The Journal of Systems and Software 214 (2024) 112059

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT[☆]

Phuong T. Nguyen [a], Juri Di Rocco [a], Claudio Di Sipio [a], Riccardo Rubei [a], Davide Di Ruscio [a,*], Massimiliano Di Penta [b]

[a] *Università degli studi dell'Aquila, 67100 L'Aquila, Italy*
[b] *Università degli studi del Sannio, Italy*

ARTICLE INFO

*Keywords:*
ChatGPT
Code classification
CodeBERT
Pre-trained Models

ABSTRACT

Since its launch in November 2022, ChatGPT has gained popularity among users, especially programmers who use it to solve development issues. However, while offering a practical solution to programming problems, ChatGPT should be used primarily as a supporting tool (e.g., in software education) rather than as a replacement for humans. Thus, detecting automatically generated source code by ChatGPT is necessary, and tools for identifying AI-generated content need to be adapted to work effectively with code. This paper presents GPTSniffer– a novel approach to the detection of source code written by AI – built on top of CodeBERT. We conducted an empirical study to investigate the feasibility of automated identification of AI-generated code, and the factors that influence this ability. The results show that GPTSniffer can accurately classify whether code is human-written or AI-generated, outperforming two baselines, GPTZero and OpenAI Text Classifier. Also, the study shows how similar training data or a classification context with paired snippets helps boost the prediction. We conclude that GPTSniffer can be leveraged in different contexts, e.g., in software engineering education, where teachers use the tool to detect cheating and plagiarism, or in development, where AI-generated code may require peculiar quality assurance activities.

## 1. Introduction

ChatGPT[1] (OpenAI, 2023) is a generative Artificial Intelligence (AI) tool, able to produce convincingly human answers to queries from users. Since its public release on November 30, 2022, ChatGPT has attracted the attention of both expert- and non-expert users worldwide, reaching one million users only five days after the launching. ChatGPT rises to fame thanks to its ability to provide human-like answers, as well as to maintain a thread of conversation in a natural way.

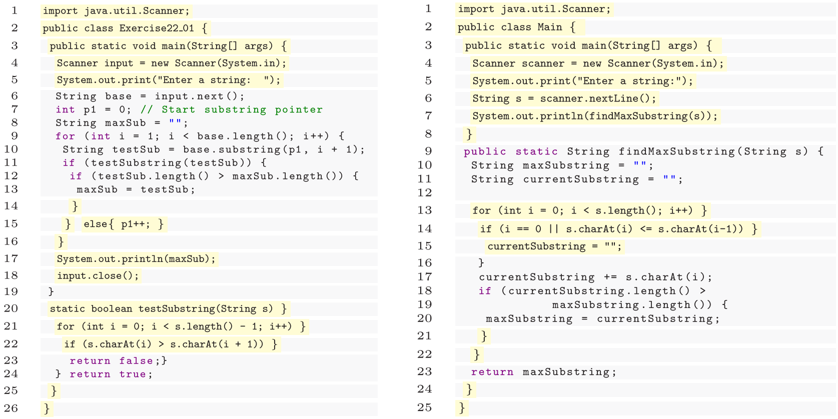
One of the areas in which ChatGPT appears to be particularly promising is its ability to support developers in a variety of tasks (Bucaioni et al., 2024), that range from writing source code that fulfills a given (natural language) specification, to creating a software architecture/design, generating tests, or fixing a bug.

Leveraging ChatGPT–as well as some previously-existing AI-based code generation tools such as GitHub Copilot (GitHub, 2024), OpenAI Codex (OpenAI, 2024), or Tabnine (Tabnine, 2023)–to get recommendations for source code solutions is becoming very popular among developers. This does not happen without risks, as it has been shown that generative models could provide vulnerable code (EuroPol, 2023; Pearce et al., 2021), and also there is a wide yet controversial discussion on possible copyright and licensing infringements (Reda, 2023; StephanieGlen, 2023).

Moreover, when students use ChatGPT or other code generators during their learning processes, issues on risks and benefits arise, and this has triggered quite some discussion among educators. On the positive side, code snippets generated by ChatGPT provide students with a practical way to complete their assignments. At the same time, one significant risk is that students would not develop some essential skills that can be acquired only through self-learning, e.g., critical thinking and problem-solving. Moreover, handing in code written by ChatGPT without additional work can be considered a form of fraud. Such behaviors trigger ethical concerns, as students have their work

[☆] Editor: Xiao Liu.
[*] Corresponding author.
*E-mail addresses:* phuong.nguyen@univaq.it (P.T. Nguyen), juri.dirocco@univaq.it (J. Di Rocco), claudio.disipio@univaq.it (C. Di Sipio), riccardo.rubei@univaq.it (R. Rubei), davide.diruscio@univaq.it (D. Di Ruscio), dipenta@unisannio.it (M. Di Penta).
[1] GPT stands for Generative Pre-trained Transformer.
[2] http://bit.ly/3QlhSWV, https://bit.ly/3qfHqu4

https://doi.org/10.1016/j.jss.2024.112059
Received 2 August 2023; Received in revised form 5 April 2024; Accepted 8 April 2024
Available online 16 April 2024



```
1  import java.util.Scanner;
2  public class Exercise22_01 {
3   public static void main(String[] args) {
4    Scanner input = new Scanner(System.in);
5    System.out.print("Enter a string:  ");
6    String base = input.next();
7    int p1 = 0; // Start substring pointer
8    String maxSub = "";
9    for (int i = 1; i < base.length(); i++) {
10    String testSub = base.substring(p1, i + 1);
11    if (testSubstring(testSub)) {
12     if (testSub.length() > maxSub.length()) {
13      maxSub = testSub;
14     }
15    }  else{ p1++; }
16   }
17   System.out.println(maxSub);
18   input.close();
19  }
20  static boolean testSubstring(String s) }
21   for (int i = 0; i < s.length() - 1; i++) }
22    if (s.charAt(i) > s.charAt(i + 1)) }
23     return false;}
24   } return true;
25  }
26 }
```

(a) Human-written code

```
1  import java.util.Scanner;
2  public class Main {
3   public static void main(String[] args) {
4    Scanner scanner = new Scanner(System.in);
5    System.out.print("Enter a string:");
6    String s = scanner.nextLine();
7    System.out.println(findMaxSubstring(s));
8   }
9   public static String findMaxSubstring(String s) {
10   String maxSubstring = "";
11   String currentSubstring = "";
12
13   for (int i = 0; i < s.length(); i++) }
14    if (i == 0 || s.charAt(i) <= s.charAt(i-1)) }
15     currentSubstring = "";
16    }
17    currentSubstring += s.charAt(i);
18    if (currentSubstring.length() >
19              maxSubstring.length()) {
20     maxSubstring = currentSubstring;
21    }
22   }
23   return maxSubstring;
24  }
25 }
```

(b) ChatGPT-generated code

**Fig. 1.** Two code snippets to display the maximum consecutive increasingly ordered substring.

done without actually performing their own research. Lately, as a precautionary measure, universities in different countries even imposed a ban on ChatGPT,[2] prohibiting their students from using the system to generate homework solutions or to compose essays.

As software engineering researchers, on the one hand, we need to promote the democratized use of AI tools to facilitate daily programming tasks. On the other hand, we believe that it is necessary to recognize whether a source code element has been written by AI for various reasons, and, in particular *(i)* from the professional development side, dealing with reliability, security, and legal problems; and *(ii)* from the educational side, coping with cheating and plagiarism.

Recently, tools such as GPTZero (GPTZero, 2023) and OpenAI Text Classifier (Classifier, 2023) have been developed to automatically recognize if a text is written by OpenAI technologies. Unfortunately, we noticed, by some attempts, that such tools are not necessarily good at distinguishing between source code written by humans and AI. We conjecture that the underpinning engine has been trained on natural language text, rather than source code. This makes it necessary to train specific classifiers aimed at identifying AI-generated code.

This paper presents an empirical study to investigate the extent to which it is possible to automatically detect whether a code snippet is written by ChatGPT or humans, as well as the factors that can influence this ability. To achieve this, we present GPTSniffer—a machine learning solution to determine whether a piece of source code has been generated by ChatGPT. The classification engine is based on CodeBERT (Feng et al., 2020), a pre-trained model built on top of a code search dataset, i.e., CodeSearchNet (Husain et al., 2019). To the best of our knowledge, there is no specific approach able to identify whether source code has been generated by AI.

We evaluated GPTSniffer on two datasets collected from GitHub and ChatGPT. In the evaluation, we studied how characteristics of the training and test, and preprocessing steps impact the prediction performance. Also, we empirically compare GPTSniffer with GPTZero and OpenAI Text Classifier. The experimental results reveal interesting outcomes, while GPTSniffer cannot work well given that the training data and testing data are collected from completely independent sources, it obtains a perfect prediction by most of the configurations, where there are pairwise relationships between code written by humans and generated by ChatGPT.

The main contributions of our work are summarized as follows:

- A novel approach–named GPTSniffer–to the recognition of source code generated by ChatGPT.
- An empirical evaluation and comparison with two state-of-the-art baselines, i.e., GPTZero and OpenAI Text Classifier.
- The tool developed and the datasets curated through this work are made available to allow for future research (Nguyen et al., 2023a).

**Paper Structure.** Section 2 provides a motivating example, and the proposed approach is described in Section 3. Section 4 presents the materials and methods used to conduct an empirical evaluation on the proposed approach. Afterwards, Section 5 reports and analyzes the experimental results. We have some discussion and highlight the threats to validity in Section 6. The related work is reviewed in Section 7, and the paper is concluded in Section 8.

## 2. Background and motivations

As outlined in the introduction, concerns related to security, copyright/licensing infringement, or education ethics make it particularly important to identify whether a snippet has been generated by an AI.

In principle, some solutions to cope with this problem exist. For example, GPTZero is one of the existing systems designed to automatically detect text generated by OpenAI technologies. However, by testing GPTZero on source code, we notice that the outcome is far from satisfactory, suggesting how a well-defined text classifier fails to detect the origin of source code.

Fig. 1 shows an example with two code snippets, which are implemented exactly for the same purpose, i.e., displaying the maximum consecutive increasingly ordered substring.[3] However, one of them is written by humans (Fig. 1(a)), and the other one is generated by ChatGPT (Fig. 1(b)). The snippets look pretty standard, i.e., they use common API calls, such as `chatAt()`, `substring()`, or the `java.util.Scanner` package. Essentially, it is not easy to spot any concrete sign that can be used to recognize the source code's origin.

We fed the code in Figs. 1(a) and 1(b) to GPTZero, one by one, and asked for identification. Surprisingly, the platform gave the same conclusion for both snippets, i.e., "*Your text is likely to be written entirely by a human*". This means that the system wrongly classifies the second snippet. Moreover, GPTZero also added a remark, saying that: "*Sentences highlighted are more likely to be written by AI*". Such sentences, i.e., lines of code, are marked using yellow in Fig. 1. By comparing the designated parts in both snippets, we see that GPTZero evaluates many common code lines as written by AI, e.g., `public static void main(String[] args)` (Line 3) or `System.out.print(‘‘Enter a string:’’);` (Line 5). This is interesting as these lines can be written by both humans and ChatGPT.

[3] The original snippet is available online: https://bit.ly/3MZCDWy.

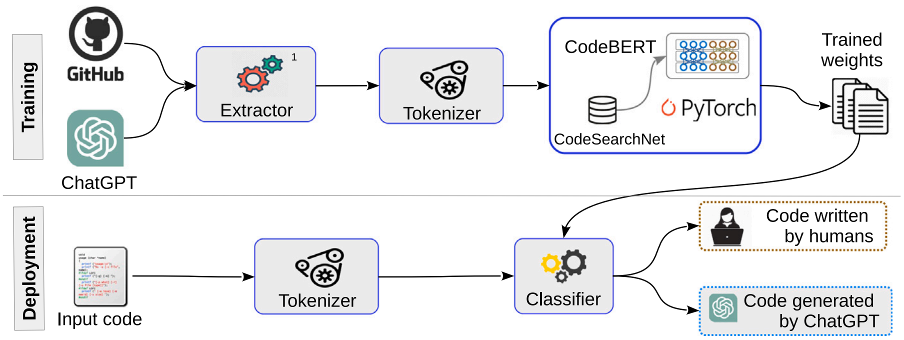


**Fig. 2.** System components.

Moreover, affirming that the snippets are "*written **entirely** by a human*", while still highlighting lines that "*are more likely to be written by AI*", is somewhat contradictory, rendering the classification result even more confusing.

By further testing GPTZero with more code from humans and ChatGPT, we witnessed similar outcomes. One likely explanation is that, while the underlying GPT model of GPTZero has been trained on a large corpus of text from the Internet (including source code too), it has not been specifically fine-tuned for source code. Altogether, we see room for improvement, i.e., the pattern in which commands are written, or the way comments are generated, are among distinguishable features that can be used to detect the origin of a snippet. This motivates us to investigate how well specifically-trained models can effectively recognize AI-generated code, as it is described in the rest of the paper.

## 3. Proposed approach

This section describes in detail GPTSniffer—a practical approach to distinguish code written by humans from that generated by ChatGPT, leveraging state-of-the-art pre-trained models. While we aim for a generic architecture with the adoption of various classification engines, the first version of GPTSniffer is built on top of CodeBERT, inheriting the well-defined technical foundation from the underlying pre-trained model. The ultimate aim is to achieve an ideal classification outcome for source code written in different languages.

As shown in Fig. 2, the GPTSniffer architecture consists of three main components, i.e., *Extractor*, *Tokenizer*, and *Classifier*. To train the classification engine, input data is collected from two data sources, i.e., GitHub and ChatGPT: while the former is a huge store of human-written code, the latter provides code generated by AI. Through the *Extractor* component, the input data is then undergone different preprocessing steps to enrich the training corpus. *Tokenizer* is employed to encode data for providing input to the *Classifier* component, which performs the training to yield the final model. Such a model can then be used to perform a prediction for unseen code snippets. The GPTSniffer components are described in the following subsections.

### 3.1. Extractor

Empirical studies have shown that source code features such as package names, class names, code comments, or import directives are the unique features to identify the so-called *coding style* (Bosu et al., 2015; Li et al., 2022; Ogura et al., 2018). We conjecture that style-related features can be an effective means to distinguish snippets written by humans from those generated by ChatGPT.

The *Extractor* component collects and prepares suitable data to train GPTSniffer. The ultimate aim is to create different derivations of the original code snippets, allowing the classifier to learn from diverse coding styles. The *Extractor* implements a set of rewriting rules, defined by adopting regular expressions. Given that an artifact, e.g., imports, package names, or code comments, matches the regular expression, it can either be removed or replaced with one that resembles a certain coding style. It is important to note that the *Extractor* currently provides the rewriting rules that allow us to create the settings defined in Section 4.5, i.e., rewriting class names, and removing package definition, imports, and comments. As an example, Fig. 3 depicts two rewriting rules used by the *Extractor* component, and the remaining ones are provided in our online replication package for reference.

### 3.2. Tokenizer

Once preprocessed by the *Extractor*, the source code is provided as input to the *Tokenizer* component, which transforms the code into a proper format that can then be consumed by CodeBERT. In particular, the input code is split into independent units called tokens, and padded with signaling tokens to separate the snippet from each other.

For instance, given the class in Fig. 4, the *Tokenizer* parses and transforms it into the sequence shown in Listing 1.

**Listing 1:** The transformed sequence.

```
BOS, public, class, Example, {, public, static,
    void, main, (, String, [, ], args, ), {, int, x,
    =, 5, ;, int, y, =, 7, ;, int, z, =, x, +, y, ;,
    System, ., out, ., println, (, z, ), ;, }, },
    EOS
```

in which BOS and EOS are the two special tokens to signal the beginning and end of the sequence. The resulting sequence is then fed as input to the classification to perform the training and prediction.

### 3.3. Classifier

CodeBERT has been pre-trained on CodeSearchNet (Husain et al., 2019), a code search dataset with more than 2M bimodal code-documentation pairs and 6.4M unimodal code snippets written in different languages, including Java, and Python. *Classifier* is built on top of CodeBERT,[4] and run with Pytorch. In this respect, it inherits the well-founded technical features from the original model. Starting from the sequence of tokens generated by *Tokenizer*, *Classifier* uses a series of encoding layers to transform it into a fixed-length vector representation. Each encoding layer performs a series of computations on the input sequence to generate a new sequence of vectors that captures different aspects of the input's context.

## 4. Empirical study design and methodology

This section describes the empirical evaluation to study the GPTSniffer ability to detect ChatGPT-generated snippets, and investigate the factors that impact on such ability.

[4] We make use of the CodeBERT pre-trained model provided by Huggingface (https://huggingface.co/microsoft/codebert-base).

```
C2: ’\n’.join([x if not x.startswith(‘package’)
        else ‘’ for x in code.splitlines()])
C3: res = re.sub(r‘(import(.)*ch_(.)*;)‘’,code)
```

**Fig. 3.** Example of the rewriting rules.

```
public class Example {
 public static void main(String[] args) {
  int x = 5;
  int y = 7;
  int z = x + y;
  System.out.println(z);
 }
}
```

**Fig. 4.** Example input code.

### 4.1. Research questions

We study the performance of GPTSniffer through a series of experiments to answer the following research questions:

- **RQ$_1$**: *How do the input data and the preprocessing settings impact on the GPTSniffer prediction performance?* Using three datasets collected from GitHub and ChatGPT, we conducted a series of experiments to identify the characteristics of the training data that can influence the accuracy of GPTSniffer, also under different preprocessing configurations.
- **RQ$_2$**: *To which extent can GPTSniffer detect ChatGPT-generated source code on the paired dataset under different preprocessing settings?* In this case we put GPTSniffer under a particularly favorable scenario, i.e., the presence of paired snippets (human vs. AI) in the training data, and investigated how GPTSniffer would perform in such a scenario under different code preprocessing configurations. This also aims to test the ability of GPTSniffer to detect code altered by prompt engineering.
- **RQ$_3$**: *How does GPTSniffer compare with GPTZero and OpenAI Text Classifier in recognizing ChatGPT-generated source code?* Using a set of common queries, we compare GPTSniffer with GPTZero (GPTZero, 2023) and OpenAI Text Classifier (Classifier, 2023) two state-of-the-art systems for identifying whether a text is generated by AI, including ChatGPT, or written by humans.

### 4.2. Data collection

To simulate real-world scenarios, we collected data from different sources, as GPTSniffer is expected to detect code written by different developers. Also, we considered source code obtained by querying ChatGPT under different conditions. The retrieval was performed following the process depicted in Fig. 5, i.e., we conducted two separate phases to obtain both unpaired and paired snippets, explained as follows.

#### 4.2.1. Unpaired snippets ($\mathcal{U}$)

To test the generalizability of GPTSniffer, we generated a set of additional queries to fetch code from ChatGPT. Such queries cover a wide range of tasks, from simple to complex ones, aiming to study the usefulness of GPTSniffer. The final corpus consists of 137 snippets, and they are summarized in Table 1. We gather representative samples from several common programming fields, such as Algorithms, File management, Networks, Search & sort, to name a few. This aims at ensuring a balanced distribution, as well as diversity among different types of tasks. We have established queries that do not require any contextual knowledge on the project under development, and their outcomes can be implemented independently. An example of a query is as follows: "*Can you write a Java program to implement the binary search algorithm?*"

Moreover, we independently collected **137 human-written snippets** being most starred from GitHub Gist.[5] The GitHub API provides the functionality to determine the most starred Java gists.[6] Gists containing fewer than 100 lines of code were excluded from the filtering process. To gather a comparable volume of AI-generated data, a total of 137 snippets that have received the highest number of stars were ultimately extracted.

Unlike the paired snippets—that will be described in Section 4.2.2, the unpaired ChatGPT-generated and human-written snippets are not related to each other. This simulates real use cases, where data is supposed to be collected from various repositories, and there exists no pairwise relationship between the snippets.

#### 4.2.2. Paired snippets ($\mathcal{P}$)

We consider problem implementations from a book on Java programming (Liang, 2003). Such a book has a supporting GitHub repository (Dulaney, 2023), storing the proposed solutions to the end-of-chapter exercises. We suppose that these snippets might have been used as training data for ChatGPT, though we have no trace of that. Each code snippet is associated with a task assignment, placed at the beginning of the snippet as a source code comment. An example of such an assignment is shown in Fig. 6.

By manually scrutinizing them, we noticed that not all of the snippets are eligible for our experiments, as there are many of them containing only the assignment, without any code. Thus, these were not selected for the experiments. Eventually, we obtained a corpus containing **601 human-written snippets**.

Starting from these human-written snippets, we extracted the task assignments and used them as queries. The queries were then split among the co-authors of this paper, that directly interacted with ChatGPT to retrieve generated solutions to the assignments. Fig. 7 shows an example of interacting with ChatGPT for the corresponding query transformed from the task assignment in Fig. 6. In particular, the prompt is formulated as follows: "*Write a java program that sorts an ArrayList of elements.*"

After this step, we got a corpus of **609 ChatGPT-generated snippets**. Although the queries have been extracted from 601 human-written snippets, we got a few ChatGPT implementations more as some of them are split among different snippets.

#### 4.2.3. Prompt engineered snippets ($\mathcal{E}$)

Large language models (LLMs), including ChatGPT, have demonstrated their ability to mimic language styles (Ozkaya, 2023). To mitigate potential biases, and ensure that the code does not evidently appear as computer-generated, we deliberately crafted the code generated by ChatGPT with prompt engineering (Wang et al., 2023) as shown in Table 2, by following two different strategies as given below.

- Prompts with a related ChatGPT interaction history $\mathcal{E}_1$: To assess the effect of masking the programming style in the query, we curated an additional test dataset comprising 100 task assignments randomly chosen from $\mathcal{P}$. After that, we reopened the corresponding chat threads, and asked ChatGPT to rewrite the code snippets to make them appear more resemble to code written by real developers.

[5] https://gist.github.com/discover

[6] https://docs.github.com/en/free-pro-team@latest/rest/gists/gists?apiVersion=2022-11-28#list-starred-gists

**Table 1**
Summary on the additional queries.

| Domain | Description | # snippets |
|---|---|---|
| Algorithms | Feed-forward neural networks, convolutional neural networks, graph neural networks, Boyer Moore algorithm, Dijkstra algorithm, greatest common division, Levenshtein, logistic regression, matrix multiplication, Rocchio algorithm, SVM | 35 |
| Files & Folders | Append text, concatenate file, Excel file, read file, read file list, write file, copy file, get modified time, MP3 file | 25 |
| Networks | Email, HTTP client/server, FTP client/server, chat client/server | 20 |
| Search & Sort | Binary search, exponential search, sequential search, breadth-first search, depth-first search, linear search, bubble sort, merge sort | 22 |
| Strings & Arrays | Array blocking issue, compare two strings, delete word, dequeue, common elements, minimum element | 15 |
| Others | Binary tree, collaborative filtering, content-based filtering, hash table, lunar calendar, password checker, quadratic equation | 20 |
| | Total | **137** |

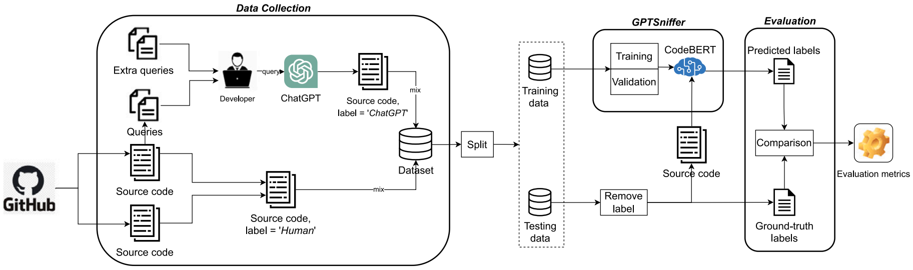
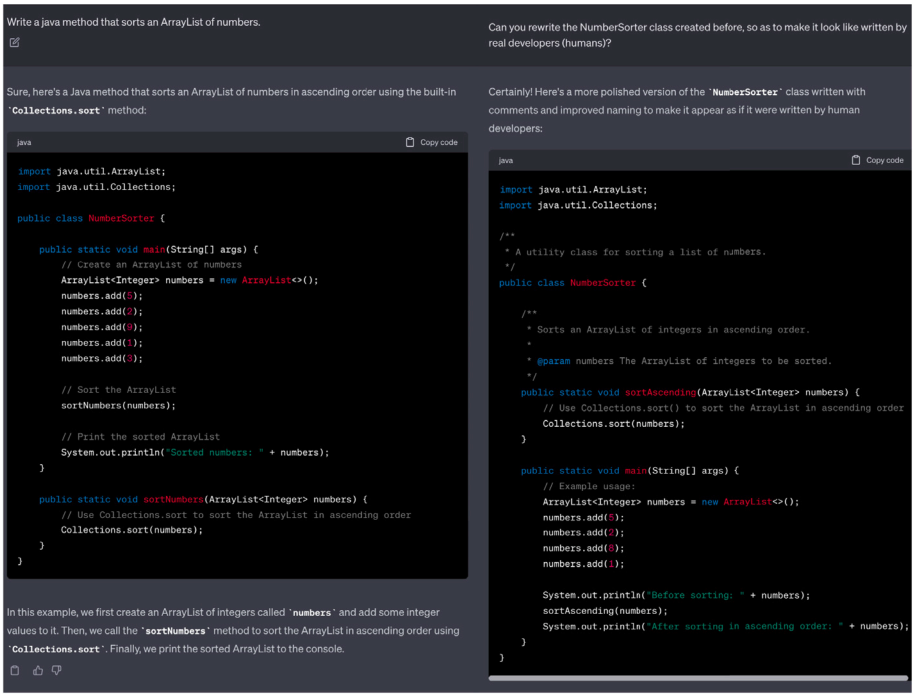


**Fig. 5.** The evaluation process.

```
/**
* *17.6 (Store Loan objects) The Loan class in Listing 10.2 does not implement
* Serializable.
* Rewrite the Loan class to implement Serializable. Write
* a program that creates five Loan objects and stores them in a file named
* Exercise17_06.dat.
*/
```

**Fig. 6.** Example of a code assignment.



Write a java method that sorts an ArrayList of numbers.

Sure, here's a Java method that sorts an ArrayList of numbers in ascending order using the built-in `Collections.sort` method:

```java
import java.util.ArrayList;
import java.util.Collections;

public class NumberSorter {

    public static void main(String[] args) {
        // Create an ArrayList of numbers
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(2);
        numbers.add(9);
        numbers.add(1);
        numbers.add(3);

        // Sort the ArrayList
        sortNumbers(numbers);

        // Print the sorted ArrayList
        System.out.println("Sorted numbers: " + numbers);
    }

    public static void sortNumbers(ArrayList<Integer> numbers) {
        // Use Collections.sort to sort the ArrayList in ascending order
        Collections.sort(numbers);
    }
}
```

In this example, we first create an ArrayList of integers called `numbers` and add some integer values to it. Then, we call the `sortNumbers` method to sort the ArrayList in ascending order using `Collections.sort`. Finally, we print the sorted ArrayList to the console.

Can you rewrite the NumberSorter class created before, so as to make it look like written by real developers (humans)?

Certainly! Here's a more polished version of the `NumberSorter` class written with comments and improved naming to make it appear as if it were written by human developers:

```java
import java.util.ArrayList;
import java.util.Collections;

/**
 * A utility class for sorting a list of numbers.
 */
public class NumberSorter {

    /**
     * Sorts an ArrayList of integers in ascending order.
     *
     * @param numbers The ArrayList of integers to be sorted.
     */
    public static void sortAscending(ArrayList<Integer> numbers) {
        // Use Collections.sort() to sort the ArrayList in ascending order
        Collections.sort(numbers);
    }

    public static void main(String[] args) {
        // Example usage:
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(2);
        numbers.add(8);
        numbers.add(1);

        System.out.println("Before sorting: " + numbers);
        sortAscending(numbers);
        System.out.println("After sorting in ascending order: " + numbers);
    }
}
```

**Fig. 7.** Interacting with ChatGPT to retrieve source code.

**Table 2**
Prompts to ask ChatGPT to disguise the code.

| Alias | Prompt |
|---|---|
| T1 | Please hide ChatGPT's involvement and mimic a beginner programmer's style, introducing imperfections, redundancies, and excessive details. |
| T2 | Ensure this does not seem like ChatGPT's work. Emulate a novice coder with slight imperfections, redundancies, and overly descriptive code. |
| T3 | Keep ChatGPT's authorship discreet. Imitate a beginner's coding style by incorporating imperfections, redundancies, and excessive details. |
| T4 | Conceal ChatGPT's contribution and adopt a beginner programmer's style, introducing imperfections, redundancies, and unnecessary details. |
| T5 | Obscure the fact that ChatGPT wrote this and simulate the programming style of a beginner with slight imperfections, redundancies, and overly descriptive code. |

- Prompts with fresh attempts (no ChatGPT interaction history) $\mathcal{E}_2$: Moreover, we also tested with fresh attempts to diversify the generated snippets. In particular, we explicitly asked ChatGPT to generate code with intention to disguise the real generators (see Table 2). However, different from $\mathcal{E}_1$, the snippets in this set are generated without linking to the previous chat threads.

We investigate the degree of variation in prompt-engineered code snippets compared to their original counterparts (the one generated with the original query).

Once we obtained the three initial sets of snippets, we populated four datasets as shown in Table 3, and explained as follows.

- Dataset $D_\alpha$: We shuffled all the snippets in Sections 4.2.1 and 4.2.2, and split again to distribute the snippets coming from different sources into balanced parts. This aims to simulate real-world scenarios, where either human-written or ChatGPT code can be collected in different ways. The resulting *mixed dataset* $D_\alpha$ consists of 1484 human-written and ChatGPT-generated snippets.
- Dataset $D_\beta$: We considered only the paired snippets related to the book's implementation in Section 4.2.2, resulting in the *paired dataset* $D_\beta$ with 1210 snippets.
- Dataset $D_\gamma$: All the snippets in $D_\beta$ are used for training, while the snippets in $\mathcal{E}_1$ are for testing the ability of GPTSniffer to detect code generated with prompt engineering. $D_\gamma$ contains in total a set of 1310 snippets.
- Dataset $D_\delta$: All the snippets in $D_\beta$ are used for training, while the snippets in $\mathcal{E}_2$ are for testing the ability of GPTSniffer to detect code generated with prompt engineering, using no history of interactions. $D_\delta$ contains in total a set of 1510 snippets.

It is worth noting that $D_\alpha$, $D_\beta$, $D_\gamma$, and $D_\delta$ are not independent, as $D_\alpha$ is a combination of re-shuffled $D_\beta$, plus additional data; while $D_\gamma$ is made of $D_\beta$ together with the prompt engineered snippets $\mathcal{E}_1$; and eventually $D_\delta$ is curated on top of $D_\beta$ with the fresh prompt engineered snippets $\mathcal{E}_2$. The goal of having different types of datasets is to study the generalizability of GPTSniffer in detecting code coming from heterogeneous sources. By counting the number of lines of code (LOC) for all the snippets collected from humans and ChatGPT by unpaired $\mathcal{U}$, and paired $\mathcal{P}$ snippets, we see that most of the snippets have a small LOC, i.e., lower than 80. Only a few of them are longer than 100 LOC. As for the code written by humans, there are some considerably long snippets, with up to 1200 LOC. More details about the distribution of the LOCs are available in Fig. 8.

**Table 3**
Number of collected code snippets.

| Snippets | ChatGPT | Humans |
|---|---|---|
| Unpaired snippets $\mathcal{U}$ | 137 | 137 |
| Paired snippets $\mathcal{P}$ | 609 | 601 |
| Prompt engineered snippets $\mathcal{E}_1$ | 100 | – |
| Prompt engineered snippets $\mathcal{E}_2$ | 300 | – |
| $D_\alpha = \mathcal{U} \cup \mathcal{P}$ | 1484 | |
| $D_\beta \equiv \mathcal{P}$ | 1210 | |
| $D_\gamma \equiv \mathcal{P} \cup \mathcal{E}_1$ | 1310 | |
| $D_\delta \equiv \mathcal{P} \cup \mathcal{E}_2$ | 1510 | |

## 4.3. Comparison with the baselines

The comparison with the baselines was performed using the paired snippets $\mathcal{P}$ (see Section 4.2.2) for which each human-written snippet has a counterpart generated by ChatGPT.

GPTZero and OpenAI Text Classifier are different with respect to the length of the input data they can handle. OpenAI Text Classifier accepts only text with more than 1000 characters, and GPTZero can handle shorter text with at least 250 characters. For this reason, we had to create two separate lists of queries. In particular, for comparing GPTZero with GPTSniffer, 50 snippets of small size were chosen for each of the two categories "*Human*", and "*ChatGPT*". For comparing OpenAI Text Classifier with GPTSniffer, there were 50 snippets of more than 1000 characters for each of the two categories "*Human*", and "*ChatGPT*".

## 4.4. Evaluation settings and metrics

We split the data using the 80:10:10 ratio, i.e., 80%, 10%, and 10% of the data are used for training, validation, and testing, respectively. For each testing snippet, before being fed as input to the prediction engine, its real category, i.e., either "*Human*" or "*ChatGPT*" is removed to use as ground-truth data. For every testing snippet, we evaluated it by comparing its actual category with the predicted one returned by GPTSniffer, and computed the number of True positives (TP), False positives (FP), False negatives (FN), and True negatives (TN) (Dalianis, 2018).

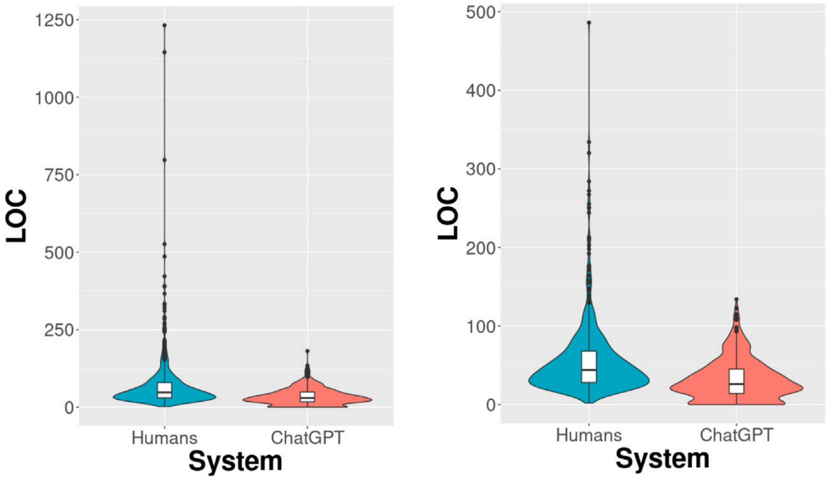
- *True positives* (TP): the snippets that are classified to their correct category;
- *False positives* (FP): the snippets classified to a category that does not match with the real one;
- *False negatives* (FN): the snippets that should have been classified into a category, but they are classified to the other category;
- *True negatives* (TN): the snippets that are not classified to a category and they also do not belong to that category.

The final performance is evaluated using Accuracy, Precision, Recall, and $F_1$-score, defined as follows.

### 4.4.1. Accuracy

It measures the ratio of correctly classified snippets to the total number of snippets for all the considered categories, computed as follows.

$$Accuracy = \frac{|TP + TN|}{|TP + TN + FP + FN|}$$



(a) $D_\alpha$

(b) $D_\beta$

**Fig. 8.** Number of lines of code (LOC) for the considered datasets.

### 4.4.2. Precision, recall, and $F_1$-score

Given a category, Precision measures the fraction of correctly classified items to the total number of items; Recall is the ratio of actual positive cases that are correctly classified; $F_1$-score (or $F_1$) is a harmonic combination of the two aforementioned metrics.

$$P = \frac{|TP|}{|TP + FP|};$$

$$R = \frac{|TP|}{|TP + FN|};$$

$$F_1 = \frac{2 \times P \times R}{P + R}$$

In the evaluation, we also make use of *macro average*, and *weighted average* score of these metrics. The former is the arithmetic mean of all the scores for the two categories, while the latter weighs the varying degree of importance of the categories in a dataset.

In $RQ_3$, to compare GPTSniffer with GPTZero and OpenAI Text Classifier, we use McNemar's test (McNemar, 1947), which is a proportion test for paired samples. As we perform multiple comparisons, $p$-values are adjusted using Holm's correction (Holm, 1979). McNemar's test is complemented by the Odds Ratio (OR) effect size measure.

### 4.4.3. Levenshtein ratio

The Levenshtein distance (Levenshtein, 1966) provides a numerical value representing the number of single-character edits required to change the original snippet into its generated counterpart. We used such a metric to quantify the changes made in the code generation process, as this metric has been used for the same purpose (Nguyen et al., 2021; Nguyen et al., 2023b). The metric is defined below.

$$L(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ L(tail(a), tail(b)) & \text{if } head(a) = head(b), \\ 1 + min \begin{cases} L(tail(a), b), \\ L(a, tail(b)), \\ L(tail(a), tail(b)) \end{cases} & otherwise. \end{cases}$$

Where $head(s)$ is the first character of string $s$, and $tail(s)$ is $s$ minus its first character.

The Levenshtein ratio (Yujian and Bo, 2007) normalizes the Levenshtein distance by the max length of the two sequences being compared, providing a measure of similarity that ranges from 0.0 to 1.0. A ratio of 1.0 indicates that the sequences are identical, while a ratio of 0.0 means that the sequences are completely different. The ratio is computed as follows:

$$Levenshtein\ ratio(a,b) = 1 - \frac{L(a,b)}{max(|a|,|b|)}$$

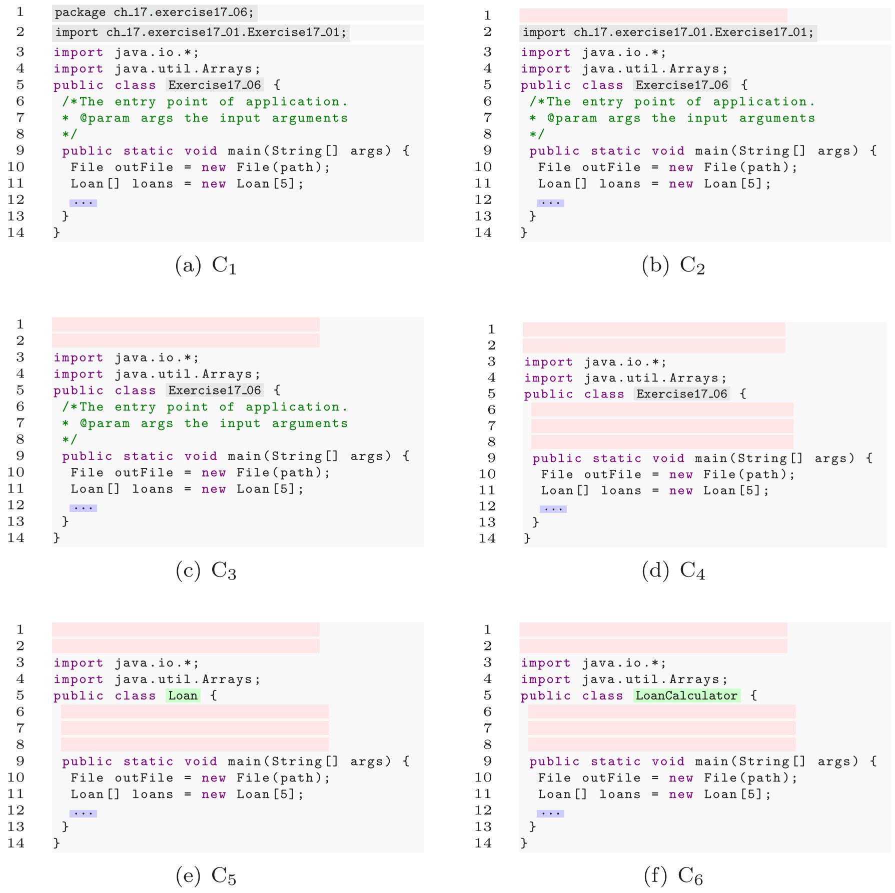
We use the metric since the original code snippets are of different lengths, making unnormalized distances confusing and not comparable. The Levenshtein ratio is computed after dropping multiple spaces, from snippets generated by original and prompt-engineered queries.

## 4.5. Configurations

Table 4 shows the nine experimental configurations, which are indeed not exhaustive, as we cannot consider all possible combinations of artifacts. Thus, we pay attention only to those most representative and realistic, as explained below. We use the check mark symbol ✔ to indicate that the corresponding feature is kept; and a uncheck mark symbol ✖ to signal the opposite, i.e., removing the feature; the hand-written symbol ✍ represents a modification in the feature, where ✍G means the original name is replaced by that coming from the corresponding ChatGPT snippet, and ✍H signals that such a name is replaced by humans.

By default, ChatGPT never generates a package name (we noticed this after several attempts of interacting with the platform), thus with package definition, we only consider snippets written by humans. Fig. 9 illustrates different snippets corresponding to the considered configurations shown in Table 4, and explained as follows:

- **$C_1$**: We keep the code by ChatGPT and human unchanged, and run the experiments with the code as it is. An example of such code is shown in Fig. 9(a).
- **$C_2$**: In this configuration, the package definition from the human-written code is removed. As shown in Fig. 9(b), compared to Fig. 9(a), `package ch_17.exercise17_06` is no longer seen.
- **$C_3$**: From **$C_2$**, imports to project packages are dropped. The code in Fig. 9(c) is similar to that in Fig. 9(b), however the import directive `import ch_17.exercise17_01.Exercise17_01` is removed.
- **$C_4$**: From $C_3$, comments embedded in the code by of humans and ChatGPT are deleted. Fig. 9(c) shows the snippet written by humans but without code comments.
- **$C_5$**: We conjecture that strings related to the hierarchical names, e.g., `Exercise17_06`, might be a discriminant feature, creating a bias in the prediction performance. Thus, from the human-written code in **$C_4$**, we replace the class name with that of the corresponding snippet written by ChatGPT (Fig. 9(e)).



```
package ch_17.exercise17_06;
import ch_17.exercise17_01.Exercise17_01;
import java.io.*;
import java.util.Arrays;
public class Exercise17_06 {
 /*The entry point of application.
 * @param args the input arguments
 */
 public static void main(String[] args) {
  File outFile = new File(path);
  Loan[] loans = new Loan[5];
  ...
 }
}
```

(a) C1

```

import ch_17.exercise17_01.Exercise17_01;
import java.io.*;
import java.util.Arrays;
public class Exercise17_06 {
 /*The entry point of application.
 * @param args the input arguments
 */
 public static void main(String[] args) {
  File outFile = new File(path);
  Loan[] loans = new Loan[5];
  ...
 }
}
```

(b) C2

```


import java.io.*;
import java.util.Arrays;
public class Exercise17_06 {
 /*The entry point of application.
 * @param args the input arguments
 */
 public static void main(String[] args) {
  File outFile = new File(path);
  Loan[] loans = new Loan[5];
  ...
 }
}
```

(c) C3

```


import java.io.*;
import java.util.Arrays;
public class Exercise17_06 {



 public static void main(String[] args) {
  File outFile = new File(path);
  Loan[] loans = new Loan[5];
  ...
 }
}
```

(d) C4

```


import java.io.*;
import java.util.Arrays;
public class Loan {



 public static void main(String[] args) {
  File outFile = new File(path);
  Loan[] loans = new Loan[5];
  ...
 }
}
```

(e) C5

```


import java.io.*;
import java.util.Arrays;
public class LoanCalculator {



 public static void main(String[] args) {
  File outFile = new File(path);
  Loan[] loans = new Loan[5];
  ...
 }
}
```

(f) C6

**Fig. 9.** Different configurations for human-written code.

**Table 4**
Experimental configurations.

| Artifact | Configurations | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | $C_9$ |
| Package definition | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | P |
| Self-made class name | ✔ | ✔ | ✔ | ✔ | ✍G | ✍H | ✍H | ✍H | P |
| Imports to self-made packages | ✔ | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ | ✖ | P |
| Code comments | ✔ | ✔ | ✔ | ✖ | ✖ | ✖ | ✖ | ✖ | P |
| All imports | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✖ | ✖ | P |
| \t and \n | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✖ | P |

- **$C_6$**: We attempt to make the code more human-like by giving a name that reflects well the task. From the human-written code in **$C_4$**, the co-authors of this paper read the task assignment, and renamed the class with a name that reflects better the task (Fig. 9(f)).
- **$C_7$**: From the human-written code in **$C_6$**, and the ChatGPT-generated code in **$C_3$**, we removed all the imports statements.
- **$C_8$**: From the human-written and the ChatGPT-generated code in **$C_7$**, we removed all the formatting characters, including \t and \n. For the sake of clarity, we do not display a figure to illustrate the code examples for **$C_7$** and **$C_8$**. Further examples are available in our online appendix (Nguyen et al., 2023a).
- **$C_9$**: Finally, we consider prompt engineering as a special configuration, in which we ask ChatGPT to alter the generated code, making it look like as it were written by humans. **$C_9$** is independent from the remaining configurations, as we do not change the code, but ChatGPT does. Thus, in Table 4, we mark all the rows corresponding to **$C_9$** with "P" to signal the differences.

In the experiments, we executed GPTSniffer on the datasets along the aforementioned configurations. The obtained results are reported and analyzed in the next section.

## 5. Results

This section reports the results of the study addressing the research questions formulated in Section 4.1. We experiment GPTSniffer on the two datasets in Section 4.2, compared it with the baselines introduced in Section 4.3. The prediction performance is measured by means of the evaluation metrics in Section 4.4.

### 5.1. *RQ$_1$: How do the input data and the preprocessing settings impact on the GPTSniffer prediction performance?*

In this research question, we study the effect of the training data on the accuracy of GPTSniffer. This aims at investigating the extent to which GPTSniffer can cope with real-world situations, where code snippets are supposed to be of different origins, and written by many developers, rendering the identification more difficult. To this end,

**Table 5**
$\mathcal{P}$ for training, and $\mathcal{U}$ for testing.

| | Precision | Recall | $F_1$ score | Support (#) |
|---|---|---|---|---|
| ChatGPT | 0.57 | 1.00 | 0.73 | 120 |
| Humans | 1.00 | 0.25 | 0.40 | 120 |
| Accuracy | | | 0.62 | 240 |
| Macro avg | 0.79 | 0.62 | 0.56 | 240 |
| Weighted avg | 0.79 | 0.62 | 0.56 | 240 |

**Table 6**
$D_\alpha$: Precision.

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | # |
|---|---|---|---|---|---|---|---|---|---|
| ChatGPT | 0.90 | 0.88 | 0.91 | 0.92 | 0.93 | 0.91 | 0.86 | 0.84 | 148 |
| Humans | 0.98 | **1.00** | **1.00** | **1.00** | 0.99 | **1.00** | **1.00** | 0.98 | 147 |
| Macro avg | 0.94 | 0.94 | 0.96 | 0.96 | 0.96 | 0.95 | 0.93 | 0.91 | 295 |
| Weighted avg | 0.94 | 0.94 | 0.96 | 0.96 | 0.96 | 0.95 | 0.93 | 0.91 | 295 |

**Table 7**
$D_\alpha$: Recall.

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | # |
|---|---|---|---|---|---|---|---|---|---|
| ChatGPT | 0.99 | **1.00** | **1.00** | **1.00** | 0.99 | **1.00** | **1.00** | 0.98 | 148 |
| Humans | 0.89 | 0.86 | 0.90 | 0.91 | 0.93 | 0.90 | 0.83 | 0.82 | 147 |
| Macro avg | 0.94 | 0.93 | 0.95 | 0.96 | 0.96 | 0.95 | 0.91 | 0.90 | 295 |
| Weighted avg | 0.94 | 0.93 | 0.95 | 0.96 | 0.96 | 0.95 | 0.92 | 0.90 | 295 |

we consider two use cases: *(i) Testing and training data come from independent sources*; *(ii) Testing and training data come from same sources*. Note that in the rest of this paper, the term "independent" means that snippets come from completely different datasets. It is always the case that there is no duplicate between the training and test set.

#### 5.1.1. Testing and training data come from independent sources

We first investigate how GPTSniffer performs when it is tested from a dataset coming from a completely different source/domain than the training set. To this aim, we use the unpaired snippets $\mathcal{U}$ (see Section 4.2.1) to test the system, which has already been trained with the paired snippets, i.e., $\mathcal{P}$ (see Section 4.2.2). Our goal is to replicate a realistic scenario where GPTSniffer will make predictions on snippets that were not used during training.

Table 5 reports the evaluation metrics for this experiment, considering the dataset without any preprocessing, i.e., $C_1$. The **Support (#)** column indicates the number of testing items for each category. Overall, GPTSniffer obtains a low prediction performance for both categories. While it achieves 1.00 as Recall for code written by ChatGPT, it yields 0.57 as Precision, resulting in 0.73 as $F_1$ score for the set of 120 testing instances. When detecting code written by humans, GPTSniffer also achieves a low Recall, i.e., 0.25, thus decreasing the corresponding $F_1$ score to 0.40.

In summary, the empirical evidence indicates that in the presence of completely different data sources between training and test sets, the prediction becomes challenging and results in mediocre performance.

#### 5.1.2. Testing and training data come from same sources

For this experiment, we use $D_\alpha$, where snippets in $\mathcal{P}$ and $\mathcal{U}$ are mixed and split again to distribute the snippets coming from different sources into balanced parts. The execution of GPTSniffer on $D_\alpha$ produced the results reported in Tables 6, 7, and 8 (with the best results being printed in bold), analyzed as follows.

As shown in Table 6, GPTSniffer is more precise in identifying human-written code than ChatGPT-generated code. For five out of nine configurations, GPTSniffer achieves 1.00 as Precision, and by the remaining three configurations, the corresponding values are 0.98, 0.99, and 0.98. Concerning macro and weighted average Precision scores, for $C_8$ the tool yields the lowest performance, i.e., 0.91 as the overall macro average and weighted average Precision.

Table 7 reports the Recall scores for all the considered configurations. Over 148 ChatGPT-generated snippets, GPTSniffer yields a Recall of 1.00 for five out of nine configurations. At the same time, the corresponding scores obtained on the code written by humans are a bit lower. In particular, for four configurations, the Recall values for this category are below 0.90.

Table 8 reports the accuracy and $F_1$-scores obtained for all the configurations. Overall, the $F_1$-scores are greater than or equal to 0.90. Among the configurations, GPTSniffer gets the best accuracy, i.e., 0.96, by $C_4$ and $C_5$. This implies that once all the package definitions and code comments have been removed from the original snippets, GPTSniffer improves its performances. One possible interpretation of this phenomenon is that, on the one hand, package definitions just include recurring items present on both human-written and ChatGPT-generated code. On the other hand, comments (and natural language elements in general) may not contain features that GPTSniffer can leverage to successfully perform a classification.

To sum up, on the one hand, GPTSniffer does not perform well when tested on code belonging to a completely different domain dataset. But on the other hand, its performance considerably improves when the common patterns – those that may occur in data curated from the same domains – have been learned during the training.

**Answer to $\mathbf{RQ_1}$.** GPTSniffer cannot recognize well data originating from completely extraneous sources than the one it has been trained with. At the same time, its performance metrics are all at least 90% when it is trained with data from a source seen before.

### 5.2. $\mathbf{RQ_2}$: *To which extent can GPTSniffer detect ChatGPT-generated source code on the paired dataset under different preprocessing settings?*

The effectiveness of GPTSniffer is important in the context of detecting code written by humans and AI. This research question is dedicated to study how GPTSniffer can deal with different types of preprocessing conducted on the input data, i.e., $\mathbf{C_1}$-$\mathbf{C_8}$, and when the generated snippets are changed with prompt engineering, i.e., $\mathbf{C_9}$. This aims at investigating the extent to which GPTSniffer can work in real-world situations, where code snippets are supposed to be of different origins, and written by many developers, making the identification even more challenging.

**Table 8**
$D_{\alpha}$: Accuracy and $F_1$ score.

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | # |
|---|---|---|---|---|---|---|---|---|---|
| ChatGPT | 0.94 | 0.94 | 0.95 | 0.96 | 0.96 | 0.95 | 0.92 | 0.91 | 148 |
| Humans | 0.94 | 0.93 | 0.95 | 0.95 | 0.95 | 0.95 | 0.91 | 0.89 | 147 |
| Accuracy | 0.94 | 0.93 | 0.95 | **0.96** | **0.96** | 0.95 | 0.92 | 0.90 | 295 |
| Macro avg | 0.94 | 0.93 | 0.95 | **0.96** | **0.96** | 0.95 | 0.91 | 0.90 | 295 |
| Weighted avg | 0.94 | 0.93 | 0.95 | **0.96** | **0.96** | 0.95 | 0.91 | 0.90 | 295 |

**Table 9**
$D_{\beta}$: $C_1$, $C_2$, $C_3$, $C_5$, $C_6$, and $C_7$.

| | Precision | Recall | $F_1$ score | # |
|---|---|---|---|---|
| ChatGPT | 1.00 | 1.00 | 1.00 | 120 |
| Humans | 1.00 | 1.00 | 1.00 | 120 |
| accuracy | | | 1.00 | 240 |
| macro avg | 1.00 | 1.00 | 1.00 | 240 |
| weighted avg | 1.00 | 1.00 | 1.00 | 240 |

**Table 10**
$D_{\beta}$: $C_4$ and $C_8$.

| | Precision | Recall | $F_1$ score | # |
|---|---|---|---|---|
| ChatGPT | 0.99 | 0.98 | 0.99 | 120 |
| Humans | 1.00 | 0.99 | 0.99 | 120 |
| Accuracy | | | 0.99 | 240 |
| Macro avg | 0.99 | 0.99 | 0.99 | 240 |
| Weighted avg | 0.99 | 0.99 | 0.99 | 240 |

**Table 11**
$D_{\gamma}$: $\mathcal{P}$ for training, and $\mathcal{E}_1$ for testing.

| | Precision | Recall | $F_1$ score | Support (#) |
|---|---|---|---|---|
| ChatGPT | 1.00 | 1.00 | 1.00 | 100 |
| Accuracy | | | 1.00 | 100 |
| Macro avg | 1.00 | 1.00 | 1.00 | 100 |
| Weighted avg | 1.00 | 1.00 | 1.00 | 100 |

**Table 12**
$D_{\delta}$: $\mathcal{P}$ for training, and $\mathcal{E}_2$ for testing.

| | Precision | Recall | $F_1$ score | Support (#) |
|---|---|---|---|---|
| ChatGPT | 1.00 | 1.00 | 1.00 | 300 |
| Accuracy | | | 1.00 | 300 |
| Macro avg | 1.00 | 1.00 | 1.00 | 300 |
| Weighted avg | 1.00 | 1.00 | 1.00 | 300 |

**Table 13**
Dunn test, comparison of Levenshtein ratio by prompt type (Benjamini–Hochberg).

| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| **T2** | −2.567264<br>0.0256 | | | |
| **T3** | −0.589048<br>0.3088 | 1.939194<br>0.0656 | | |
| **T4** | −1.326484<br>0.1539 | 1.194060<br>0.1660 | −0.728344<br>0.2915 | |
| **T5** | −2.806451<br>0.0250 | −0.257222<br>0.3985 | −2.179626<br>0.0488 | −1.438930<br>0.1502 |

### *5.2.1. Manipulating training data*

After having investigated how GPTSniffer performs on completely different training and test sets, as well as on related ones, we experiment with the most favorable conditions, i.e., the use of a paired dataset ($D_{\beta}$), for which each human-written snippet is associated with a corresponding version generated by ChatGPT.

For this dataset, we achieve an almost perfectly consistent outcome, i.e., by six out of eight configurations, i.e., $C_1$, $C_2$, $C_3$, $C_5$, $C_6$, and $C_7$, the accuracy is 1.0, so are Precision, Recall, and $F_1$-score. For the sake of clarity, we report the results for these configurations using a common table, i.e., Table 9. The results indicate that on the paired dataset $D_{\beta}$, GPTSniffer obtains a perfect prediction for the majority of the considered configurations. In particular, from the table, it is evident that GPTSniffer does not miss any prediction for the testing instances, by both categories of snippets, i.e., Humans and ChatGPT.

Table 10 depicts the results obtained for $C_4$ and $C_8$, in which GPTSniffer exhibits a slightly lower accuracy compared to that of the other configurations shown in Table 9. In particular, the tool always gets 0.99 as macro average and weighted average for Precision, Recall, and $F_1$ score. In these configurations, GPTSniffer is better at detecting code by humans, compared to code by ChatGPT, i.e., the obtained Precision, Recall, and $F_1$ score are 1.00, 0.99, and 0.99 for human code. The corresponding scores for code generated by ChatGPT are 0.99, 0.98, and 0.99. Still, GPTSniffer can properly distinguish between code written by ChatGPT and humans in the two aforementioned configurations.

From the results of Table 9 and Table 10, we conclude that, for the paired dataset, GPTSniffer can properly tell apart code written by humans and AI. While the results of this scenario seem obvious, they acquired knowledge that can be leveraged to properly train GPTSniffer for applications in which it is expected to obtain several similar code snippets, e.g., to detect plagiarism in assignments for Software Engineering courses.

### *5.2.2. The ability to detect code altered with prompt engineering*

We consider the alteration by prompt engineering as a kind of preprocessing steps. Using the paired dataset, i.e., $D_{\beta}$ as training, and the whole set of 100 snippets with related history, i.e., $\mathcal{E}_1$, as queries, we obtained a stunning outcome. As shown in Table 11, all the snippets are correctly classified as written by ChatGPT, resulting in a perfect prediction, i.e., 1.00 for Precision, Recall, and $F_1$-score. In this respect, it is evident that even with snippets altered with prompts, GPTSniffer is still able to tell them apart, without mistaking them with the ones written by real developers.

Next, we used $D_{\beta}$ as training, and the whole set of 300 snippets with fresh prompts ($\mathcal{E}_2$), i.e., task assignments combined with prompts as queries. In Table 12, it is evident that all the snippets are correctly classified as written by ChatGPT, yielding 1.00 for Precision, Recall, and $F_1$-score. We conclude that even with attempts to disguise the code, GPTSniffer is still able to properly classify them.

The results in Table 11 and Table 12 show that GPTSniffer is capable of detecting well code written by ChatGPT, even after prompt engineering has been applied to make it look less computed-generated.

To further investigate the differences between the original snippets, and those that have been altered with prompt engineering, we compute the Levenshtein ratio between them.

Fig. 10 shows that the Levenshtein ration ranges from 25% to 75% for all the five types of prompts.

To study the distribution of Levenshtein distances across different classes, we also performed a Dunn test (Dinno, 2015). This non-parametric method was chosen to identify if there were statistically significant differences in the distributions of distances among the various classes. As the Dunn's test performs multiple comparisons, $p$-values are adjusted using the Benjamini–Hochberg correction (Benjamini and Hochberg, 1995). The test results shown in Table 13 reveal interesting clusters within the classifications. Using $\alpha = 0.05$ and $p <= \alpha/2$ to reject $H_0$, we can conclude that there are statistically significant

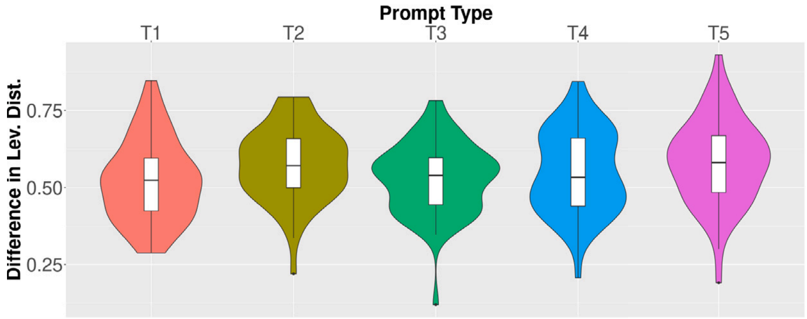


**Fig. 10.** Difference in Levenshtein distance with the five types of prompts in Table 2.

**Table 14**
$\mathcal{P}$ formatted with $C_8$ for training, and $\mathcal{E}_2$ formatted with $C_8$ for testing.

| | Precision | Recall | $F_1$ score | Support (#) |
|---|---|---|---|---|
| ChatGPT | 1.00 | 0.59 | 0.74 | 300 |
| Accuracy | | | 0.59 | 300 |
| Weighted avg | 1.00 | 0.59 | 0.74 | 300 |

**Table 15**
$\mathcal{P}$ formatted with $C_1$, $C_2$, $C_3$, $C_4$, $C_5$, $C_6$, $C_7$, and $C_8$ for training, and $\mathcal{E}_2$ formatted with $C_8$ for testing.

| | Precision | Recall | $F_1$ score | Support (#) |
|---|---|---|---|---|
| ChatGPT | 1.00 | 0.95 | 0.97 | 300 |
| Accuracy | | | 0.95 | 300 |
| Weighted avg | 1.00 | 0.95 | 0.97 | 300 |

differences in the code generated from different prompts. Specifically, snippets generated from prompt T1 are significantly different from those generated by prompt types T2 and T5. Moreover, snippets from T5 show a significant difference when compared to those from prompt T3. This implies that the observed differences are unlikely to be due to chance or random fluctuation. They possess a level of significance that allows statistical tests to identify them.

This outcome highlights the intricate dynamics of code modifications and their quantifiable impacts. The Levenshtein distance, coupled with the Dunn test, provides a valuable insight into these dynamics. It suggests that the variations in prompts indeed lead to the production of distinct code. In other words, the way a prompt is structured or phrased has a real, measurable impact on the output of the code generation process.

Despite these significant differences in the code generated from different prompts, the GPTSniffer performances remain unchanged. Future research may delve deeper into the specific types of modifications that lead to similar or divergent patterns in code snippet variations.

### 5.2.3. *The effect of the configurations on the accuracy*

To investigate the influence of different types of training data on the prediction, we removed all package names, and blank lines in the training and testing datasets ($\mathcal{P}$ and $\mathcal{E}_2$, respectively), using Configuration $C_8$ (see Table 4). The experimental results are shown in Table 14. Interestingly, it can be seen that the accuracy drops dramatically compared to the results presented before in Table 12. In particular, while precision is 1.00, the recall value only reaches 0.59, resulting in an F1 score of 0.74. We suppose that this happens due to the training data, i.e., GPTSniffer has not been trained with data that covers the possible formats of the code generated by ChatGPT.

To confirm our hypothesis, we augmented the training data by considering all the configurations presented in Table 4. To be concrete, we cumulatively add to the training set of GPTSniffer by formatting $\mathcal{P}$ with $C_1$, then $C_2$, and the other configurations up to $C_8$. This results in a final set with 1210 × 8 = **9680 samples**, which was then fed to GPTSniffer. Once we have finished training GPTSniffer with this dataset, we tested it with $\mathcal{E}_2$ whose snippets are formatted by removing all the blank lines and package definitions, i.e., $C_8$. The experimental results are depicted in Table 15.

It is evident that the prediction accuracy improves substantially with respect to the results in Table 14. GPTSniffer gets 0.95 as the accuracy, and this is indeed much better compared to 0.59, the corresponding score obtained for the case when the training data is limited to only one configuration. Altogether, this demonstrates that GPTSniffer benefits from training data altered with various formats. This implies a clear advantage of data augmentation (Shorten and Khoshgoftaar, 2019) in the classification of code generated by ChatGPT.

**Answer to RQ$_2$.** On the paired dataset, GPTSniffer obtains a perfect prediction by the majority of the experimental settings. This indicates that when being trained with pairwise code, our proposed tool works well as a detector for code written by ChatGPT. GPTSniffer is capable of detecting code changed with initial prompt engineering attempts, even though this ability should be further validated with more complex prompts. Augmenting the training data with different formats allows GPTSniffer to deal better with variability in the testing code.

## 5.3. RQ$_3$: *How does GPTSniffer compare with GPTZero and OpenAI Text Classifier in recognizing ChatGPT-generated source code?*

In this section, we report the comparison of GPTSniffer with GPTZero and OpenAI Text Classifier. For GPTSniffer, we selected the two most representative configurations, i.e., $C_1$ and $C_8$, as they correspond to the cases when GPTSniffer performs the best and the worst, respectively (see Table 9 and Table 10).

### 5.3.1. *Comparison with GPTZero*

The classification results returned by GPTZero are shown in Table 16. The first column reports the answer text returned by GPTZero, the second column shows a binary classification given by us to allow for a comparison with GPTSniffer, and the third column shows the number of queries. Most of the queries, i.e., 81 + 7 snippets, are classified as written by humans, and only 6 + 6 snippets are predicted as generated by ChatGPT.

By matching with the ground-truth data, we obtained the following classification results: 64 out of 100 snippets are correctly predicted by GPTZero, corresponding to an Accuracy of 0.64. The outcome obtained by GPTSniffer is as follows: among 100 snippets, 99 are correctly classified, i.e., Accuracy = 0.99. McNemar's test indicates a statistically significant difference ($p$-value $< 2e-09$), with an OR = 42 in favor of GPTSniffer. *This means that GPTSniffer significantly outperforms GPTZero in recognizing code generated by ChatGPT.*

**Table 16**
Classification results by GPTZero.

| Answer | Final class | # |
|---|---|---|
| Your text is likely to be written entirely by a human | Humans | 81 |
| Your text is most likely human written but there are some sentences with low perplexities | Humans | 7 |
| Your text is likely to be written entirely by AI | ChatGPT | 6 |
| Your text may include parts written by AI | ChatGPT | 6 |

**Table 17**
Classification results by OpenAI Text Classifier.

| Answer | Final class | # |
|---|---|---|
| The classifier considers the text to be unclear if it is AI-generated | Humans | 35 |
| The classifier considers the text to be unlikely AI-generated | Humans | 3 |
| The classifier considers the text to be likely AI-generated | ChatGPT | 20 |
| The classifier considers the text to be possibly AI-generated | ChatGPT | 42 |

### 5.3.2. Comparison with OpenAI Text Classifier

Table 17 depicts the classification results obtained by running with OpenAI Text Classifier. Similar to Table 16, the first column depicts the answer text given by OpenAI Text Classifier, the second column is the binary final classification given by us to compare with GPTSniffer, and the third column represents the number of queries. Among them, 35 + 3 snippets are marked as written by humans, and the rest, i.e., 20 + 42 snippets are predicted by OpenAI Text Classifier as generated by ChatGPT.[7]

Comparing with the real category of the query snippets, we see that 61 out of 100 are correctly predicted by OpenAI Text Classifier, resulting in an Accuracy of 0.61. The classification by GPTSniffer is almost perfect: 99 out of 100 snippets are correctly recognized, corresponding to an Accuracy of 0.99. McNemar's test indicates a statistically significant difference ($p$-value $< 2e-09$), with an OR=38 in favor of GPTSniffer.

Altogether, we can conclude that *the prediction by GPTSniffer is significantly better than the one provided by OpenAI Text Classifier*. Notably, OpenAI Text Classifier has a disclaimer saying that: "*The classifier isn't always accurate; it can mislabel both AI-generated and human-written text. AI-generated text can be edited easily to evade the classifier.*"

**Answer to RQ$_3$.** GPTSniffer considerably outperforms both GPTZero and OpenAI Text Classifier in the ability to detect if a code snippet is generated by ChatGPT.

## 6. Discussion

In this section, we discuss possible implications derived from the empirical results, as well as the threats to the validity of our findings.

### 6.1. Implications

**Usage.** We assume that GPTSniffer is useful due to various reasons. First, code produced by generative models might contain vulnerability (EuroPol, 2023; Pearce et al., 2021), as well as other issues, including copyright and licensing infringements (Reda, 2023; StephanieGlen, 2023). Second, in an educational context, using code snippets generated by ChatGPT, without understanding them, will hamper students' essential skills that can be acquired only through self-learning. Essentially, handing in code written by ChatGPT without self-work can be deemed a form of fraud. Altogether, it is of great importance to detect whether a source code element has been written by AI, owing to *(i)* the professional development side, coping with security and legal issues; and *(ii)* the educational side, combating cheating and plagiarism. We anticipate two possible use cases for GPTSniffer as follows. *First*, detecting AI-generated code plays an important role in code reviewing. In a development scenario, where, when the code is committed, our tool (for example implemented as a GitHub action) could warn that it is AI-generated code and recommend accurate/specific code reviews. *Second*, we use the tool to check code submitted by students at our universities, i.e., the University of L'Aquila and the University of Sannio.[8] Indeed, GPTSniffer will not be used as the sole means to fail students, but as a supporting tool. In particular, GPTSniffer can be used to classify students' code assignment, and in case the tool marks any snippets as written by ChatGPT, then examiners may question the corresponding student(s) for greater details. In this way, it is easy to find out if the students are really the authors of the snippets or not. In contrast, if the code is classified as written by humans, then teachers will possibly spend less effort to ask the submitters.

The first result emerging from our study is that training performed on completely different data may lead to sub-optimal results. This is analogous to what happens to other kinds of predictors in software engineering, for example, defect prediction models, for which a cross-project prediction performs well only when the training and test feature closely-related (e.g., in terms of metrics) code elements (Menzies et al., 2013). We assume that training GPTSniffer with data coming from different sources, as well as different types of programming tasks will enable us to boost the prediction accuracy.

**Finding 1.** GPTSniffer can be used in different scenarios, both in industrial and educational settings. To effectively recognize AI-generated code, classifiers like GPTSniffer need to be trained on source code being relevant to the context under consideration.

By experimenting GPTSniffer with different configurations, we noticed how certain preprocessing steps, such as removing comments or package names actually help to improve the performance of the classification. This is similar to data augmentation in Computer Vision (Shorten and Khoshgoftaar, 2019), where the original training images are transformed and distorted, allowing models to learn from different perspectives. In fact, data augmentation is also an effective means to mitigate the effect of overfitting (Dvornik et al., 2021), i.e., when the model learns well on the training data but performs poorly on the testing data. In this respect, we believe that the preprocessing steps contribute to the robustness of GPTSniffer's detection engine. While future work is needed to perform a feature importance analysis, the obtained results suggest that, in general, it is useful to apply an aggressive preprocessing phase to make the learning model be better generalizable and to improve its performances.

**Finding 2.** ML-based models to recognize AI-generated code should properly preprocess the input source code to achieve adequate results and be generalizable.

[7] We ran the experiments with OpenAI Text Classifier in May 2023. As of July 2023, the service was taken down by OpenAI due to a low accuracy. Full story is available at https://openai.com/blog/new-ai-classifier-for-indicating-ai-written-text.

[8] Both universities offer many programming courses at different levels, mainly with Java and Python.

We found out that GPTSniffer works almost perfectly for paired code elements, even in the presence of an aggressive preprocessing. This suggests that the more paired data is used to train the tool, the better prediction accuracy it achieves. While such a usage scenario is not common in development activities, it may occur in an educational environment. In such a case, an instructor may have different instances of the same source code, e.g., by the book, produced by themselves, or by different students, along with solutions generated by an AI. This is feasible in practice, in which one can collect source code with assignments from open source platforms, such as GitHub, and their counterparts from ChatGPT following the process we described in Section 4.2.

**Finding 3.** Whenever possible, e.g., in educational scenarios, one can train an ML model with paired snippets, as well as with those written by various developers, to achieve almost perfect predictions.

### 6.2. Prompt engineering and LLMs

Using clever prompt engineering to avoid being caught by classifiers is essentially a topic that needs further attention in the very near future. Being a detector, one should try their best to anticipate a wide range of altering scenarios. In the meantime, being a plagiarizer, people may find clever ways to dodge such a detector. For instance, they may formulate the queries, so as to ask ChatGPT to provide snippets resembling to those written by humans; or they may distort the code in different parts, making it look more *human*. In this respect, to render GPTSniffer more effective, it is necessary to consider more scenarios, training the tool with data of different origins.

The proliferation of LLMs in recent years has the potential to enable various applications in Software Engineering as discussed in recent work (Ozkaya, 2023). Nevertheless, to the best of our knowledge, no work has been conducted to exploit LLMs as a means to generate source code. We assume that a code snippet generated by ChatGPT and then modified by other LLMs would be more difficult to detect. Thus, to make GPTSniffer be more sensitive to this kind of alteration, we need to train it with the corresponding data, and this necessitates the curation of suitable datasets from ChatGPT, and a large language model. This requires additional effort, and thus we would like to address the issue in our future work.

### 6.3. Concerns and limitations

While GPTSniffer gets a satisfying classification performance, we still see some limitations – both in the approach itself, and the evaluation conducted – that need to be tackled as explained in the following points.

1. Identifying code authorship attribution is a crucial task in software engineering, as it paves the way for various activities, including bug report assignments, or software forensics (Gong and Zhong, 2022). However, within a software project, source code can be written by many developers, rendering such a detection become more challenging (Gong and Zhong, 2021). We anticipate that this also applies to the classification of code written by humans and ChatGPT, i.e., when snippets are written by different developers, it is more difficult to identify their creators. In the scope of this paper, we did not manage to test the prediction performance of GPTSniffer on code collected from several developers with various coding styles.
2. Similarly, the ability of GPTSniffer in detecting code written following task-oriented assignments has not been validated. In such an evaluation, students or developers are asked to write code following specific assignments. Moreover, a user study with crossover-designed experiments is also meaningful, in which participants are split into groups: each participant has to carry out real programming tasks, one using answers given by ChatGPT and another without the availability of ChatGPT's support.
3. The human-written snippets from a book that we used in the evaluation could cause overfitting. The standard solutions might introduce some bias if they have been used to train ChatGPT before, even though we have no trace of this. Again, using self-curated data, i.e., source code written by students or developers, would make the evaluation more solid.

### 6.4. Future research directions

We anticipate that additional work should be done in different directions to further improve the detection performance of GPTSniffer, as listed below.

1. It is important to explore the extent to which the code snippets produced by ChatGPT and then modified by programmers exhibit an increased level of difficulty in terms of identification capability. For the sake of truth, both GPTZero and OpenAI Text Classifier mitigate that issue by predicting different classes (see Tables 16 and 17). To examine the effectiveness of GPTSniffer in that case, we plan to also consider unbalanced datasets, where there is more code written and altered by humans compared to that generated by AI, and perform more empirical experiments on this type of training data. Moving forward, we aim to assess GPTSniffer using data from diverse sources and expand the approach to manage additional SE artifacts such as documentation and bug reports.
2. Recently, apart from ChatGPT, there has been a proliferation of Large Language Models (LLMs). Such models can perform a wide range of tasks (Ozkaya, 2023), and among others, they are capable of generating code in different languages. In this respect, we could train GPTSniffer with data generated by different LLMs, e.g., Llama, Bing, Chinchilla, to name but a few. This will allow GPTSniffer to learn from various types of code, and equip it with the ability to detect code coming from different sources.
3. Also, we could exploit LLMs as a means to enhance the classification engine of GPTSniffer. In fact, LLMs offer prompt engineering, allowing us to tailor the query, and get the desired answers. Essentially, prompt engineering has demonstrated its effectiveness in different contexts (Taulli, 2023; Henrickson and Meroño-Peñuela, 2023), and thus it can be further employed to enable GPTSniffer to be more effective in detecting code written in different languages.
4. GPTSniffer has been implemented using CodeBERT (Feng et al., 2020), even though other alternatives–possibly more advanced code pre-trained models do exist. That being said, *(i)* GPTSniffer already reaches a very good performance, outperforming existing classifiers, including GPTZero and OpenAI Text Classifier; *(ii)* the goal of our work was not to develop the best AI-code detector possible, but rather, to show how a specific fine-tuning of a code-pretrained model works better than just leveraging out-of-the-box classifiers. In this respect, future work may further improve GPTSniffer with better engines, i.e., we anticipate that the application of other 'lightweight' (compared to LLMs) pre-trained models, such as CodeT5 can also be considered as the classification engine for GPTSniffer. In this respect, a comparison of pre-trained models and LLMs in the detection of code written by an AI can be considered as an interesting research topic.

### 6.5. Threats to validity

*Construct validity* concerns the relationship between theory and observation. In principle, the dataset labeling is correct "by construction" as we know a priori its origin. For code hosted on GitHub, ChatGPT may have seen it already. However, this does not necessarily bias our study, as ChatGPT (OpenAI, 2023) generates code rather than retrieving snippets relevant to a query, introducing its peculiar (and recognizable) elements, e.g., imports, formatting, or other code-style features. Along this line, the different preprocessing configurations $C_1$–$C_8$ simulate the different ways a generative model could make its code "looking different" than the human code it may have been trained with. Another threat could be that for $RQ_3$ we mapped the four categories provided by GPTZero and OpenAI Text Classifier to binary categories. We used the "likely/possibly" written by ChatGPT as the ChatGPT category.

Last but not least, as detailed in Section 4.2.3, we have tried to use different prompts, with and without previous history, to ask ChatGPT generating code that resembles what a novice programmer could produce. However, while we have tried different prompts, it is still possible that there could exist some smarter ways for circumventing the proposed approach behind GPTSniffer.

*Internal validity* threats concern factors, internal to our study, that can influence the results. We ran the queries directly with GPTZero and OpenAI Text Classifier using their Web interface. As this makes the process time-consuming, we ran each query once, yet it is possible that multiple runs would produce different results. Concerning the set of hyperparameters to train GPTSniffer, we tried with different combinations of batch size, warm-up steps, or weight decay, so as to rule out possible internal threats. In the experiment, we tested with data from programming exercises, and their counterpart generated by ChatGPT. We anticipate that the results obtained for an imbalanced dataset–where there are more snippets written by humans compared to the ones generated by ChatGPT–can be worse in the real setting (i.e., GPTSniffer can wrongly distinguish humans' code from that generated by ChatGPT). Moreover, ChatGPT can be less effective in generating code that calls local APIs and methods. While we attempted to mitigate such threats by considering diverse datasets, i.e., paired and unpaired snippets, we also acknowledge that GPTSniffer needs to be trained with data of diverse origins as well as imbalance, in order to make it more effective in real-world settings.

Our study could possibly suffer from overfitting, given that we employed human-generated snippets from a book. To mitigate such a threat, we complemented that set of snippets with an additional set of 137 snippets from GitHub Gist. Moreover, the preprocessing steps applied in different configurations can be considered as data augmentation (Shorten and Khoshgoftaar, 2019), allowing the model to learn from different versions of the same input data. We believe that the evaluation of GPTSniffer would benefit from more data coming from various sources. Besides the handcrafted code and the generated ChatGPT code, we may enrich the training set with a third class of code, i.e., code that is hand-written according to some task descriptions, but not from any textbook standard solutions, which might introduce some bias if they have been used to train ChatGPT before.

*Conclusion validity* threats concern the relationship between the experimentation and outcome. Rather than reporting descriptive statistics of the performances of the different classifiers, we compare them by using a suitable test, i.e., McNemar's test, and because of multiple comparisons, adjust $p$-values using the Holm's correction (Holm, 1979). As for the analysis of Levenshtein distances among the code snippets generated with different prompts, we used the Dunn's test (Dinno, 2015).

*External validity* threats concern the generalizability of our findings. The findings of this paper may be valid only for the given datasets. We diversified the data by collecting it from different sources, attempting to simulate real-world scenarios. Also, in the evaluation, we experimented with a method definition, not a whole software project. By properly training it, GPTSniffer can be adapted to work at the project level. Finally, to avoid adding a further variable to the study, we focused on Java, yet further studies with other languages are highly desirable.

## 7. Related work

This section reviews work related to identifying automatically-generated artifacts, and applications of ChatGPT and CodeBERT to software engineering problems.

### 7.1. Distinguishing human-written and automated artifacts

Cassee et al. compared three different models to detect whether a GitHub account is human or bot-based using the repositories' comments (Cassee et al., 2021). Their evaluation show that combining the textual data with account metadata information leads to better performance, although detecting mixed accounts is still challenging, i.e., the best configuration identifies bot activity in only 10% of mixed accounts. On the same line, BIMAN (Dey et al., 2020) is a hybrid approach that identifies bot accounts on GitHub by relying on three different features, i.e., the names of the account, the commit messages, and associations between commits and projects. The evaluation showed that BIMAN succeeds in recognizing bots with good accuracy.

Paltenghi and Pradel (2021) compared the neural network attention mechanism, and the human one. By focusing on code summarization tasks, the authors collected 1508 human records and extracted 250 labeled methods by 91 participants. Their experiments showed that the neural attention mechanism *(i)* struggles in recognizing the longer methods, and *(ii)* underestimates the value of strings in the code.

Morales et al. (2020) performed an empirical study investigating whether automated refactorings can be as effective as human ones. To this end, the authors involved 80 developers in classifying 20 refactoring tasks, including human-written code and refactorings generated by RePRO—a state-of-the-art tool. The study results showed that developers cannot identify automated refactoring by five different anti-patterns.

A comparison between several classifiers was conducted to identify bots from issue and pull request comments (Golzadeh et al., 2021). To this end, the input data collected from a pre-labeled dataset is encoded by combining bag of words and TF-IDF indexing. The results show that Multinomial Naive Bayesian outperforms the others in terms of precision, recall, and F1 score.

Being particularly relevant to our work are alternative approaches to detect text generated by ChatGPT, i.e., GPTZero and OpenAI Text Classifier. GPTZero (GPTZero, 2023) which is an academic tool specifically conceived to detect text generated by ChatGPT. Given a textual content between 250 and 5000 characters, the underpinning model can categorize can distinguish if the snippet belongs to ChatGPT, human, or mixed implementation. An alternative to GPTZero is OpenAI Text Classifier (Classifier, 2023), a tool developed by OpenAI. The results of the conducted evaluation show that our approach outperforms both GPTZero and OpenAI Text Classifier for the queries we considered.

### 7.2. Applications of ChatGPT in Software Engineering

Software Engineering studies have recently been placing more emphasis on ChatGPT. Recent work (Ahmad et al., 2023) has been done to test the effectiveness of ChatGPT in aiding a software architect. The study leveraged ChatGPT for analyzing, synthesizing, and evaluating a services-oriented software application's architecture. The effectiveness of ChatGPT as coding assistance has been explored by Chauvet et al. using ChatGPT to get help with developing in HTML, CSS, and JavaScript (Avila-Chauvet et al., 2023).

Sobania et al. (2023) studied the performance of ChatGPT in fixing bugs. They compared ChatGPT with state-of-the-art tools. The results indicated that ChatGPT performs bug-fixing tasks successfully in most cases (31 of 40 bugs tested).

Recently, the feasibility of employing ChatGPT for deep learning program repair with fault detection and fault localization has been examined (Cao et al., 2023). Furthermore, they investigated the impact of prompts on debugging performance and eventually proposed a template to achieve better results.

The work by Azeem Akbar and Khan (2023) investigated the possible ethical problems related to using ChatGPT in the context of Software Engineering. The authors presented a taxonomy based on the motivators, demotivators, and their potential impact on ChatGPT ethics aspects.

The effectiveness of ChatGPT as coding assistance has been explored by a recent study (Avila-Chauvet et al., 2023), which leveraged ChatGPT to get assistance with developing in HTML, CSS, and JavaScript.

To the best of our knowledge, GPTSniffer is the first attempt to employ a pre-trained model to detect if a code snippet is written by humans or generated by ChatGPT. Among others, our approach is expected to contribute to software engineering education, helping teachers ward off cheating and plagiarism.

### 7.3. Applications of code pre-trained models to software engineering tasks

The success of pre-trained models NLP has led to the development of similar models for programming language understanding and generation. Examples of such models include CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), PLBART (Ahmad et al., 2021), or CodeT5 (Wang et al., 2021) (Mastropaolo et al., 2021). As GPTSniffer is based on CodeBERT, in the following we discuss recent work based on such a model. For a more comprehensive review of deep learning in software engineering, readers can refer to a survey (Watson et al., 2022).

Wang et al. (Wang et al., 2022) improved CodeBERT models for discriminative code tasks by combining data augmentation and curriculum-learning strategy. Their results confirm that the preprocessing pipeline proposed by Wang et al. increases CodeBERT's performance in three code-related tasks, i.e., algorithm classification, code clone detection, and code search.

An empirical evaluation of several pre-trained models for code diagnostic tasks, called probes, has been conducted by Karmakar and Robbes (2022). To enable the comparison, Karmakar and Robbes reuse a labeled dataset of compilable Java projects categorized in terms of four different probing tasks, i.e., syntactic, surface, semantic, and structure. The outcome of their study shows that CodeBERT is effective in classifying code snippets using semantic information.

The CCBERT approach (Zhang et al., 2022) combines Copy mechanism with CodeBERT to support the generation of enhanced Stack Overflow questions. After a preparatory phase in which bi-modal information is encoded, CodeBERT generates the questions by using a copy attention layer to improve the results that outperform notable baselines.

AdaMO (Gu et al., 2022) is an automatic code summarization tool based on GPT-2. AdaMO uses adaptive learning strategies, i.e., continuous pre-training and intermediate fine-tuning, to increase the overall performance and Gaussian noise strategy to capture contextual information. Compared with state-of-the-art approaches, AdaMO achieves better results in terms of ROUGE, METEOR, and BLEU scores.

## 8. Conclusion and future work

Since its release, ChatGPT revealed itself as promising to support various software development tasks, and in particular to create software artifacts, including source code that meets natural language specifications. At the same time, it has emerged the need for techniques and tools that can help users distinguish between automatically generated and human-specified content.

This paper presented GPTSniffer as a practical approach to the detection of source code generated by ChatGPT. We also performed an empirical study to investigate the extent to which it is possible to perform the task, as well as the factors that can influence this ability. GPTSniffer can distinguish code written by humans from ChatGPT-generated code under different experimental settings. Moreover, it is also resilient to some prompt engineering attempts, which involved different types of prompts – executed with and without a previous ChatGPT history – in which we requested ChatGPT to generate code that "does not look AI-generated" and, possibly, resembles what a novice programmer could write. Also, GPTSniffer outperforms two tools that recognize AI-generated text, i.e., GPTZero and OpenAI Text Classifier. While experimenting GPTSniffer under various configurations, we have identified how different preprocessing, as well as the characteristics of training and test impact the GPTSniffer prediction accuracy.

The achieved performances make GPTSniffer applicable to scenarios such as plagiarism detection in academic environments, but, also, in the context of development activities, where for example a continuous integration and delivery pipeline could identify the presence of AI-generated code and recommend a suitable code reviewing activity for it.

## CRediT authorship contribution statement

**Phuong T. Nguyen:** Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Conceptualization. **Juri Di Rocco:** Writing – review & editing, Writing – original draft, Visualization, Validation, Data curation. **Claudio Di Sipio:** Writing – review & editing, Writing – original draft, Visualization, Validation. **Riccardo Rubei:** Writing – review & editing, Writing – original draft, Data curation. **Davide Di Ruscio:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Funding acquisition, Data curation. **Massimiliano Di Penta:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Data curation.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## Acknowledgments

All the numerical simulations/evaluations have been realized on the Linux HPC cluster Caliban of the High-Performance Computing Laboratory of the Department of Information Engineering, Computer Science and Mathematics (DISIM) at the University of L'Aquila. This work has been partially supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY). The work has also been partially supported by the European Union–NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 PNRR *"FRINGE: context-aware FaiRness engineerING in complex software systEms"* grant n. P2022553SL. We acknowledge the Italian "PRIN 2022" project TRex-SE: *"Trustworthy Recommenders for Software Engineers"*, grant n. 2022LKJWHC.

## References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W., 2021. Unified pre-training for program understanding and generation. In: Toutanova, K., Rumshisky, A., Zettlemoyer, L., Hakkani-Tur, D., Beltagy, I., Bethard, S., Cotterell, R., Chakraborty, T., Zhou, Y. (Eds.), Proceedings of the 2021 NAACL. Association for Computational Linguistics, pp. 2655–2668. http://dx.doi.org/10.18653/v1/2021.naacl-main.211, Online, URL https://aclanthology.org/2021.naacl-main.211.

Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M.S., Mikkonen, T., 2023. Towards human-bot collaborative software architecting with chatgpt. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. EASE '23, Association for Computing Machinery, New York, NY, USA, ISBN: 9798400700446, pp. 279–285. http://dx.doi.org/10.1145/3593434.3593468.

Avila-Chauvet, L., Mejía, D., Acosta Quiroz, C.O., 2023. Chatgpt as a support tool for online behavioral task programming. URL https://papers.ssrn.com/abstract=4329020.

Azeem Akbar, M., Khan, A., 2023. Ethical aspects of ChatGPT in software engineering research.

Benjamini, Y., Hochberg, Y., 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. J. R. Stat. Soc. Ser. B Stat. Methodol. (ISSN: 00359246) 57 (1), 289–300.

Bosu, A., Greiler, M., Bird, C., 2015. Characteristics of useful code reviews: An empirical study at microsoft. In: Proceedings of the International Conference on Mining Software Repositories. IEEE - Institute of Electrical and Electronics Engineers, URL https://www.microsoft.com/en-us/research/publication/characteristics-of-useful-code-reviews-an-empirical-study-at-microsoft/.

Bucaioni, A., Ekedahl, H., Helander, V., Nguyen, P.T., 2024. Programming with chatgpt: How far can we go? Mach. Learn. Appl. (ISSN: 2666-8270) 15, 100526. http://dx.doi.org/10.1016/j.mlwa.2024.100526, URL https://www.sciencedirect.com/science/article/pii/S2666827024000021.

Cao, J., Li, M., Wen, M., chi Cheung, S., 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. http://dx.doi.org/10.48550/arXiv.2304.08191.

Cassee, N., Kitsanelis, C., Constantinou, E., Serebrenik, A., 2021. Human, bot or both? a study on the capabilities of classification models on mixed accounts. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 654–658. http://dx.doi.org/10.1109/ICSME52107.2021.00075.

Classifier, O., 2023. OpenAI text classifier. https://platform.openai.com/ai-text-classifier.

Dalianis, H., 2018. Evaluation Metrics and Evaluation. Springer International Publishing, Cham, ISBN: 978-3-319-78503-5, pp. 45–53. http://dx.doi.org/10.1007/978-3-319-78503-5_6.

Dey, T., Mousavi, S., Ponce, E., Fry, T., Vasilescu, B., Filippova, A., Mockus, A., 2020. Detecting and characterizing bots that commit code. In: Proceedings of the 17th International Conference on Mining Software Repositories. MSR '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450375177, pp. 209–219. http://dx.doi.org/10.1145/3379597.3387478.

Dinno, A., 2015. Nonparametric pairwise multiple comparisons in independent groups using Dunn's test. Stata J. 15 (1), 292–300.

Dulaney, H., 2023. In: Liang, Y. Daniel (Ed.), Solutions to Introduction to Java Programming, 10th ed. URL https://github.com/HarryDulaney/intro-to-java-programming.

Dvornik, N., Mairal, J., Schmid, C., 2021. On the importance of visual context for data augmentation in scene understanding. IEEE Trans. Pattern Anal. Mach. Intell. 43 (6), 2014–2028. http://dx.doi.org/10.1109/TPAMI.2019.2961896.

EuroPol, 2023. The impact of large language models on law enforcement. https://bit.ly/3V30PJH.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics. EMNLP 2020, Association for Computational Linguistics, pp. 1536–1547. http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139, Online, Nov., URL https://aclanthology.org/2020.findings-emnlp.139.

GitHub, 2024. GitHub CoPilot. https://github.com/features/copilot.

Golzadeh, M., Decan, A., Constantinou, E., Mens, T., 2021. Identifying bot activity in github pull request and issue comments. In: 2021 IEEE/ACM Third International Workshop on Bots in Software Engineering. BotSE, pp. 21–25. http://dx.doi.org/10.1109/BotSE52550.2021.00012.

Gong, S., Zhong, H., 2021. Code authors hidden in file revision histories: An empirical study. In: 29th IEEE/ACM International Conference on Program Comprehension. ICPC 2021, Madrid, Spain, May (2021) 20-21, IEEE, pp. 71–82. http://dx.doi.org/10.1109/ICPC52881.2021.00016.

Gong, S., Zhong, H., 2022. A study on identifying code author from real development. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2022, Association for Computing Machinery., New York, NY, USA, ISBN: 9781450394130, pp. 1627–1631. http://dx.doi.org/10.1145/3540250.3560878.

GPTZero, 2023. GPTZero. https://gptzero.me/.

Gu, J., Salza, P., Gall, H.C., 2022. Assemble foundation models for automatic code summarization. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, (ISSN: 1534-5351) pp. 935–946. http://dx.doi.org/10.1109/SANER53432.2022.00112.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., 2021. Graphcodebert: Pre-training code representations with data flow. arXiv:2009.08366, URL http://arxiv.org/abs/2009.08366 arXiv:2009.08366 [cs].

Henrickson, L., Meroño-Peñuela, A., 2023. Prompting Meaning: A Hermeneutic Approach to Optimising Prompt Engineering with Chatgpt. AI and Society, (ISSN: 0951-5666) http://dx.doi.org/10.1007/s00146-023-01752-8, Publisher Copyright, © 2023, The Author(s).

Holm, S., 1979. A simple sequentially rejective multiple test procedure. Scand. J. Stat. 65–70.

Husain, H., Wu, H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. CoRR, abs/1909.09436, URL http://arxiv.org/abs/1909.09436.

Karmakar, A., Robbes, R., 2022. What do pre-trained code models know about code? In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering. ASE '21, IEEE Press, Melbourne, Australia, ISBN: 978-1-66540-337-5, pp. 1332–1336. http://dx.doi.org/10.1109/ASE51524.2021.9678927, URL https://dl.acm.org/doi/10.1109/ASE51524.2021.9678927.

Levenshtein, V., 1966. Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Doklady 10, 707.

Li, Z., Chen, G.Q., Chen, C., Zou, Y., Xu, S., 2022. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450392211, pp. 1906–1918. http://dx.doi.org/10.1145/3510003.3510181.

Liang, Y.D., 2003. Introduction to Java Programming. Pearson Education India, ISBN: 9780133761313.

Mastropaolo, A., Scalabrino, S., Cooper, N., Nader Palacio, D., Poshyvanyk, D., Oliveto, R., Bavota, G., 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, Madrid, ES, ISBN: 978-1-66540-296-5, pp. 336—-347. http://dx.doi.org/10.1109/ICSE43902.2021.00041, URL https://ieeexplore.ieee.org/document/9401982/.

McNemar, Q., 1947. Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika 12 (2), 153–157.

Menzies, T., Butcher, A., Cok, D.R., Marcus, A., Layman, L., Shull, F., Turhan, B., Zimmermann, T., 2013. Local versus global lessons for defect prediction and effort estimation. IEEE Trans. Softw. Eng. 39 (6), 822–834. http://dx.doi.org/10.1109/TSE.2012.83.

Morales, R., Khomh, F., Antoniol, G., 2020. RePOR: Mimicking humans on refactoring tasks. Are we there yet? Empir. Softw. Eng. 25 (4), 2960–2996. http://dx.doi.org/10.1007/s10664-020-09826-7,

Nguyen, P.T., Di Rocco, J., Di Sipio, C., Di Ruscio, D., Di Penta, M., 2021. Recommending API function calls and code snippets to support software development. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2021.3059907.

Nguyen, P.T., Di Rocco, J., Di Sipio, C., Rubei, R., Di Ruscio, D., Di Penta, M., 2023a. GPTSniffer replication package. URL https://github.com/MDEGroup/GPTSniffer.

Nguyen, P.T., Di Sipio, C., Di Rocco, J., Di Ruscio, D., Di Penta, M., 2023b. Fitting missing API puzzles with machine translation techniques. Expert Syst. Appl. 216, 119477. http://dx.doi.org/10.1016/j.eswa.2022.119477, URL https://www.sciencedirect.com/science/article/pii/S0957417422024964,

Ogura, N., Matsumoto, S., Hata, H., Kusumoto, S., 2018. Bring your own coding style. In: Oliveto, R., Penta, M.D., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering. SANER 2018, Campobasso, Italy, March (2018) 20-23, IEEE Computer Society, pp. 527–531. http://dx.doi.org/10.1109/SANER.2018.8330253.

OpenAI, 2023. ChatGPT. https://openai.com/blog/chatgpt.

OpenAI, 2024. OpenAI Codex. https://openai.com/blog/openai-codex.

Ozkaya, I., 2023. Application of large language models to software engineering tasks: Opportunities, risks, and implications. IEEE Softw. 40 (3), 4–8. http://dx.doi.org/10.1109/MS.2023.3248401.

Paltenghi, M., Pradel, M., 2021. Thinking like a developer? Comparing the attention of humans with neural models of code. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 867–879. http://dx.doi.org/10.1109/ASE51524.2021.9678712.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2021. Asleep at the keyboard? assessing the security of github copilot's code contributions. arXiv:2108.09293, URL http://arxiv.org/abs/2108.09293.

Reda, F., 2023. GitHub Copilot is not infringing your copyright. https://felixreda.eu/2021/07/github-copilot-is-not-infringing-your-copyright/.

Shorten, C., Khoshgoftaar, T.M., 2019. A survey on image data augmentation for deep learning. J. Big Data 6, 60. http://dx.doi.org/10.1186/s40537-019-0197-0.

Sobania, D., Briesch, M., Hanna, C., Petke, J., 2023. An analysis of the automatic bug fixing performance of chatgpt.

StephanieGlen, 2023. Developers warned: GitHub Copilot code may be licensed. https://www.techtarget.com/searchsoftwarequality/news/252526359/Developers-warned-GitHub-Copilot-code-may-be-licensed.

Tabnine, 2023. AI assistant for software developers. https://www.tabnine.com/.

Taulli, T., 2023. Prompt Engineering. Apress, Berkeley, CA, ISBN: 978-1-4842-9852-7, pp. 51–64. http://dx.doi.org/10.1007/978-1-4842-9852-7_4.

Wang, D., Jia, Z., Li, S., Yu, Y., Xiong, Y., Dong, W., Liao, X., 2022. Bridging pre-trained models and downstream tasks for source code understanding. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery., New York, NY, USA, ISBN: 978-1-4503-9221-1, pp. 287–298. http://dx.doi.org/10.1145/3510003.3510062, URL https://dl.acm.org/doi/10.1145/3510003.3510062.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. Codet5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., Lyu, M.R., 2023. Prompt tuning in code intelligence: An experimental evaluation. IEEE Trans. Softw. Eng. (ISSN: 1939-3520) (01), 1–17. http://dx.doi.org/10.1109/TSE.2023.3313881.

Watson, C., Cooper, N., Palacio, D.N., Moran, K., Poshyvanyk, D., 2022. A systematic literature review on the use of deep learning in software engineering research. ACM Trans. Softw. Eng. Methodol. (ISSN: 1049-331X) 31 (2), http://dx.doi.org/10.1145/3485275, URL https://doi-org.univaq.idm.oclc.org/10.1145/3485275.

Yujian, L., Bo, L., 2007. A normalized levenshtein distance metric. IEEE Trans. Pattern Anal. Mach. Intell. 29 (6), 1091–1095.

Zhang, F., Yu, X., Keung, J., Li, F., Xie, Z., Yang, Z., Ma, C., Zhang, Z., 2022. Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information. Inf. Softw. Technol. (ISSN: 0950-5849) 148, 106922. http://dx.doi.org/10.1016/j.infsof.2022.106922, URL https://www.sciencedirect.com/science/article/pii/S0950584922000763.

**Phuong T. Nguyen** is a tenure-track assistant professor at the University of L'Aquila, Italy. He obtained a Ph.D. in Computer Science from the University of Jena, Germany. Phuong has been a teaching and research assistant in Vietnam and Italy. His research interests include Recommender Systems, Machine Learning, Mining Software Repositories. Recently, he has been developing recommender systems for mining open source software repositories. Website: https://www.disim.univaq.it/ThanhPhuong

**Juri Di Rocco** is a tenure-track assistant professor at the Department of Information Engineering Computer Science and Mathematics of the University of L'Aquila. Previously, He obtained a Ph.D. degree from the University of L'Aquila in the MDEGroup research team with Alfonso Pierantonio and Davide Di Ruscio. He is interested in all aspects of software language engineering. Main research interests are related to several aspects of Model Driven Engineering (MDE) including domain specific modeling languages, model transformation, model differencing, modeling repositories and mining techniques.

**Claudio Di Sipio** is a postdoctoral researcher at the University of L'Aquila, Italy. He is working on mining techniques to analyze open source software and he is also investigating the application of low-code platforms to support the development of recommendation systems. Contact him atclaudio.disipio@univaq.it.

**Riccardo Rubei** is a postdoctoral researcher at the University of L'Aquila, Italy. He is working on mining techniques to analyze open source software with the aim of providing developers with useful real-time recommendations. Contact him at riccardo.rubei@univaq.it.

**Davide Di Ruscio** is a full professor at the Department of Information Engineering Computer Science and Mathematics of the University of L'Aquila. His main research interests are related to several aspects of Software Engineering and Model-Driven Engineering (MDE), including domain-specific modeling languages, model differencing, coupled evolution, and recommendation systems. Davide is on the editorial board of the International Journal on Software and Systems Modeling (SoSyM), of IEEE Software, of the Journal of Object Technology, and of the Business and Information Systems Engineering journal, and he is a regular reviewer of many journals including IEEE Transactions on Software Engineering, Science of Computer Programming, Software and Systems Modeling, and Journal of Systems and Software. He serves and has served in the organization and program committees of more than 100 international events, including MODELS, STAF, ICSE, FSE, EASE, and SANER. Web: https://www.disim.univaq.it/DavideDiRuscio

**Massimiliano Di Penta** is a full professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and software testing. He is an author of over 320 papers that appeared in international journals, conferences, and workshops. He has received several awards for research and service, including four ACM SIGSOFT Distinguished paper awards. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, and ICSME. Among others, he has been program co-chair of ICSE 2023, ASE 2017, and ESEC/FSE 2021. He is associate editor-in-chief of IEEE Transactions of Software Engineering, and co-editor in chief of the Journal of Software: Evolution and Processes edited by Wiley. He is editorial board member of Empirical Software Engineering Journal edited by Springer, and has served the editorial board of ACM Transactions on Software Engineering and Methodology and IEEE Transactions on Software Engineering. Home Page: Home page: https://mdipenta.github.io.