Automatic Detection of LLM-generated Code: A Case Study of Claude 3 Haiku

MUSFIQUR RAHMAN, Concordia University, Canada
SAYEDHASSAN KHATOONABADI, Concordia University, Canada
AHMAD ABDELLATIF, University of Calgary, Canada
EMAD SHIHAB, Concordia University, Canada

Using Large Language Models (LLMs) has gained popularity among software developers for generating source code. However, the use of LLM-generated code can introduce risks of adding suboptimal, defective, and vulnerable code. This makes it necessary to devise methods for the accurate detection of LLM-generated code. Toward this goal, we perform a case study of Claude 3 Haiku (or Claude 3 for brevity) on CodeSearchNet dataset. We divide our analyses into two parts: function-level and class-level. We extract 22 software metric features, such as *Code Lines* and *Cyclomatic Complexity*, for each level of granularity. We then analyze code snippets generated by Claude 3 and their human-authored counterparts using the extracted features to understand how unique the code generated by Claude 3 is. In the following step, we use the unique characteristics of Claude 3-generated code to build Machine Learning (ML) models and identify which features of the code snippets make them more detectable by ML models. Our results indicate that Claude 3 tends to generate longer functions, but shorter classes than humans, and this characteristic can be used to detect Claude 3-generated code with ML models with 82% and 66% accuracies for function-level and class-level snippets, respectively.

CCS Concepts: • **Software and its engineering** → **Source code generation**; *Automatic programming*; Software reliability; Software safety.

Additional Key Words and Phrases: Large Language Models, Claude 3 Haiku, Code Generation, Code Stylometry, Code Complexity, Empirical Software Engineering

ACM Reference Format:
Musfiqur Rahman, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. 2024. Automatic Detection of LLM-generated Code: A Case Study of Claude 3 Haiku. 1, 1 (September 2024), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the growing popularity of Large Language Models (LLMs) and the use of LLM-generated code in software engineering, it is now more critical than ever to build robust systems that can accurately detect code snippets written by LLMs. The importance of the detection task can be motivated by existing literature that reports that LLMs can provide vulnerable code [56]. Another issue of controversy regarding the use of LLM-generated code is the ownership of the code [42]. *Who wons the LLM-generated code?* – still remains an open question.

Authors' addresses: Musfiqur Rahman, Concordia University, Montréal, QC, Canada, musfiqur.rahman@mail.concordia.ca; SayedHassan Khatoonabadi, Concordia University, Montréal, QC, Canada, sayedhassan.khatoonabadi@concordia.ca; Ahmad Abdellatif, University of Calgary, Calgary, AB, Canada, ahmad.abdellatif@ucalgary.ca; Emad Shihab, Concordia University, Montréal, QC, Canada, emad.shihab@concordia.ca.

ACM XXXX-XXXX/2024/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The aforementioned issues necessitate a mechanism for the accurate detection of LLM-generated code to facilitate code review. Moreover, with many open source software (OSS) packages such as *npm* and *PyPI* packages being used in many commercial applications, the risk of unintentional inclusion of LLM-generated code also becomes a possibility. With all these potentially negative effects of the code snippets generated by LLMs, it has become a critical aspect of software engineering to enable the detection of such code snippets.

However, only a few works [40, 62] in the existing literature address the detection of LLM-generated code. Although existing work shows promising results, they only focus on code generated for standalone functions. Recent work [71] shows that in real software projects, less than 30% of authored code is associated with standalone functions and they have a more complex architecture with a significant amount of source code artifacts, like classes, coming from the object-oriented paradigm. Another recent work [28] curates a benchmark dataset to facilitate class-level code generation with LLMs. However, each of these existing works has some limitations. For example, in [40] the function-level dataset used for analysis was curated from competitive programming problems and not real-life software code. On the other hand in [62], although the dataset used for analysis and modeling was from real-life software projects, they also used only function-level code with a focus primarily on the stylometric features (such as token diversity) of code. Lastly, the only work on class-level LLM-generated code [28] used a relatively smaller dataset of 100 hand-curated classes that are not from real-life projects either. The focus of [28] was the generation of class-level code and its evaluation, and not the detection of such code.

Our objective in the work is to bridge the gap in the existing literature by devising an LLM-generated code detection method for real-life projects. In order to determine how well we can detect LLM-generated code on both function and class levels, we perform a case study of Claude 3 Haiku [8] (or just Claude 3 for brevity) involving a multi-dimensional analysis with a focus on accurate detection as well as identifying potential features that help the correct detection of such code. For this purpose, we break down our analysis into two categories: function-level analysis and class-level analysis.

We start by generating code with Claude 3 for both function-level tasks as well as class-level tasks. Then we compare the Claude 3-generated functions and classes with respect to 22 features, which includes 17 code stylometric features and 5 code complexity features, against the corresponding metrics from human-written code to identify if (and to what extent) Claude 3-generated code is different from human-written code. Next, we train several classifiers and compare their effectiveness in detecting function-level versus class-level Claude 3-generated code. Furthermore, we perform an explanatory step where we identify the major predictors for each. In summary, we aim to answer the following three research questions (RQs) in this paper:

- **RQ1: The exploratory analysis: How unique is Claude 3-generated code?** To determine the uniqueness of Claude 3-generated code we compare the generated code with corresponding human-written code with respect to 22 features. We find that the generated code has unique characteristics. 9 out of 22 features at the function level and 6 out of 22 features at the class level are significantly different between human-written code and Claude 3-generated code.
- **RQ2: The detection: How well can we detect Claude 3-generated code?** To determine how well the uniqueness of Claude 3-generated code can be leveraged to differentiate them from human-written code we train multiple Machine Learning (ML) models. We find that our best-performing model is CatBoost, which can detect the generated function-level code with an F1-score of 0.83 and an AUC-ROC score of 0.82. The detection performance is lower for class-level code with 0.69 for F1-score and 0.66 for AUC-ROC. Our finding shows that ML models are more accurate in detecting function-level code than class-level code.

RQ3: The explanatory analysis: What are the major predictors in detecting Claude 3-generated code? To determine which features contribute the most in differentiating between Claude 3-generated and human-written code, we perform SHapley Additive exPlanations (SHAP) [51] analysis for model interpretation. Our analysis shows that the most influential features for both function-level and class-level codes are the produced lines of code including comments and blank lines. This confirms that as reported in a recent work [40] regarding OpenAI's GPT-4 [9], the detectability of Claude 3-generated code is also highly influenced by code stylometric features.

Our findings imply that code snippets generated by Claude 3 have unique characteristics with respect to different software metric-related features that make them distinguishable from their corresponding human-written code. This uniqueness can be leveraged to build ML models that can accurately detect if a given code snippet is written by Claude 3.

Our Contributions. We make the following contributions in this paper:

- To the best of our knowledge this is the first study that focuses on Claude 3-generated code. We perform our analysis of the detectability of the generated code for both functions and classes. Furthermore, this is the first class-level code generated for real-life projects.
- We provide empirical evidence of the uniqueness of Claude 3-generated function-level versus class-level code. We propose an ML approach to accurately detect the generated code.
- To promote the reproducibility of our study and facilitate future research on this topic, we publicly share our scripts and dataset online at [11].

Paper Organization. The rest of this paper is organized as follows. In Section 2 we describe our data curation and feature extraction approach. Section 3, Section 4, and Section 5 explain our approaches and findings for each RQ. Section 6 describes the implications of our findings while in Section 7 and Section 8 we discuss related works and threats to the validity of our work. Finally, we conclude this paper in Section 9.

2 DATASET

To compare human-written and Claude 3-generated code we need a data source containing code already authored by human programmers so that we can generate code using Claude 3 for the same task. In this section, we explain how we choose our data source, and generate corresponding code from Claude 3 for our analysis.

2.1 Data Source

As mentioned before, we break our analyses into two levels: function-level and class-level. In the following, we explain how we prepare the dataset for these two levels of code.

2.1.1 Function-level: In this work, we choose CodeSearchNet [39] as our data source for function-level code. We chose this dataset because our goal in this paper is to study real-life software projects and this dataset was curated using real-life OSS projects from GitHub. Furthermore, it has been used by many existing works [14, 29, 31, 53, 55, 61, 66, 72]. CodeSearchNet is a collection of functions (both standalone functions and methods) extracted from real-life projects on GitHub along with their function signatures compiled as *(comment, code)* pairs. A *comment* refers to a top-level function docstring [2], and a *code* refers to the corresponding human-written function. An example of such *(comment, code)* pair is shown in Listing 1.

Listing 1. Example of a standalone function from pysubs2 [6] with its docstring.

```
def timestamp_to_ms(groups):
    """
```

```
    Convert groups from :data:`pysubs2.time.TIMESTAMP` match to milliseconds.
    Example:
    >>> timestamp_to_ms(TIMESTAMP.match("0:00:00.42").groups())
    420
    """
    h, m, s, frac = map(int, groups)
    ms = frac * 10**(3 - len(groups[-1]))
    ms += s * 1000
    ms += m * 60000
    ms += h * 3600000
    return ms
```

This dataset consists of over 150, 000 pairs of *(comment, code)* standalone Python functions. We choose a random subset of 20, 000 such pairs to limit the processing time and expense related to the code generation using Claude 3.

2.1.2 Class-level: To perform a comparative analysis between function-level and class-level Claude 3-generated code, we curate our class-level dataset by extracting standalone classes from the same OSS projects that were used in curating the CodeSearchNet dataset. A class is a standalone class when no other classes inherit from this class and this class inherits from no other class. There are two main reasons behind choosing only standalone classes. Firstly, when there is a hierarchical relationship between classes due to inheritance, an LLM (Claude 3 in this case) needs to be prompted with not only class definitions but also many other contexts associated with the class hierarchy. This makes the input prompts arbitrarily long which becomes too expensive in terms of both cost and processing time related to code generation. Secondly, the code snippets used in the function-level analysis are all standalone functions. Therefore, by choosing only standalone classes we make sure that the source as well as the basic characteristics of the data for both function-level code and class-level code remain identical. Following is an example of a standalone class with its docstring from our dataset.

Listing 2. Example of a standalone class with its docstring from python-songpal [5] project.

```
class Notification:
    """Wrapper for notifications.

    In order to listen for notifications, call `activate(callback)`
    with a coroutine to be called when a notification is received.
    """

    def __init__(self, endpoint, switch_method, payload):
        """Notification constructor.

        :param endpoint: Endpoint.
        :param switch_method: `Method` for switching this notification.
        :param payload: JSON data containing name and available versions.
        """
        self.endpoint = endpoint
        self.switch_method = switch_method
        self.versions = payload["versions"]
        self.name = payload["name"]
        self.version = max(x["version"] for x in self.versions if "version" in x)

        _LOGGER.debug("notification␣payload:␣%s", pf(payload))

    def asdict(self):
        """Return a dict containing the notification information."""
        return {"name": self.name, "version": self.version}

    async def activate(self, callback):
        """Start listening for this notification.

        Emits received notifications by calling the passed `callback`.
        """
        await self.switch_method({"enabled": [self.asdict()]}, _consumer=callback)
```

Table 1. Top 5 *Comment to Code Ratio* values and corresponding counts of classes.

Comment To Code Ratio	# Classes
0.00	20,374 (33%)
0.17	2,080 (3.3%)
0.50	1,840 (2.9%)
0.33	1,583 (2.5%)
1.00	1,458 (2.3%)

```
def __repr__(self):
    return "<Notification {}, versions={}, endpoint={}>".format(
        self.name,
        self.versions,
        self.endpoint,
    )
```

We extracted 62,565 standalone classes from all the projects belonging to the CodeSearchNet dataset. However, our qualitative analysis of the data shows that only a much smaller number of classes have the necessary instructions as part of their docstrings for generating the code with Claude 3. Therefore, we rely on *Comment to Code Ratio* of each class to determine which classes have the necessary instructions. *Comment To Code Ratio* is calculated by taking the total number of lines of comments and dividing it by the total number of lines of code [1]. Table 1 shows the top 5 *Comment to Code Ratio* values and their corresponding counts of classes. These values were extracted using `Understand by SciTools` [10]. Further qualitative analysis shows that the average *Comment To Code Ratio* in the extracted classes is 0.39. We chose all classes with above average *Comment To Code Ratio* because our qualitative analysis shows that *Comment To Code Ratio* ≥ 0.4 usually gives enough information necessary for Claude 3 to generate the class-level code. This gives us 13,199 standalone classes.

2.2 Code Generation with Claude 3:

We chose Claude 3 for our case study because, at the time of writing this paper [1], it is one of the top 3 best-performing models for Python code generation and the cheapest one among the top 3 [7]. We use function and class docstrings as part of the prompt sent to the model, and the response received from the model is the corresponding Claude 3-generated code. We format our prompt as follows:

> Assume that you're an expert Python programmer. Please generate a Python [FUNCTION|CLASS] from the given docstring. Do not explain the code.
>
> {the [FUNCTION|CLASS] docstring}

To reduce the cost of generating code with Claude 3, we added the 'Do not explain the code' instruction as part of the prompt so that the generated response does not get unnecessarily long. With the output from this step, we obtain pairs of human-written code and corresponding Claude 3-generated code for all functions and classes in our dataset.

2.3 Feature Extraction:

Exisiting works on program comprehension reveal that software metrics can be a valuable source of information for understanding the properties of a piece of software [25, 63, 73]. Building on top of this existing finding, we aim to leverage software metrics from the point of view of distinguishing

[1] May 2024

Table 2. List of the metrics extracted to characterize the differences between Claude 3-generated code and human-written code.

Feature	Feature Type
Average Lines	Code stylometry
Average Blank Lines	Code stylometry
Average Code Lines	Code stylometry
Average Comment Lines	Code stylometry
Classes	Code stylometry
Executable Units	Code stylometry
Functions	Code stylometry
Lines	Code stylometry
Blank Lines	Code stylometry
Code Lines	Code stylometry
Declarative Code Lines	Code stylometry
Executable Code Lines	Code stylometry
Comment Lines	Code stylometry
Statements	Code stylometry
Declarative Statements	Code stylometry
Executable Statements	Code stylometry
Comment to Code Ratio	Code stylometry
Max Nesting	Code complexity
Cyclomatic Complexity	Code complexity
Max Cyclomatic Complexity	Code complexity
Average Cyclomatic Complexity	Code complexity
Sum Cyclomatic Complexity	Code complexity

between human-written and Claude 3-generated code. We used `Understand by SciTools` [10] to extract software metrics from the functions and classes in our dataset. `Understand` is an industry-standard tool for software analytics with support for all popular programming languages. As shown in Table 2 we extracted a total of 22 metrics. It is to be noted that there are metrics provided by `Understand` that are not part of our analysis. For example, on a class level `Understand` provides metrics like *Base Classes*, *Derived Classes*, *Coupled Classes*, and *Couple Classes Modified* that are not relevant to our research because we only consider standalone classes for which these metrics always have the value of zero. In the rest of the paper, we use the term 'feature' instead of 'metric' to follow ML nomenclature [46].

3 RQ1: THE EXPLORATORY ANALYSIS: HOW UNIQUE IS CLAUDE 3-GENERATED CODE?

Identifying the unique characteristics of Claude 3-generated code is the first step toward building predictive models for the detection task. To achieve that goal, we set out to understand the uniqueness of the code generated by Claude 3. In this RQ, we aim to identify and quantify to what extent the generated code differs from human-written code with respect to the extracted features in the previous section. In the following, we first explain our approach and then discuss our findings for answering this RQ.

3.1 Approach

To determine the differences in the features described in Table 2 between Claude 3-generated code and human-written code, we begin our analysis by comparing, for each feature, the distributions of the generated code and human-written code. Next, we test the statistical significance and practical significance of the differences between the distributions.

We perform Mann-Whitney U test [52] to check the differences in the extracted features. We chose this method because the features are not guaranteed to follow a normal or near-normal distribution and as a nonparametric test, this method does not require the distribution of data to be normal. We set the level of significance $\alpha = 0.01$, which determines the probability of observing the obtained results due to chance.

Hypothesis tests, such as Mann-Whitney U test, tell us whether or not there is a statistically significant difference between two distributions. However, such tests do not convey any information about how big or small the difference is. For this purpose, we use Cliff's delta [21] which estimates the magnitude of the difference, also known as effect size. Cliff's delta, d, is bounded between -1 and 1. Based on the value of d, the effect size can be categorized as one of the following qualitative magnitudes [36]:

$$\text{Effect size} = \begin{cases} \text{Negligible,} & \text{if } |d| \leq 0.147 \\ \text{Small,} & \text{if } 0.147 < |d| \leq 0.33 \\ \text{Medium,} & \text{if } 0.33 < |d| \leq 0.474 \\ \text{Large,} & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

In this study, any effect size other than 'negligible' is considered to be of practical significance. If a feature is both statistically and practically different between Claude 3-generated code and human-written code then we consider that feature to be significantly different.

3.2 Findings

3.2.1 Function-level: In Table 4 the middle column presents the differences in the features of Claude 3-generated code compared to human-written code on a function level. We find that total 9 features are significantly different on a function level. 8 out of these 9 features are code stylometric features and only one is code complexity feature (*Average Cyclomatic Complexity*). The biggest effect size (medium) observed is for *Average Comment Lines*, *Blank Lines*, *Comment Lines*, and *Comment to Code Ratio*. For all of these four features, Claude 3-generated code has greater value than Human-written code. However, in terms of *Average Code Lines*, humans tend to write longer code compared to Claude 3 meaning that Claude 3 generates code in a more concise manner than their human programmer counterparts. However, due to the presenence of more comments and blank lines Clude 3-generated code is overall lenghtier than human-written code as evident from *Lines* feature. To identify the underlying reason behind this finding, we perform a qualitative analysis. We observe that Claude 3-generated code snippets, excluding comments, are relatively smaller than human-written code. An example is presented in Table 3. We can see from the example that the core functionality of the function is implemented in a broken-down way (lines 5-15) by the human programmer whereas Claude 3 implemented the similar functionality in only a few lines of code (lines 16, 19, and 22). We conjecture from this finding that on a function level, humans usually write code in small steps one at a time to facilitate program comprehension and readability. Breaking down complex tasks into multiple smaller sub-tasks can make a piece of code more readable and understandable, which makes it easier to debug. On the contrary, Claude 3 does not consider the potential cognitive burdens that may be posed by implementing a complex task in fewer lines of code. The potential effect of this style of implementation can be seen in all other features under consideration. For example, *Average Cyclomatic Complexity* shows that generated code is less complex than human-written code, which can be directly attributed to the shorter length of the generated code snippets. In other words, the lesser the number of *Average Code Lines*, the lesser the number of decision points, and hence the lesser the complexity of the code.

Table 3. An example from `ns-api` [3] showing that Claude 3-generated function tends to be loger due to more comments and blank lines.

<table>
<tr><th>Author</th><th>Code Snippet</th></tr>
<tr><td>Human</td><td>

```python
def load_datetime(value, dt_format):
    """
    Create timezone-aware datetime object
    """
    if dt_format.endswith('%z'):
        dt_format = dt_format[:-2]
        offset = value[-5:]
        value = value[:-5]
        if offset != offset.replace(':', ''):
            # strip : from HHMM if needed (isoformat() adds it between HH and MM)
            offset = '+' + offset.replace(':', '')
            value = value[:-1]
        return OffsetTime(offset).localize(datetime.strptime(value, dt_format))

    return datetime.strptime(value, dt_format)
```

</td></tr>
<tr><td>Claude 3</td><td>

```python
def create_timezone_aware_datetime(date_string, timezone_str):
    """
    Create a timezone-aware datetime object from a date string and a timezone string.

    Args:
        date_string (str): A string representing the date and time .
        timezone_str (str): A string representing the timezone.

    Returns:
        datetime: A timezone-aware datetime object.

    Raises:
        ValueError: If the `date_string` or `timezone_str` is invalid.
    """
    # Parse the date string into a naive datetime object
    naive_datetime = datetime.strptime(date_string, '%Y-%m-%d %H:%M:%S')

    # Get the timezone object from the timezone string
    timezone = pytz.timezone(timezone_str)

    # Convert the naive datetime object to a timezone-aware datetime object
    timezone_aware_datetime = timezone.localize(naive_datetime)

    return timezone_aware_datetime
```

</td></tr>
</table>

3.2.2 Class-level: The right-most column in Table 4 presents the differences in the features of Claude 3-generated code compared to human-written code on a class level. Only 6 features including 4 code stylometric features (*Average Lines*, *Average Blank Lines*, *Average Code Lines*, and *Executable Code Lines*) and 2 code complexity features (*Average Cyclomatic Complexity*, and *Max Cyclomatic Complexity*) show significant differences between the generated class-level code snippets and the corresponding human-written snippets. The degree of difference is less compared to the function-level code with a samll effect size in all 6 cases. The class-level result shows that human-written code is slighter longer than the Claude 3-generated code, however, in this case, the difference is relatively smaller compared to the generated functions. Qualitaitve analysis of the classes reveals that, similar to standalone functions, Claude 3 tend to write more conscise methods within a class than human programmers which eventually reduces the overall length of the class as shown in

Table 4. Differences in all extracted features between Claude 3-generated code and human-written code. Effect Size represents the degree of difference. ↑ means human-written code has values greater than Claude 3-generated code, and ↓ means the opposite.

Feature	Function-level Effect Size	Class-level Effect Size
Average Lines	Negligible (↓)	Small (↑)
Average Blank Lines	Small (↓)	Small (↑)
Average Code Lines	Small (↑)	Small (↑)
Average Comment Lines	Medium (↓)	Negligible (↑)
Classes	Negligible (↓)	Negligible (↓)
Executable Units	Small (↓)	Negligible (↓)
Functions	Negligible (↓)	Negligible (↓)
Lines	Small (↓)	Negligible (↑)
Blank Lines	Medium (↓)	Negligible (↑)
Code Lines	Negligible (↑)	Negligible (↑)
Declarative Code Lines	Negligible (↓)	Negligible (↑)
Executable Code Lines	Negligible (↑)	Small (↑)
Comment Lines	Medium (↓)	Negligible (↑)
Statements	Negligible (↓)	Negligible (↑)
Declarative Statements	Negligible (↓)	Negligible (↓)
Executable Statements	Negligible (↑)	Negligible (↑)
Comment to Code Ratio	Medium (↓)	Negligible (↓)
Max Nesting	Negligible (↑)	Negligible (↑)
Cyclomatic Complexity	Negligible (↓)	Negligible (↓)
Average Cyclomatic Complexity	Small (↑)	Small (↑)
Max Cyclomatic Complexity	Negligible (↑)	Small (↑)
Sum Cyclomatic Complexity	Negligible (↑)	Negligible (↑)

Table 5. As the length of the snippet reduces, so does the complexity as evident from *Average Cyclomatic Complexity* and *Max Cyclomatic Complexity* of the Claude 3-generated classes. However, unlike function-level code, human programmers author slightly higher number blank lines than the Claude 3 generated code which can again be attributed to the readability of the code and program comprehension. We conjecture that due to the fact that classes usually accomplish more complex and compound functionalities than standalone funtions, human programmers need to structure the methods defined in the class properly by adding enough linebreaks to make the code more readable.

Claude 3-generated code has unique features compared to human-written code which can be characterized based on code stylometric features like the length of the snippet. Claude 3 tends to generate longer functions but shorter classes than human programmers. However, the generated classes are more similar to their human-written counterparts than the generated functions.

Table 5. An example from `NitPycker` [4] showing that Claude 3-generated class tends to be slighter shorter than corresponding human-written class.

<table>
<tr><th>Author</th><th>Code Snippet</th></tr>
<tr><td>Human</td><td><pre>
class FrozenExcInfo:
    """
    Execution information that can be serialized

    :param exc_info: original execution information
    """
    def __init__(self, exc_info):
        builtins.quit = non_private_exit
        builtins.exit = non_private_exit
        self.infos = exc_info[:2] + (FrozenTraceback(exc_info[2]),)

    def __getitem__(self, item):
        return self.infos[item]

    def __iter__(self):
        for i in self.infos:
            yield i
</pre></td></tr>
<tr><td>Claude 3</td><td><pre>
class FrozenExcInfo:
    """
    Execution information that can be serialized

    :param exc_info: original execution information
    """
    def __init__(self, exc_info):
        self.exc_type, self.exc_value, self.traceback = exc_info

    def __getitem__(self, item):
        return (self.exc_type, self.exc_value, self.traceback)[item]

    def __iter__(self):
        return iter((self.exc_type, self.exc_value, self.traceback))
</pre></td></tr>
</table>

4 RQ2: THE DETECTION: HOW WELL CAN WE DETECT CLAUDE 3-GENERATED CODE?

Findings from RQ1 show that Claude 3-generated code has unique characteristics that are both statistically and practically significant on both function and class levels. Our aim in RQ2 is to leverage the uniqueness of the generated code to build ML models that can accurately distinguish between human-written and Claude 3-generated code.

4.1 Approach

To determine how well predictive models can differentiate between Claude 3-generated code and human-written code, we experiment with different families of classifiers. The classifiers we train are Logistic Regression (LR) [24] (a linear classifier), K-Nearest Neighbour (KNN) [23] (a distance-based classifier), Support Vector Machine (SVM) [35] (a kernel-based classifier), Random Forest (RF) [20] (a tree-based bagging classifier), and CatBoost (CB) [57] (a tree-based boosting classifier). We choose these classifiers because they have been used in existing software engineering literature and have shown high performance in software engineering tasks [16, 30, 38, 40, 44, 45, 70].

For all models, the target variable is whether the author is Claude 3 or human and the features are the software metrics extracted in RQ1. However, we realize that many features extracted are highly correlated with each other. We remove the highly correlated features because keeping correlated features can have a negative effect on the interpretation of the models [27]. We use Spearman's correlation [64] to identify the correlated features. If two features have a Spearman correlation coefficient $\rho \geq 0.8$, we keep one of the features. We also remove the features that have a 'negligible' difference (obtained from Cliff's delta) between human-authored and Claude 3-generated code because they are unlikely to add any additional information for the model to learn from. For the function-level data the features used to train the models are *Average Blank Lines*, *Average Code Lines*, *Average Comment Lines*, *Average Cyclomatic Complexity*, *Executable Units*, *Lines*, and *Comment To Code Ratio*. For the class-level data the features used are *Average Blnak Lines*, *Executable Code Lines*, *Average Cyclomatic Complexity*, and *Average Code Lines*. For each model, we perform a K-fold cross-validation [26, 33] with $K = 10$ which gives an estimate of a classifier's generalization ability [17] to reduce bias in evaluation. The performance of the classifiers is determined based on the classification metrics listed below. Our datasets do not suffer from class unbalance which makes sure that none of these metrics show biased results towards one or the other class [32]. In the following classification metrics, *TP* stands for True Positives - the number of correctly classified Claude 3-generated code, *FP* stands for False Positives - the number of human-written code incorrectly classified as Claude 3-generated code, *TN* stands for True Negatives - the number of correctly classified human-written code, and *FN* stands for False Negatives - the number of Claude 3-generated code incorrectly classified as human-written code.

- **Precision:** Also known as Positive Predictive Value, this metric determines what proportion of data points classified as positive class is correctly classified.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall:** Also known as Sensitivity or True Positive Rate, this metric determines how well a model can classify the positive class.

$$Recall = \frac{TP}{TP + FN}$$

- **Accuracy:** This metric determines the proportion of data points correctly classified by the model.

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

- **F1-score:** This is the harmonic mean of Precision and Recall.

$$\textit{F1-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- **AUC-ROC:** This metric measures the area under the Receiver Operating Characteristic (ROC) curve [19].

All the aforementioned classification metrics are bounded between 0 and 1. The closer the value is to 1, the better the performance of the model is.

4.2 Findings

Table 6 shows the performance of various ML models in detecting Claude 3-generated code.

4.2.1 Function-level: The column representing the function-level detection performance in Table 6 shows that the tree-based models, specifically the CB model, outperform other families of models across all of the metrics except *Recall*. The detection performance improvement achieved by the CB model range from 1% to 4% for *Precision*, from 2% to 7% for *Accuracy*, from 2% to 9% for *F1-Score*, and from 2% to 7% for *AUC-ROC*. In the case of *Recall*, the SVM model shows between 1% and 16% improved performance compared to other models.

Table 6. Performance of different classifiers obtained from 10-fold cross-validation for detecting Claude 3-generated code. **Bolded** values represent the best score for each metric. Asterisks represent a joint best score.

		Fucntion-level	**Class-level**
Logistic Regression	*Precision*	0.75	0.58
	Recall	0.74	0.74
	Accuracy	0.75	0.60
	F1-Score	0.74	0.65
	AUC-ROC	0.75	0.60
K-Nearest Neighbour	*Precision*	0.78	0.65
	Recall	0.72	0.41
	Accuracy	0.76	0.60
	F1-Score	0.75	0.50
	AUC-ROC	0.76	0.60
Support Vector Machine	*Precision*	0.75	0.56
	Recall	**0.88**	**0.90**
	Accuracy	0.79	0.60
	F1-Score	0.81	**0.69***
	AUC-ROC	0.79	0.60
Random Forest	*Precision*	**0.79***	0.62
	Recall	0.82	0.69
	Accuracy	0.80	0.64
	F1-Score	0.80	0.65
	AUC-ROC	0.80	0.64
CatBoost	Precision	**0.79***	**0.64**
	Recall	0.87	0.76
	Accuracy	**0.82**	**0.66**
	F1-Score	**0.83**	**0.69***
	AUC-ROC	**0.82**	**0.66**

4.2.2 Class-level: The right-most column in Table 6 shows the performance of different models for class-level detection. The detection performance achieved is identical to the function-level detection in that the CB model outperforms all other models with respect to all metrics except *Recall*. As evident from both class-level and function-level detection performance, the SVM model achieves higher *Recall* compared to all other models. However, in class-level detection *Recall*, the SVM model outperforms other models with a much bigger margin with an increased *Recall* ranging from 14% to 49%. In the case of all other metrics, the increased performances achieved by the CB model range from 2% to 8% for *Precision*, from 2% to 4% for *Accuracy*, from 4% to 19% for *F1-Score*, and from 2% to 6% for *AUC-ROC*.

> The CB model can detect Claude 3-generated code with an average improved performance of 2.5% to 5% in *Precision*, 3% to 4.5% in *Accuracy*, 5.5% to 11.5% in *F1-Score* and 4% to 4.5% in *AUC-ROC*. The SVM model, on the other hand, can detect Claude 3-generated code with an average improved performance of 8.5% to 31.5%.

5 RQ3: THE EXPLANATORY ANALYSIS: WHAT ARE THE MAJOR PREDICTORS IN DETECTING CLAUDE 3-GENERATED CODE?

As mentioned before the goal of this research is not only to study how well Claude 3-generated code can be automatically detected using ML techniques but also to explain the performance of the detection models by identifying which features have the maximum impact on the detection performance. The explainability of these models can pave the way for new research on LLM-generated code. With the goal of explainability, we aim to determine which features contribute the most towards the correct detection. In order to find the most impactful features on the model performance, we take the Shapley Additive Explanations [49] or SHAP analysis approach using the SHAP framework [51] to compute Shapely values [67]. Shapely values are a method for showing the relative impact of each feature from a model on the output of the model by comparing the relative effect of the inputs against the average. SHAP is a popular tool which has been used in existing works [40, 44].

5.1 Approach

In this RQ, we focus on the overall best-performing model which is the CB model. We generate SHAP layered violin plots for the previously trained function-level and class-level Claude 3-generated code detection models. The layered violin plot combines feature importance with feature effects. Each violin represents the distribution of Shapley values for a feature.

5.2 Findings

5.2.1 Function-level: Figure 1 shows that the most important factors in detecting the function-level generated code are code stylometric features representing *Lines* in the code snippet including *Average Code Lines*, *Average Comment Lines*, and *Average Blank Lines*. The plot shows that as the number of *Lines*, *Blank Lines*, and *Comment Lines* in the snippet increases it is more likely to be Claude 3-generated. However, if the value of *Average Code Line* in a snippet increases it is more likely to be human-authored. The only non-stylometric feature is *Average Cyclomatic Complexity* which shows that Claude 3-generated code tends to be less complex than human-written code.

5.2.2 Class-level: Figure 2 shows that similar to function-level code, class-level Claude 3-generated code can also be detected mostly based on stylometric features. Claude 3 tends to have smaller *Average Code Lines*, *Executable Code Lines*, and *Average Blank Lines* than humans. Similar to function-level code, the only non-stylometric feature contributing to differentiating between human-written and Claude 3-generated classes is *Average Cyclomatic Complexity*. The *Average Cyclomatic Complexity* of Claude 3-generated classes is lower than that of human-written classes. For all these features, as the values tend to increase the detected code snippet is more likely to be human-written.

> Our findings show that length-related features are the most dominant ones in differentiating between Claude 3-generated and human-written code. The concise nature of the generated code along with the presence of more comments and blank lines make it detectable using predictive models on a function level, whereas, relatively shorter as well as less complex classes generated by Claude 3 contribute most to the class-level detection.

6 DISCUSSION

In this paper, we study the uniqueness of Claude 3-generated code with respect to different features obtained from various software metrics. We do our analysis on two levels of granularity of source code: function-level and class-level. In this section, we discuss the implications of our findings.

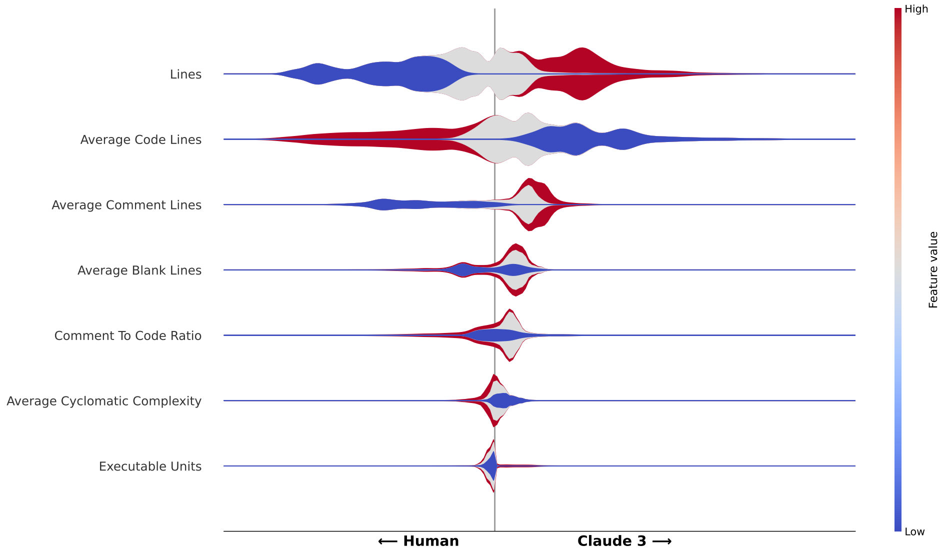

Fig. 1. SHAP feature importance for function-level detection.

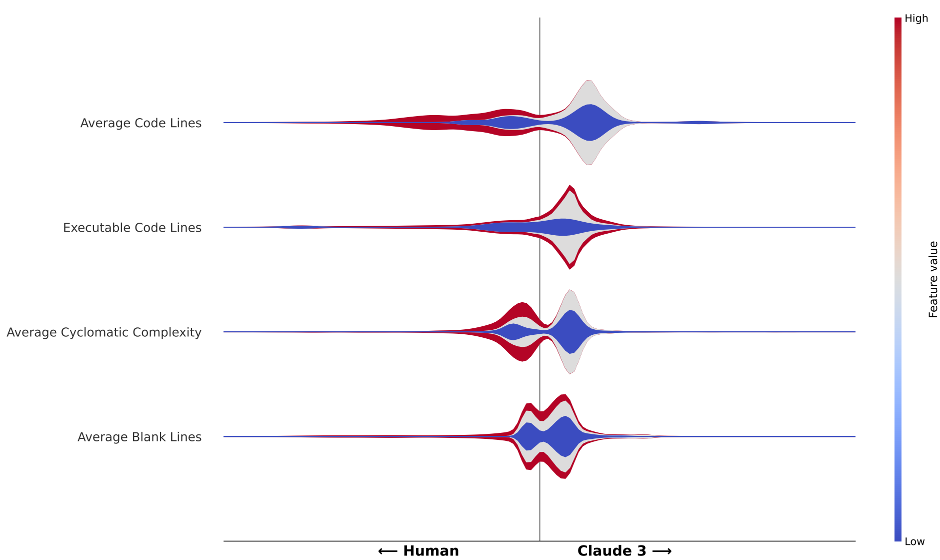

Fig. 2. SHAP feature importance for class-level detection

6.1 Implications for Practitioners

We find that Claude 3-generated functions and classes have distinct characteristics compared to corresponding human-authored code that can be leveraged to detect the generated code snippets. The detection models obtain better performance on function-level data than class-level data although the major predictors in both cases are mostly related to the length of the code. However, we realize that the detection performance can potentially vary between LLMs. To determine whether the detection performance is indeed affected by what LLM was used to generate code we run an additional experiment on function-level data where we generate code for the same functions using GPT-3.5 [2]. Table 7 shows that although we used the same functions and the corresponding docstrings along with the same prompt to generate the code, the resulting functions were different from different between Claude 3 and GPT-3.5. Accordingly, the detection performance also varies between the two LLMs under investigation. The better the generated code is, the harder it is to be detected.

Table 7. Comparision of function-level detection performance between GPT-3.5 and Claude 3 Haiku.

		Claude 3 Haiku	**GPT-3.5**
CatBoost	**Precision**	0.79	0.83
	Recall	0.87	0.84
	Accuracy	0.82	0.82
	F1-Score	0.83	0.84
	AUC-ROC	0.82	0.91

6.2 Implications for Researchers

Comparing our results with the existing work presented in [40] we find that, on a function level, it is relatively harder to detect Claude 3-generated code for real-life software tasks compared to competitive programming tasks. This finding may be attributed to two things. Firstly, real-life tasks are more complex and diverse compared to competitive programming tasks. This issue is observable in [40] as well where we see the drop in detection performance for more difficult programming tasks. Secondly, unlike competitive programming tasks where the problem is well-defined with a set of *(input, output)* pairs, real-life tasks may be more abstractly represented in docstrings. Our qualitative analysis reveals that the majority of the real-life function docstrings do not show any example *(input, output)*. Therefore, the prompts passed to an LLM from real-life projects are more abstract than the prompts passed from the programming contest problems. This may cause higher diversity in LLM-generated functions for real-life projects making it harder for the detection models to detect the generated code snippets.

7 RELATED WORK

With the advancement of LLMs and the increase in the use of LLMs in a variety of software engineering tasks, many researchers have already worked on various applications of LLMs in the domain of software engineering. Although, the work on the detection of LLM-generated code is a relatively new topic of interest in the community, the detection of LLM-generated text has been being worked on for a while. Other related topics on the intersection of LLMs and software engineering exist. In this section, we summarize some of the existing works.

Work on LLM-generated code detection: Detection of LLM-generated code is a very recent topic of interest among software engineering researchers. In the most recent work by Idialu *et al.* [40] the authors trained a graident boosting classifier to detect OpenAI's GPT-4-generated code on a function level. They used programming competition problems to generate code from GPT-4. They reported achieving high detection accuracy and, similar to our findings, they also found that the most important features in detecting GPT-4-generated code are code stylometric features. Shi *et al.* [62] proposed a perturbation-based detection technique inspired by the naturalness of code

[2]We choose GPT-3.5 because it is cheaper than GPT-4.

[37, 60]. Both works reported that the stylometric features are the features that make LLM-generated code unique. These works, however, did not study the class-level code detection. Nguyen *et al.* [54] proposed GPTSniffer where the authors reported achieving the highest detection correctness among the existing works. However, this work is different from the other two works mentioned above in that this was a CodeBERT-based approach that did not perform an explanatory analysis of the achieved performance, probably due to the black-box nature of the detection model. Puryear *et al.* [58] focused on detecting Copilot-generated code detection and compared their results with existing plagiarism detection tools like MOSS [15] and CopyLeaks [22].

LLMs in software engineering: Many other works used LLMs in different software engineering tasks. For example, Abedu *et al.* [12] studied the challenges and opportunities in using LLM-based chatbots in software repository mining. Kang *et al.* [43] reported the use of LLMs for bug reproduction and program repairs. Wnag *et al.* [66] proposed “CodeT5+”, which can support programming-related tasks such as natural language to code generation. Other LLM-supported software engineering tasks have been reported including but not limited to automated code review [48, 50], generation of comments [47], and code summarization [13].

Work on LLM-generated text detection: Much effort has been put into detecting LLM-generated content, especially text lately. Beresneva *et al.* [18] reported in their survey study that the initial computer-authored text detection works mostly focused on machine translation problems and used simple statistical approaches. Later, Jawahar *et al.* [41] published another survey which was the first work on detecting text generated by more sophisticated and powerful LLMs like GPT-2 from OpenAI. Tang *et al.* [65] in their latest study categorized detection methods into black-box detection and white-box detection and highlighted that technologies like watermarking can be used for the detection tasks. Yang *et al.* [69] and Wu *et al.* [68] reported in their survey studies that the two most common detection methods are zero-shot detection and training-based detection.

Our work is different from the existing works in several ways. Firstly, none of the aforementioned works studied Claude 3, the LLM used in this study. Secondly, they did not perform a comparison between the detection of generated functions and generated classes. Thirdly, we incorporated a set of unique complexity-related features like *Cyclomatic Complexity* and its variants which were not used before. Lastly, we compared multiple ML models in detecting Claude 3-generated code which is another unique contribution.

8 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of our study.

Internal Validity:

Threats to the internal validity of our study is two-fold. Firstly, although we trained different types of ML models, this is not an exhaustive list of models. There can be other models (or even the same models with different values of hyperparameters) that can outperform the best-performing model reported in this paper. Secondly, in our analysis, we only include standalone functions and classes due to the fact that there may be a hierarchical dependency between classes and methods of different classes due to inheritance and it may not be possible to provide Claude 3 with enough context, and hence there is a higher likelihood of receiving incomplete code, or no code at all from Claude 3. Furthermore, providing enough context to CLaude 3 for tasks with a hierarchical nature will require much longer prompts, and by extension will cause higher costs. However, we acknowledge that in real life software consists of both standalone and non-standalone artifacts and including non-standalone artifacts may change the detection performance.

External Validity:

First, in our analysis, we only included OSS projects. Although existing studies suggest that the quality of OSS projects is not very different from that of commercial software [34, 59] because many OSS projects do take standard quality control measures, we cannot guarantee that all projects in our dataset did the same. We cannot claim that the addition of commercial software data will not change the performance of the detectors. Another threat to the external validity concerns the generalizability of our findings. As mentioned in the previous section, the performance of the detection model can vary due to several reasons including but not limited to the difficulty of the tasks for which the code is being generated, and the LLM used in the generation of the code. Therefore, our findings may not be generalizable for all cases. For real-life applications, we suggest that classifiers should be trained or tuned based on the data at hand because, as evident in Section 6, given the current state-of-the-art of LLMs, it is unrealistic to expect that any detection model will be LLM-agnostic and will perform equally well across the board. We leave this discussion for future work.

9 CONCLUSION

In this work, we analyzed Claude 3-generated functions and classes to identify their distinct patterns and used those patterns to automatically detect generated code snippets. We find that Claude 3-generated functions are longer compared to human-written functions, whereas the opposite is true for class-level code. Our results further show that ML models are more accurate in detecting Claude 3-generated functions than Claude 3-generated classes. Complementing existing works [40, 62] we also find that code stylometric features are the major contributors to the success of the detection tasks. The existing works focused on function-level code whereas we performed our analysis on both function and class levels. To the best of our knowledge, we curated the first class-level dataset from real-life projects that can be leveraged by other researchers. Our findings do not negate any existing works, rather it complement them by investigating the performance of detection models on real-life problems. We make our data and scripts available for the other researchers to make our work reproducible.

REFERENCES

[1] Comment to code ratio | Engineering Metrics Library — software.com. https://www.software.com/engineering-metrics/comment-to-code-ratio. [Accessed 16-07-2024].
[2] Docstring - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Docstring. [Accessed 30-01-2024].
[3] GitHub - aquatix/ns-api: Query the Dutch railways about your routes — github.com. https://github.com/aquatix/ns-api/. [Accessed 02-09-2024].
[4] GitHub - BenjaminSchubert/NitPycker: parallelism for python's unittest — github.com. https://github.com/BenjaminSchubert/NitPycker/. [Accessed 02-09-2024].
[5] GitHub - rytilahti/python-songpal: Python library for interfacing with Sony's Songpal devices — github.com. https://github.com/rytilahti/python-songpal. [Accessed 16-07-2024].
[6] GitHub - tkarabela/pysubs2: A Python library for editing subtitle files — github.com. https://github.com/tkarabela/pysubs2. [Accessed 28-08-2024].
[7] LLM Leaderboard 2024 — vellum.ai. https://www.vellum.ai/llm-leaderboard. [Accessed 15-05-2024].
[8] Models overview - Anthropic — docs.anthropic.com. https://docs.anthropic.com/en/docs/models-overview. [Accessed 15-05-2024].
[9] OpenAI Platform — platform.openai.com. https://platform.openai.com/docs/models/gpt-3-5. [Accessed 30-01-2024].
[10] Understand: The Software Developer's Multi-Tool — scitools.com. https://scitools.com/. [Accessed 15-05-2024].
[11] Rplication package: scripts and data. https://anonymous.4open.science/r/replication_package_llm_generated_code_detection-13D3, 2024.
[12] S. Abedu, A. Abdellatif, and E. Shihab. Llm-Based Chatbots for Mining Software Repositories: Challenges and Opportunities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*. ACM, 2024.

[13] T. Ahmed and P. Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022.
[14] T. Ahmed and P. Devanbu. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455, 2022.
[15] A. Aiken. Plagiarism Detection — theory.stanford.edu. https://theory.stanford.edu/~aiken/moss/. [Accessed 29-08-2024].
[16] M. Akour, M. Alenezi, and H. Alsghaier. Software refactoring prediction using svm and optimization algorithms. *Processes*, 10(8):1611, 2022.
[17] D. Anguita, L. Ghelardoni, A. Ghio, L. Oneto, S. Ridella, et al. The'k'in k-fold cross validation. In *ESANN*, pages 441–446, 2012.
[18] D. Beresneva. Computer-generated text detection using machine learning: A systematic review. In *Natural Language Processing and Information Systems: 21st International Conference on Applications of Natural Language to Information Systems, NLDB 2016, Salford, UK, June 22-24, 2016, Proceedings 21*, pages 421–426. Springer, 2016.
[19] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.
[20] L. Breiman. Random forests. *Machine learning*, 45:5–32, 2001.
[21] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.
[22] Copyleaks. AI-Based Plagiarism & AI Content Detection | Copyleaks — copyleaks.com. https://copyleaks.com/. [Accessed 28-08-2024].
[23] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.
[24] D. R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 20(2):215–232, 1958.
[25] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on software engineering*, SE-5(2):96–104, 1979.
[26] T. G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10(7):1895–1923, 1998.
[27] C. F. Dormann, J. Elith, S. Bacher, C. Buchmann, G. Carl, G. Carré, J. R. G. Marquéz, B. Gruber, B. Lafourcade, P. J. Leitão, et al. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46, 2013.
[28] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.
[29] Z. Gong, Y. Guo, P. Zhou, C. Gao, Y. Wang, and Z. Xu. Multicoder: Multi-programming-lingual pre-training for low-resource code completion. *arXiv preprint arXiv:2212.09666*, 2022.
[30] J. Goyal and R. Ranjan Sinha. Software defect-based prediction using logistic regression: Review and challenges. In *Second International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2021*, pages 233–248. Springer, 2022.
[31] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.
[32] X. Guo, Y. Yin, C. Dong, G. Yang, and G. Zhou. On the class imbalance problem. In *2008 Fourth international conference on natural computation*, volume 4, pages 192–201. IEEE, 2008.
[33] I. Guyon, A. Saffari, G. Dror, and G. Cawley. Model selection: beyond the bayesian/frequentist divide. *Journal of Machine Learning Research*, 11(1), 2010.
[34] T. J. Halloran and W. L. Scherlis. High quality and open source software practices. In *2nd Workshop on Open Source Software Engineering*, 2002.
[35] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.
[36] M. R. Hess and J. D. Kromrey. Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, volume 1. Citeseer, 2004.
[37] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.
[38] L. Hribar and D. Duka. Software component quality prediction using knn and fuzzy logic. In *The 33rd International Convention MIPRO*, pages 402–408. IEEE, 2010.
[39] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[40] O. J. Idialu, N. S. Mathews, R. Maipradit, J. M. Atlee, and M. Nagappan. Whodunit: Classifying code as human authored or gpt-4 generated–a case study on codechef problems. *arXiv preprint arXiv:2403.04013*, 2024.
[41] G. Jawahar, M. Abdul-Mageed, and L. V. Lakshmanan. Automatic detection of machine generated text: A critical survey. *arXiv preprint arXiv:2011.01314*, 2020.
[42] Z. Ü. Kahveci. Attribution problem of generative ai: a view from us copyright law. *Journal of Intellectual Property Law and Practice*, 18(11):796–807, 2023.
[43] S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.
[44] S. Khatoonabadi, A. Abdellatif, D. E. Costa, and E. Shihab. Predicting the first response latency of maintainers and contributors in pull requests. *IEEE Transactions on Software Engineering*, 2024.
[45] S. Khatoonabadi, D. E. Costa, R. Abdalkareem, and E. Shihab. On wasted contributions: Understanding the dynamics of contributor-abandoned pull requests—a mixed-methods study of 10 large open-source projects. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–39, 2023.
[46] P. Langley et al. Selection of relevant features in machine learning. In *Proceedings of the AAAI Fall symposium on relevance*, volume 184, pages 245–271. California, 1994.
[47] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo. Auger: Automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1009–1021, 2022.
[48] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.
[49] S. Lipovetsky and M. Conklin. Analysis of regression in game theory approach. *Applied stochastic models in business and industry*, 17(4):319–330, 2001.
[50] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405*, 2020.
[51] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.
[52] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.
[53] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.
[54] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta. Is this snippet written by chatgpt? an empirical study with a codebert-based classifier. *arXiv preprint arXiv:2307.09381*, 2023.
[55] J. Pašek, J. Sido, M. Konopík, and O. Pražák. Mqdd: Pre-training of multimodal question duplicity detection for software engineering domain. *arXiv preprint arXiv:2203.14093*, 2022.
[56] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.
[57] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31, 2018.
[58] B. Puryear and G. Sprint. Github copilot in the classroom: learning to code with ai assistance. *Journal of Computing Sciences in Colleges*, 38(1):37–47, 2022.
[59] S. Raghunathan, A. Prasad, B. K. Mishra, and H. Chang. Open source versus closed source: software quality in monopoly and competitive markets. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 35(6):903–918, 2005.
[60] M. Rahman, D. Palani, and P. C. Rigby. Natural software revisited. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 37–48. IEEE, 2019.
[61] I. Saberi, F. Fard, and F. Chen. Utilization of pre-trained language model for adapter-based knowledge transfer in software engineering. *arXiv preprint arXiv:2307.08540*, 2023.
[62] Y. Shi, H. Zhang, C. Wan, and X. Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. *arXiv preprint arXiv:2401.06461*, 2024.
[63] H. M. Sneed. Understanding software through numbers: A metric based approach to program comprehension. *Journal of Software Maintenance: Research and Practice*, 7(6):405–419, 1995.
[64] C. Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 100(3/4):441–471, 1987.
[65] R. Tang, Y.-N. Chuang, and X. Hu. The science of detecting llm-generated text. *Communications of the ACM*, 67(4):50–59, 2024.

[66] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.
[67] E. Winter. The shapley value. *Handbook of game theory with economic applications*, 3:2025–2054, 2002.
[68] J. Wu, S. Yang, R. Zhan, Y. Yuan, D. F. Wong, and L. S. Chao. A survey on llm-gernerated text detection: Necessity, methods, and future directions. *arXiv preprint arXiv:2310.14724*, 2023.
[69] X. Yang, L. Pan, X. Zhao, H. Chen, L. Petzold, W. Y. Wang, and W. Cheng. A survey on detection of llms-generated content. *arXiv preprint arXiv:2310.15654*, 2023.
[70] Y. Yang, X. Xia, D. Lo, T. Bi, J. Grundy, and X. Yang. Predictive models in software engineering: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–72, 2022.
[71] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.
[72] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*, 2023.
[73] H. Zuse. Criteria for program comprehension derived from software complexity metrics. In *[1993] IEEE Second Workshop on Program Comprehension*, pages 8–16. IEEE, 1993.