getPoint2d().y;
            }
        }
    }
    logger.debug("Translating: minx=" + minX + ", minY=" + minY);
    translate2D(atomCon, minX * -1, minY * -1);
}
```

Recommended method starting from the original description

PASS

```
public static void translateAllPositive
     (IAtomContainer atomCon){
        double minX = Double.POSITIVE_INFINITY;
        double minY = Double.POSITIVE_INFINITY;
        double minZ = Double.POSITIVE_INFINITY;
        for (IAtom atom : atomCon.atoms()) {
            if (atom.getPoint2d() != null) {
                minX = Math.min(minX, atom.getPoint2d().x);
                minY = Math.min(minY, atom.getPoint2d().y);
            }
            if (atom.getPoint3d() != null) {
                minX = Math.min(minX, atom.getPoint3d().x);
                minY = Math.min(minY, atom.getPoint3d().y);
                minZ = Math.min(minZ, atom.getPoint3d().z);
            }
        }
        for (IAtom atom : atomCon.atoms()) {
            if (atom.getPoint2d() != null) {
                 atom.setPoint2d(new Point2d(
                                atom.getPoint2d().x - minX,
                                atom.getPoint2d().y - minY));
            }
            if (atom.getPoint3d() != null) {
                atom.setPoint3d(new Point3d(
                                atom.getPoint3d().x - minX,
                                atom.getPoint3d().y - minY,
                                atom.getPoint3d().z - minZ));
            }
        }
}
```

Levenshtein distance: 165

Fig. 5. Example of recommended methods that pass the unit tests but would require 165 edit actions to match the target method.

Similarly, Fig. 5 shows an example of prediction passing the tests but that, accordingly to the Levenshtein distance, would require 165 token-level edits to match the target prediction (NTLev=63%). Differently from the previous example, it is clear that, in this case, the two methods do not have the same behavior since the recommended one also treats 3D points, while the original one only 2D points. In other words, the tests fail to capture the difference in the behavior.

These examples provide two interesting observations. The first is that, metrics such as CodeBLEU and Levenshtein distance may result in substantially wrong assessments of the quality of a prediction. Indeed, while the discussed predictions have low CodeBLEU/high Levenshtein values and, thus, would be considered as unsuccessful predictions in most of the empirical evaluations, it is clear that they are valuable recommendations for a developer, even when not 100% correct (see Fig. 5). This poses questions on the usage of these metrics in the evaluation of code recommenders. Second, also the testing-based evaluation shows, as expected, some limitations as in the second example, in which the two methods do not implement the same behavior but both pass the tests.

As a final note, it is also interesting to observe as 25% of test-failing predictions exhibit high values (>∼0.60) of CodeBLEU, indicating a high code similarity that, however, does not reflect in test-passing recommendations.

Impact of paraphrasing the input descriptions. Out of the 892 manually paraphrased descriptions, 408 (46%) result in different code recommendations as compared to the *original* description. This means that *Copilot* synthesizes different methods when it is provided as input with the *original* description and with the manually *paraphrased* description, which are supposed to summarize the same piece of code. Note that at this stage we are not focusing on the "quality" of the obtained predictions in any way. We are just observing that different input descriptions have indeed an impact on the recommended code. This implies that developers using different wordings to describe a needed method may end up with different recommendations. Such differences also result in the potential loss of correct recommendations. Indeed, out of the 112 test-passing predictions obtained with the *original* description and the 122 obtained with the manually *paraphrased* description, only 98 are in overlap, indicating that there are 38 correct recommendations only generated either by the *original* (14) or the *paraphrased* (24) description.

To have a deeper look into the 408 different predictions generated by *Copilot* with the *original* and the *paraphrased* description, the left part of Fig. 6 (light blue) shows the normalized token-level Levenshtein distance between (i) the *original* description and the *paraphrased* description (see the boxplot labeled with "Description"), and (ii) the method obtained using the *original* description and that recommended using the *paraphrased* description ("Code"). The "Description" boxplot depicts the percentage of words that must be changed to convert the *paraphrased* description into the *original* one. As it can be seen, while describing the same method, the *paraphrased* descriptions can be substantially different as compared to the *original* ones, with 50% of them requiring changes to more than 70% of their words. Similarly, the different methods recommended in the 408 cases under analysis, can be substantially different, with a median of ∼30% of code tokes that must be changed to convert the recommendation obtained with the *original* description into the one obtained using the *paraphrased* description (see the "Code" boxplot).

These findings are confirmed for the automatically paraphrased descriptions (see the middle and the right part of Fig. 6 for the results achieved with the PEGASUS and TP paraphrases, respectively). As it can be seen, the main difference as compared to the results of the manually paraphrased description (left part of Fig. 6) is that TP changes a substantially lower number of words in the *original* description as compared to PEGASUS and to the manual paraphrasing. Such a finding is expected considering that TP just translates the *original* description back and forth from English to French, thus rarely adding new words to the sentence, something that is likely to happen using PEGASUS or by paraphrasing the sentence manually.

Answer to RQ_1. Different (but semantically equivalent) natural language descriptions of the same method are likely to result in different code recommendations generated by DL-based code generation models. Such differences can result in a loss of correct recommendations (∼28% of test-passing methods can only be obtained either with the *original* or the *paraphrased* descriptions). These findings suggest that testing the robustness of DL-based code recommenders may play an important role in ensuring their usability and in defining possible guidelines for the developers using them.

IV. Threats to Validity

Threats to *construct validity* concern the relationship between the theory and what we observe. Concerning the performed measurements, we exploit the passing tests as a proxy for the correctness of the recommendations generated by *Copilot*. We acknowledge that passing tests does not imply code correctness. However, this it can provide hints about the code behavior. To partially address this threat we focused our study on methods having high statement coverage (median = 100%). Also, we complemented this analysis with the CodeBLEU and the normalized token-level Levenshtein distance. As for the execution of our study, we automatically invoked *Copilot* rather than using it as actual developers would do: We automatically accepted the whole recommendations and did not simulate a scenario in which a developer selects only parts of the provided recommendations. In other words, while our automated script simulates a developer invoking *Copilot* for help, it cannot simulate the different usages a developer can make of the received code recommendation.

Threats to *internal validity* concern factors, internal to our study, that could affect our results. While in RQ_2 we had multiple authors inspecting the semantic equivalence of the paraphrasing generated by the automated tools, in RQ_1 we relied on a single author to paraphrase the *original* description. This introduces some form of subjectivity bias. However, the whole point of our paper is that, indeed, subjectivity plays a role in the natural language description of a function to generate and we are confident that the written descriptions were indeed semantically equivalent to the *original* one. Indeed, the authors involved in the manual paraphrasing have an average of seven years of experience in Java. Also related to internal validity is our choice of using the first sentence of the Doc Comments as the *original* natural language description. These sentences may be of low quality and not representative of how a developer would describe a method they want to automatically generate. This could substantially influence our findings, especially in terms of the effectiveness of *Copilot* (*i.e.,* its ability to generate test-passing methods). However, such a threat is at least mitigated by the fact that *Copilot* has also been invoked using the manually written descriptions, showing a similar effectiveness. A final threat regards the projects used for our study.

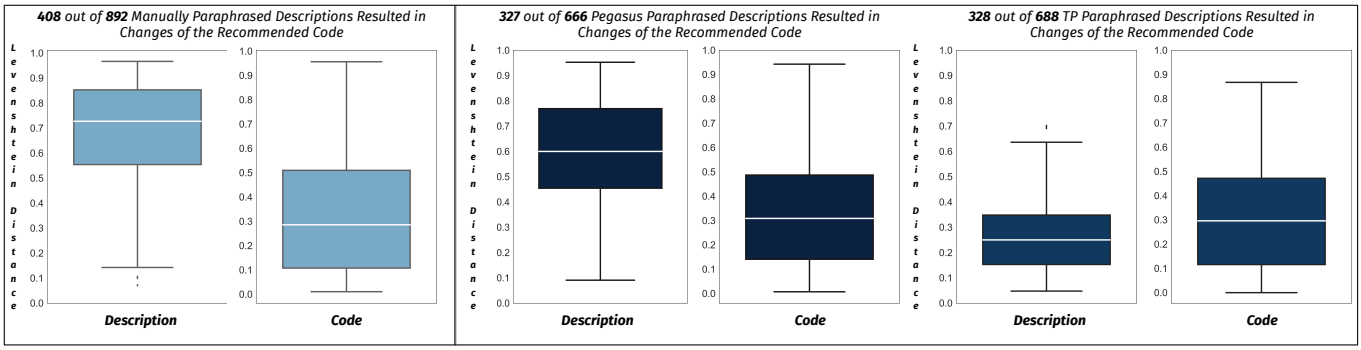

Fig. 6. Levenshtein distance between the *original* description and (i) the manually *paraphrased* descriptions (left part) and (ii) the descriptions automatically paraphrased by PEGASUS (middle part) and Translate Pivoting (right). Similarly, we report the Levenshtein distance between the method recommended using the *original* description and the three paraphrases. The latter is only computed for recommendations in which the obtained output differs.

Those are open-source projects from GitHub, and it is likely that at least some of them have been used for training Copilot itself. In other words, the absolute actual effectiveness reported might not be reliable. However, the objective of our study is to understand the differences when different paraphrases are used rather than the absolute performance of Copilot, like previous studies did (*e.g.,* [41]).

Threats to *external validity* are related to the possibility to generalize our results. Our study has been run on 892 methods we carefully selected as explained in Section II-A. Rather than going large-scale, we preferred to focus on methods having a high test coverage and a verbose first sentence in the Doc Comment. Larger investigations are needed to corroborate or contradict our findings. Similarly, we only focused on Java methods, given the effort required to implement the toolchain needed for our study, and in particular the script to automatically invoke *Copilot* and parse its output. Running the same experiment with other languages is part of our future agenda.

V. Related Work

Recommender systems for software developers are tools supporting practitioners in daily activities [36], [49], such as documentation writing and retrieval [62], [37], [38], [22], refactoring [10], [53], bug triaging [52], [61], bug fixing [28], [56], [32], etc. Among those, code recommenders, such as code completion tools, have became a crucial feature of modern Integrated Development Environments (IDEs) and support in speeding up code development by suggesting the developers code they are likely to write [11], [27], [15]. Given the empirical nature of our work, that focuses on investigating a specific aspect of code recommenders, in this section we do not discuss all pervious works proposing novel or improving existing code recommenders (see *e.g.,* [62], [37], [38], [28], [56], [32], [59], [42], [34], [26], [6], [27], [25], [58], [55]). Instead, we focus on empirical studies looking at code recommenders from different perspectives (Section V-A) and on studies specifically focused on GitHub
copilot (Section V-B).

A. Empirical Studies on Code Recommenders

Proksch *et al.* [46] conducted an empirical study aimed at evaluating the performance of code recommenders when suggesting method calls. Their study has been run on a real-world dataset composed of developers' interactions captured in the IDE. Results showed that commonly used evaluation techniques based on synthetic datasets extracted by mining released code underperform due to a context miss.

On a related research thread, Hellendoorn *et al.* [19] compared code completion models on both real-world and synthetic datasets. Confirming what observed by Proksch *et al.*, they found that the evaluated tools are less accurate on the real-world dataset, thus concluding that synthetic benchmarks are not representative enough. Moreover, they found that the accuracy of code completion tools substantially drops in challenging completion scenarios, in which developers would need them the most.

Mărășoiu *et al.* [35] analyzed how practitioners rely on code completion during software development. The results showed that the users actually ignore many synthesized suggestions. Such a finding has been corroborated by Arrebola and Junior [8], who stressed the need for augmenting code recommender systems with the development's context.

Jin and Servant [24] and Li *et al.* [31] investigated the *hidden costs* of code recommendations. Jin and Servant found that IntelliSense, a code completion tool, sometimes underperforms by providing the suitable recommendation far from the top of the recommended list of solutions. Consequently, developers are discouraged from picking the right suggestion. Li *et al.*, aware of this potential issue, conducted a coding experiment in which they try to predict whether correct results are generated by code completion models, showing that their approach can reduce the percentage of false positives up to 70%.

Previous studies also assessed the actual usefulness of these tools. Xu *et al.* [63] ran a controlled experiment with 31 developers who were asked to complete implementation tasks with and without the support of two code recommenders. They found a marginal gain in developers' productivity when using the code recommenders.

Ciniselli *et al.* [14] empirically evaluated the performance of two state-of-the-art Transformer-based models in challenging coding scenarios, for example, when the code recommender is required to generate an entire code block (*e.g.,* the body of a `for` loop). The two experimented models, RoBERTa and Text-To-Text Transfer Transformer (T5), achieved good performance (∼69% of accuracy) in the more classic code completion scenario (*i.e.,* predicting few tokens needed to finalize a statement), while reported a substantial drop of accuracy (∼29%) when dealing with the previously described more complex block-level completions.

Our study is complementary to the ones discussed above. Indeed, we investigate the robustness of DL-based code recommenders supporting what it is know in the literature as "*natural language to source code translation*". We show that semantically equivalent code descriptions can result in different recommendations, thus posing questions on the usability of these tools.

B. Empirical Studies on GitHub Copilot

GitHub *Copilot* has been recently introduced as the state-of-the-art code recommender, and advertised as an "AI pair programmer" []. Since its release, researchers started investigating its capabilities.

Most of the previous research aimed at evaluating the impact of GitHub *Copilot* on developers' productivity and its effectiveness (in terms of correctness of the provided solutions). Imai [23] investigated to what extent *Copilot* is actually a valid alternative to a human pair programmer. They observed that *Copilot* results in increased productivity (*i.e.,* number of added lines of code), but decreased quality in the produced code. Ziegler *et al.* [65] conducted a case study in which they investigated whether usage measurements about *Copilot* can predict developers' productivity. They found that the acceptance rate of the suggested solutions is the best predictor for perceived productivity. Vaithilingam *et al.* [57] ran an experiment with 24 developers to understand how *Copilot* can help developers complete programming tasks. Their results show that *Copilot* does not improve the task completion time and success rate. However, developers report that they prefer to use *Copilot* because it recommends code that can be used as a starting point and saves the effort of searching online.

Nguyen and Nadi [41] used LeetCode questions as input to *Copilot* to evaluate the solutions provided for several programming languages in terms of correctness — by running the test cases available in LeetCode — and understandability — by computing their Cyclomatic Complexity and Cognitive Complexity [12]. They found notable differences among the programming languages in terms of correctness (between 57%, for Java, and 27%, for JavaScript). On the other hand, *Copilot* generates solutions with low complexity for all the programming languages. While we also measure the effectiveness of the solutions suggested by *Copilot*, our main focus is on understanding its robustness when different inputs are provided.

Two previous studies aimed at evaluating the security of the solutions recommended by *Copilot*. Hammond *et al.* [45] investigated the likelihood of receiving from *Copilot* recommendations including code affected by security vulnerabilities. They observed that vulnerable code is recommended in 40% of cases out of the completion scenarios they experimented with. On a similar note, Sobania *et al.* [50] evaluated GitHub *Copilot* on standard program synthesis benchmark problems and compared the achieved results with those from the genetic programming literature. The authors found that the performance of the two approaches are comparable. However, approaches based on genetic programming are not mature enough to be deployed in practice, especially due to the time they require to synthesize solutions. In our study, we do not focus on security, but only on the correctness of the suggested solutions.

Albert Ziegler, in a blog post about GitHub *Copilot*[2] investigated the extent to which the tool suggestions are copied from the training set they used. Ziegler reports that *Copilot* rarely recommends verbatim copies of code taken from the training set.

VI. Conclusions and Future Work

We investigated the extent to which DL-based code recommenders tend to synthesize different code components when starting from different but semantically equivalent natural language descriptions. We selected *GitHub Copilot* as the tool representative of the state-of-the-art and asked it to generate 892 non-trivial Java methods starting from their natural language description. For each method in our dataset we asked *Copilot* to synthesize it using: (i) the *original* description, extracted as the first sentence in the Javadoc; and (ii) *paraphrased* descriptions. We did this both by manually modifying the *original* description and by using automated paraphrasing tools, after having assessed their reliability in this context.

We found that in ∼46% of cases semantically equivalent but different method descriptions result in different code recommendations. We observed that some correct recommendations can only be obtained using one of the semantically equivalent descriptions as input.

Our results highlight the importance of providing a proper code description when asking DL-based recommenders to synthesize code. In the new era of AI-supported programming, developers must learn how to properly describe the code components they are looking for to maximize the effectiveness of the AI support.

Our future work will focus on answering our first research question *in vivo* rather than *in silico*. In other words, we aim at running a controlled experiment with developers to assess the impact of the different code descriptions they write on the received recommendations. Also, we will investigate how to customize the automatic paraphrasing techniques to further improve their performance on software-related text (such as methods' descriptions).

[2] https://docs.github.com/en/github/copilot/research-recitation

REFERENCES

[1] *Jacoco*, https://www.eclemma.org/jacoco/.

[2] *Java Parser*, https://github.com/javaparser/javaparser.

[3] *jUnit*, https://junit.org/junit5/.

[4] *PEGASUS fine-tuned for paraphrasing*, https://huggingface.co/tuner007/pegasus_paraphrase.

[5] *Replication package*, https://github.com/copilot-robustness/robustness.

[6] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 281–293.

[7] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," *arXiv*, pp. arXiv–1910, 2019.

[8] F. V. Arrebola and P. T. A. Junior, "On source code completion assistants and the need of a context-aware approach," in *International Conference on Human Interface and the Management of Information*. Springer, 2017, pp. 191–201.

[9] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Context-sensitive code completion tool for better api usability," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 621–624.

[10] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, 2014.

[11] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE 2009, 2009, pp. 213–222.

[12] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *Proceedings of the 2018 international conference on technical debt*, 2018, pp. 57–58.

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[14] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. D. Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 5555.

[15] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *Proceedings of the 18th Working Conference on Mining Software Repositories*, ser. MSR '21, 2021, p. To Appear.

[16] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *Proceedings of the 18th International Conference on Mining Software Repositories*, ser. MSR'21, 2021, p. To appear. [Online]. Available: https://arxiv.org/abs/2103.04682

[17] N. A. Ernst and G. Bavota, "Ai-driven development is here: Should you worry?" *IEEE Softw.*, vol. 39, no. 2, pp. 106–110, 2022.

[18] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, p. 763?773.

[19] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 960–970.

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

[21] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, F. Khomh, C. K. Roy, and J. Siegmund, Eds. ACM, 2018, pp. 200–210.

[22] ——, "Deep code comment generation," ser. ICPC '18, 2018.

[23] S. Imai, "Is github copilot a substitute for human pair-programming? an empirical study," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2022, pp. 319–321.

[24] X. Jin and F. Servant, "The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 70–73.

[25] R. Karampatsis and C. A. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: http://arxiv.org/abs/1903.05734

[26] J. Kim, S. Lee, S. Hwang, and S. Kim, "Adding examples into java documents," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 540–544.

[27] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *arXiv preprint arXiv:2003.13848*, 2020.

[28] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.

[29] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[30] B. Li, M. Yan, X. Xia, X. Hu, G. Li, and D. Lo, "Deepcommenter: a deep code comment generation tool with hybrid lexical and syntactical information," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1571–1575.

[31] J. Li, R. Huang, W. Li, K. Yao, and W. Tan, "Toward less hidden cost of code completion with acceptance and ranking models," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 195–205.

[32] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, p. 602?614.

[33] B. Lin, F. Zampetti, G. Bavota, M. D. Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: how far can we go?" in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 94–104.

[34] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2020. Association for Computing Machinery, 2020.

[35] M. Mărășoiu, L. Church, and A. Blackwell, "An empirical investigation of code completion usage by professional software developers," in *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group*, 2015.

[36] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, 2013.

[37] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, p. 880?890.

[38] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora, "Arena: An approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, 2017.

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

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

[41] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022, pp. 1–5.

[42] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2api: Synthesizing api code usage templates from english texts with statistical translation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, p. 1013?1017.

[43] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[44] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[45] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "An empirical cybersecurity evaluation of github copilot's code contributions," *arXiv preprint arXiv:2108.09293*, 2021.

[46] S. Proksch, S. Amann, S. Nadi, and M. Mezini, "Evaluating the evaluations of code recommender systems: a reality check," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 111–121.

[47] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[48] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[49] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[50] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming," *arXiv preprint arXiv:2111.07875*, 2021.

[51] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," *arXiv preprint arXiv:2005.08025*, 2020.

[52] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, p. 365?375.

[53] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of jdeodorant: Lessons learned from the hunt for smells," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 4–14.

[54] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–280. [Online]. Available: https://doi.org/10.1145/2635868.2635875

[55] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," *CoRR*, vol. abs/2009.05634, 2020.

[56] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[57] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[58] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, p. To Appear.

[59] F. Wen, E. Aghajani, C. Nagy, M. Lanza, and G. Bavota, "Siri, write the next method," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 138–149.

[60] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[61] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2017.

[62] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," ser. MSR '06, 2006.

[63] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," 2021.

[64] J. Zhang, Y. Zhao, M. Saleh, and P. J. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," 2019.

[65] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.