Machine learning approaches to code similarity measurement: A systematic review

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.xxxxxx

Machine Learning Approaches to Code Similarity Measurement: A Systematic Review

ZIXIAN ZHANG[1], TAKFARINAS SABER[2]
[1]CRT-AI, School of Computer Science, University of Galway, Ireland, (Z.Zhang15@universityofgalway.ie)
[2]Lero, School of Computer Science, University of Galway, Ireland, (takfarinas.saber@universityofgalway.ie)

Corresponding author: Zixian Zhang.

ABSTRACT Source code similarity measurement, which involves assessing the degree of difference between code segments, plays a crucial role in various aspects of the software development cycle. These include but are not limited to code quality assurance, code review processes, code plagiarism detection, security, and vulnerability analysis. Despite the increasing application of ML technique in this domain, a comprehensive synthesis of existing methodologies remains lacking. This paper presents a systematic review of Machine Learning techniques applied to code similarity measurement, aiming to illuminate current methodologies and contribute valuable insights to the research community. Following a rigorous systematic review protocol, we identified and analyzed 84 primary studies on a broad spectrum of dimensions covering application type, devised Machine Learning algorithms, used code representations, datasets, and performance metrics, as well as performance evaluations. A deep investigation reveals that 15 applications for code similarity measurement have utilized 51 different machine learning algorithms. Additionally, the most prevalent code representation is found to be the abstract syntax tree (AST). Furthermore, the most frequently employed dataset across various code similarity research applications is BigCloneBench. Through this comprehensive analysis, the paper not only synthesizes existing research but also identifies prevailing limitations and challenges, shedding light on potential avenues for future work.

INDEX TERMS Code similarity, Code clone, Machine learning, Deep learning, Systematic literature review

I. INTRODUCTION

As a cornerstone for many code-centric tasks within the software engineering field, source code similarity measurement has captivated the attention of many researchers. Several tasks encompassing a wide array of applications including, but not limited to, code clone detection [1]–[4], plagiarism detection [5], [6], code recommendation [7], [8], code prediction [9], [10], and code generation [11], [12] rely on code similarity measurement at their core. This burgeoning interest is largely driven by the critical role that effective similarity detection plays in enhancing code quality, fostering innovation, and maintaining academic integrity within the programming community. Accurate and efficient code similarity analysis not only aids in identifying duplicate or near-duplicate code fragments—thereby helping in the reduction of technical debt—but also supports the enforcement of copyright laws and academic standards by detecting instances of plagiarism [13], [14]. Consequently, the exploration of this area represents a critical direction in software engineering research.

In recent years, the rapid advancement of Machine Learning (ML) techniques, notably in the realms of natural language processing [15], [16], recommender systems [17], [18], and autonomous vehicles [19], [20], has sparked significant interest across various sectors (e.g. [14], [21]–[24]). This trend has also positively impacted the field of code similarity measurement by bringing a large and increasing amount of studies that explore the application of ML methodologies to enhance the analysis and identification of similar code fragments.

Despite the considerable interest in code similarity, while several studies have concluded the application of ML technique in code clone, a comprehensive systematic review across the ML techniques in general code similarity-related applications is still lacking. Zakeri-Nasrabadi et al. [25] provided a comprehensive literature review on the applications of code similarity measurement and clone detection. However, their review did not specifically delve into the use of ML techniques within this area. In contrast, Kaur and Rattan [14] focused on the application of ML in the code clone domain. Nonetheless, their exploration did not extend to encompass the broader scope of general code similarity measurement. This gap highlights the need for comprehensive studies that aggregate and examine the current state of research in this area.

In this paper, we undertake a Systematic Literature Review (SLR) of ML utilization for code similarity measurement, focusing on literature published up until the end of 2023. Our review methodology adheres to the guidelines for SLRs in software engineering as proposed by Keele [26]. Initially, we established a review protocol and formulated the research questions to be addressed through this review. Subsequently, we embarked on a comprehensive search across five digital scholarly databases. Following this, we specified a set of inclusion and exclusion criteria to refine the collection of studies. To ensure comprehensive coverage, a snowballing process was adopted, enabling the inclusion of all relevant studies. Lastly, we designed a data extraction form to systematically gather and analyze the data from the included studies.

The contribution of our SLR can be summarized as follows:

- We have aggregated and synthesized studies that employ ML techniques for applications within code similarity measurement.
- Our analysis of the selected primary studies encompasses wide dimensions, including the specific types of applications related to code similarity measurement; the variety of proposed ML techniques, the data representations utilized for source code, the datasets employed for evaluation, the metrics used to assess performance of proposed approaches
- We analyse the performance of the various ML techniques across similar applications.
- We discuss the existing challenges and opportunities in the current area of code similarity measurement applications that leverage ML techniques.

This study is structured as follows: Section II provides the background knowledge of ML and code similarity and discusses related review works. Section III describes the research methodology of this review. Section IV analyzes and addresses the research questions pertinent to the application domain. Section V provides analysis and responses to the research questions concerning the ML domain. Section VI explores and resolves the research questions associated with the evaluation domain. Section **??** discuss the challenges and limitations in this field. Section IX concludes the study.

II. BACKGROUND AND RELATED WORK

In this section, we describe the background of our study in two folds (Code Similarity and Machine Learning) and discuss works related to our study.

A. CODE SIMILARITY AND CODE CLONE

The concept of code similarity quantifies the degree of resemblance between two or more code segments. Given a pair of code segments, denoted as $c1$ and $c2$, is associated with a similarity score s. A higher value of s indicates a greater similarity between the code segments. However, to the best of our knowledge, there is no universally accepted definition of code similarity due to its complex and multifaceted nature. Some studies identify code similarity based on syntactic features, where similarity is determined by textual and structural resemblances without consideration of the underlying functionality [27]–[29]. This approach evaluates elements such as code formatting, variable names, the organization of code blocks, and the presence of similar comments. Conversely, semantic similarity focuses on the meaning and functionality of code snippets, rather than their textual appearance, comparing the actions performed by the code irrespective of its written form [30], [31]. While syntactic and semantic similarities are the primary categories most commonly discussed in studies, other definitions of code similarity also exist, though they often overlap with the syntactic or semantic categories. For instance, functional similarity assesses whether different pieces of code produce the same output or perform the same task, regardless of their implementation details [32]. Structural similarity examines the organization and relationships between code elements, such as loops, conditional statements, and method calls [33], serving as a nexus between syntactic and semantic perspectives.

It is important to recognize the different between code similarity measurement and identifying code similarity or clone. Code similarity measurement serves as a general concept, whereas clone detection is a specific application of this measurement. Various applications leverage code similarity beyond detecting or identifying code clone such as code plagiarism detection [34], [35], vulnerability detection [36], [37], code recommendations [38] et al. Among the various applications, code clone detection remains the most prominent and widely studied. The definition and classification of code clones are intrinsically linked to the notion of code similarity. For instance, the most widely accepted and utilized classification of clone types is rooted in the definitions of syntactic and semantic similarities between code segments [1], [39]–[44]. Clone types 1 through 3 are categorized based on varying degrees of syntactic similarity, whereas type 4 clones are defined by semantic similarity. The specific descriptions are as follows:

- **Type-1 (T1):** These are exact copies of each other except for variations in whitespace, layout, and comments.
- **Type-2 (T2):** These involve syntactic similarity where the clones differ only in terms of identifiers, literals, types, layout, and comments.
- **Type-3 (T3):** These clones show further divergence with slightly modified code structures and statements but still maintain a noticeable similarity in overall syntax and functionality.
- **Type-4 (T4):** Represent the most abstract form of code cloning, where the code performs the same functionality but may be completely different in syntax and structure.

Moreover, an additional classification of clone types based on syntactical similarity has been proposed by Svajlenko et al. [45]. In this schema, clones are categorized based on the degree of similarity they exhibit: clones are classified as Very-Strongly Type-3 (VST3) when they exhibit a similarity ranging from 90% (inclusive) to 100%. Clones falling within the 70-90% similarity bracket are categorized as Strongly Type-3 (ST3), those within 50-70% as Moderately Type-3 (MT3), and clones with a similarity percentage between 0-50% are designated as Weakly Type-3 or Type-4 (WT3/4).

In some scenarios, the application of code similarity measurement aims to determine the similarity between code snippets, which can be quantified in various ways depending on the nature of the comparison. Examples of such similarity measures include cosine similarity [46], graph edit distance [47], or similar definition with code clone [48], [49].

B. MACHINE LEARNING

In recent years, ML techniques have been used in various fields including healthcare [21], finance development [22], Robotics [50], also in code similarity area [14], [23] since it was first proposed in 1959 [24]. Machine learning techniques can be categorized into several types, such as supervised learning, unsupervised learning, semi-supervised learning, transfer learning, and self-supervised learning, etc. The fundamental distinctions among these categories hinge on the characteristics of the input data and the nature of the output required [51]. Supervised learning is the most prevalent type of machine learning, the algorithms learn from a labeled dataset, meaning that each input data is paired with labeled outcomes [52]. This method is frequently employed in applications such as regression models [53]—designed for predicting a continuous variable—and classification models [54]—used for determining categorical outcomes. In contrast, unsupervised learning algorithms operate on unlabeled data to autonomously discover patterns and underlying structures. Typical applications of unsupervised learning include clustering [55], which involves grouping similar data points, and dimensionality reduction [56], aimed at simplifying the dataset by minimizing the number of variables involved. This hybrid approach is particularly advantageous when acquiring a fully labeled dataset is costly or impractical. Self-supervised learning is a type of unsupervised learning where the data itself provides the supervision [57]. This approach eliminates the need for human-labeled datasets, enabling models to exploit large volumes of unlabeled data. An entire different approach is transfer learning which involves taking a pre-trained model and fine-tuning it in a different but related task [58]. This is particularly useful in scenarios where one might not have a large enough dataset to train a model from scratch.

C. RELATED RESEARCH

There exists a substantial body of literature that partially addresses the code similarity measurement domain, with many published reviews either focusing on various applications of code similarity—such as detection, analysis, and management [14], [23], [59]–[65]—or offering a broader perspective on the topic [25]. The earliest relevant survey by Roy and Cordy [65] categorizes clone types, evaluates detection techniques (e.g., text-, token-, tree-, graph-based, and hybrid), and discusses the implications of code cloning for software engineering. Along similar lines, Rattan et al. [64] present a literature review emphasizing the ubiquity of code cloning and its associated maintenance challenges, proposing the need for future research in clone management and detection methods.

Recent work by Zakeri-Nasrabadi et al. [25] provides a systematic review of code similarity measurement and evaluation techniques based on 136 primary studies. While their review touches on a wide variety of topics—including code similarity applications, tools, datasets, and challenges—it also underlines key issues like the lack of reliable and accessible datasets, the necessity for more empirical evaluations, and the growing demand for hybrid approaches. Despite the breadth of these existing studies, most of the reviews primarily center on code clone detection and management or plagiarism detection.

Several reviews have also focused on how ML [14] or DL [23], [59] techniques have been integrated into the code clone detection subdomain. Kaur and Rattan [14] offer a holistic view of ML approaches for code clone detection, while Lei et al. [59] and Quradaa et al. [23] investigate the deployment of DL models—particularly RNNs—for identifying syntactic and semantic clones. In the area of source code plagiarism, Karnalim et al. [62] and Novak et al. [63] provide reviews of plagiarism detection tools, discussing techniques, obfuscation strategies, and evaluation benchmarks. A number of empirical comparisons and evaluations of code similarity tools are also reported [66]–[70], but these studies are often constrained to specific datasets or focus on limited subproblems.

Despite this variety of research, there remains a gap concerning a dedicated and comprehensive SLR that examines how ML techniques are specifically applied to the broader problem of code similarity measurement—beyond just code cloning or plagiarism detection. Addressing this gap, our SLR aims to systematically explore studies that employ ML methods for code similarity measurement, offering a comprehensive analysis of identified studies. Furthermore, we provide an in-depth analysis across multiple dimensions, including application types, ML techniques used, datasets employed, evaluation metrics, performance results, and avenues for future research.

III. METHODOLOGY

In this section, we detail the research questions, as well as the search methodology of our SLR.

A. RESEARCH QUESTIONS

The application of ML in the realm of code similarity presents a multi-dimensional challenge such as public dataset scarcity and application scarcity. The primary objective of this study is to investigate the application of ML techniques across diverse facets of code similarity.

To effectively guide and structure this investigation, we have outlined the Research Questions (RQs) and systematically categorized the content of the selected primary studies across three distinct domains: application domain, ML domain, and evaluation domain. The specifics of these domains, along with the associated research questions and the underlying motivations for each, are detailed in Table 1.

B. SEARCH STRATEGY

Our review methodology follows the structured guidelines for conducting software engineering reviews as proposed in [26]. These guidelines delineate a systematic approach for planning, executing, and reporting on the examination of existing published articles to answer specific research questions. Furthermore, our methodological framework draws inspiration from recent systematic review works in related domains [14], [23].

Figure 1 provides an overview of our comprehensive search process, which includes the creation of a search string, the selection of digital libraries, the elimination of duplicates across different libraries, and the application of inclusion and exclusion criteria for article filtering. Additionally, a snowballing process is employed to ensure a thorough coverage of relevant literature. Each step of the process is quantitatively represented in Figure 1 by the number of articles (between brackets) resulting from each phase.

In this SLR, we devised a search strategy aimed at aggregating all related articles corresponding to the predefined research questions. This strategy encompasses two primary components: search sources and search strings. The search strings, formulated as a series of keywords, are designed to retrieve a maximal number of relevant papers from preselected digital repositories.

1) Searched Sources

To conduct a comprehensive and systematic investigation into the domain of our research area, we have selected a suit of authoritative digital libraries and databases. These repositories are recognized for their extensive collections of scholarly works in computer science and related interdisciplinary fields. The following electronic databases were included in our search strategy:

- IEEE Xplore (https://ieeexplore.ieee.org)
- ACM Digitial Library (https://dl.acm.org))
- Elsevier ScienceDirect (https://www.sciencedirect.com)
- Web of Science (https://www.webofscience.com)
- Google Scholar (https://scholar.google.com)

While IEEE Xplore, ACM Digital Library and Elsevier ScienceDirect are libraries that only index publications from their respective publishers, Web of Science is renowned for its collection of peer-reviewed publications from top software engineering journals and conferences and indexes several esteemed sources. Furthermore, Google Scholar also indexes a variety of peer-reviewed publications (e.g. WILEY Online Library).

2) Search String

The construction of the search string was essential to ensuring the relevance and specificity of the retrieved literature. Firstly, based on the topic of this SLR, we divided the search string into two main groups: one related to code similarity and the other to machine learning techniques, combining them using the logical operator ‘AND‘. Secondly, within each group, we identified key topics and keywords directly related to our research questions. To ensure comprehensive coverage, we explored a broad range of synonymous and alternative terms, linking them using the logical operator ‘OR‘ to effectively incorporate variations in terminology. The final search string, designed to capture diverse expressions of code and software similarity as well as machine learning applications, is presented below.

SLR Search String

("code similarity" OR "software similarity" OR "application similarity" OR "program similarity" OR "code clone" OR "cross-language code clone" OR "software clone" OR "duplicate code" OR "code duplication" OR "code plagiarism detection" OR "software plagiarism detection" OR "code matching" OR "code reuse" OR "similarity detection")
AND
("machine learning" OR "deep learning" OR "supervised learning" OR "unsupervised learning" OR "regression" OR "classification" OR "classifier" OR "clustering" OR "transfer learning" OR "self-supervised learning" OR "semi-supervised learning" OR "transformer" OR "GPT" OR "LLM")

This search string was employed across IEEE Xplore, ACM Digital Library, and Web of Science. Given the ACM Digital Library’s vast resulting corpus, the search was restricted to the title and the abstract to manage the volume of results effectively. For IEEE Xplore and Web of Science, the string was expanded to full-text searches.

Note that due to the search constraints within ScienceDirect–specifically, the limitation of a maximum of eight keywords per field–we have split the search string for this library into shorter ones, in a way that retains the different keywords.

3) Inclusion and Exclusion Criteria

A systematic review necessitates a focus on influential and original articles in the field, termed primary studies. We conducted a manual investigation of titles, keywords, abstracts, publication types, and language to filter irrelevant articles and select primary studies. The following subsection details the criteria used to process this procedure.

Inclusion Criteria

- Articles must be in English.
- The research area must be within computer science or software engineering in general.
- The title or abstract must be related to the measurement or application of code similarity while utilizing ML techniques.
- The review will incorporate only research articles published by established academic publishers, thereby preprints and grey literature are not accepted.

Exclusion Criteria

- The absence of a novel approach, developed theory, or a documented and evaluated application.
- Articles focusing on non-source similarity, such as binary code similarity, are excluded from consideration.

TABLE 1. Research Questions of our Systematic Literature Review.

Domain	Research Questions	Motivation
Application	RQ1: What applications employ ML algorithms for measuring code similarity?	Identify and analyze how ML algorithms are being applied to measure code similarity across various software engineering contexts.
ML	RQ2: Which ML techniques have been employed?	Catalog and evaluate the diversity of ML algorithms utilized in primary studies concerning code similarity measurement.
	RQ3: Which source code representations are utilized in ML techniques?	Investigate various forms of source code representations in conjunction with ML algorithms for assessing code similarity and understanding the impact of different code representations on the effectiveness of similarity measurement.
Evaluation	RQ4: Which datasets or benchmarks have been employed in the research to facilitate the evaluation of results?	Survey and assess the datasets utilized in studies related to code similarity applications.
	RQ5: What evaluation metrics are used to measure the performance of ML techniques?	Identify and evaluate the metrics used to measure the performance of ML models in the context of code similarity applications.
	RQ6: Which ML technique demonstrates superior performance on each application?	Conduct a comparative analysis of ML techniques to find which methods produce superior results when applied to the same code similarity applications and the same dataset.

Our study is specifically tailored to explore similarities within source code.

- Any article that is not accessible.
- Theses, books, editorials, metadata, secondary, tertiary, empirical, and case studies are excluded, focusing solely on primary studies.

C. SELECTION PROCESS

The selection process for this SLR, as illustrated in Figure 1, commenced with an initial search based on the sources and search string. This review encompasses studies published up to 2023, sourced from the initial dataset of the chosen digital libraries. As depicted in Figure 1, a total of 1998 articles were initially identified.

We initially merged the same/duplicated papers obtained from various digital libraries. This step reduced the count to 738 unique papers. Then, we employed a filtering process based on titles, abstracts, and metadata, following the predefined inclusion criteria. This preliminary selection yielded 164 papers.

Before proceeding with exclusion selection on full text, we conducted a snowballing process to ensure the comprehensive inclusion of relevant articles, potentially overlooked in selected digital libraries. Following the snowballing guidelines proposed by Wohlin et al. [71], we engaged in both forward and backward snowballing. Backward snowballing involved extracting citations from each paper initially selected, while forward snowballing entailed collecting references from these papers. The papers identified through forward and backward snowballing were then filtered according to the same inclusion criteria, based on titles, abstracts, and metadata. The first round of snowballing led to the selection of 13 primary studies. Subsequently, utilizing these 13 newly identified papers, a second round of snowballing was conducted, resulting in the discovery of an additional 2 papers. A third round of snowballing, centered on these 2 papers, yielded no further findings. As illustrated in Figure 1, a total of 15 papers were identified through the entire snowballing process, demonstrating the thoroughness and comprehensive nature of our search strategy in encompassing the most pertinent research within our field of interest.

Combining the initial selection with the results of snowballing, we amassed a total of 179 papers. These were then subjected to a second stage of selection, applying full-text exclusion criteria, culminating in 104 primary studies deemed final selections for our research.

Based on the identified papers, we can assess the interest that the design and application of ML techniques for code similarity has had over time. Figure 2 illustrates the distribution over years of the identified studies that focus on utilizing ML techniques for source similarity.

Figure 2 underscores a notable upward trend in the application of ML techniques within the domain of source code similarity analysis. Remarkably, more than 80% of the primary studies were conducted in the last five years, indicating a rapid trend in interest and research activity related to the employment of ML for measuring source code similarity.

D. QUALITY ASSESSMENT

We followed the guidelines proposed by Keele [26] to conduct the quality assessment of selected studies. The quality assessment was performed after applying the inclusion and exclusion criteria to ensure that only high-quality studies were considered for analysis. This step was crucial in validating that the selected articles provided relevant, reliable, and rigorous information necessary to address the research questions of this study.

Table 15 outlines the quality screening questions used to evaluate each study. The responses to these questions were recorded as follows: **"Yes"** (assigned a score of 1), **"No"** (assigned a score of 0), or **"Partial"** (assigned a score of 0.5). Based on the total quality score, each study was ranked according to the following classification:

- **Excellent:** $7 \leq \text{score} \leq 8$
- **Good:** $4 \leq \text{score} < 7$
- **Poor:** $\text{score} < 3$

Only studies ranked as *Excellent* or *Good* were included in the final selection. A total of ? studies met these criteria and

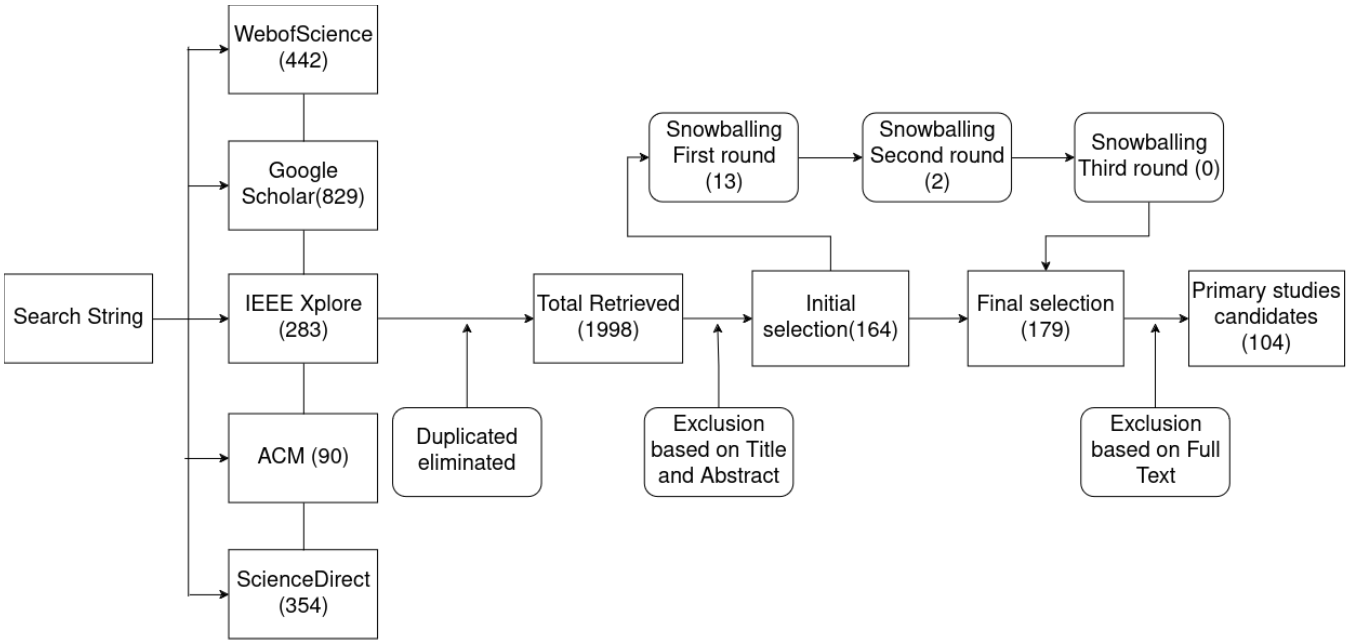

FIGURE 1. Search process in our SLR.

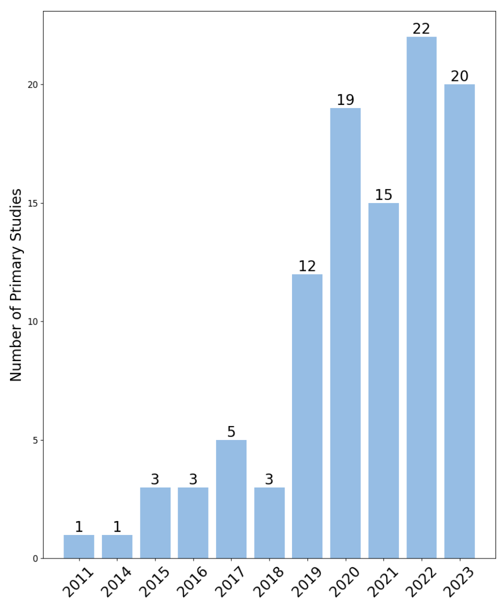

FIGURE 2. Number of primary studies in different years.

were classified as primary studies for further analysis. The detailed results of the quality assessment for each primary study are presented in Appendix B.

E. DATA EXTRACTION

Upon the completion of the primary study selection process, and in alignment with the Research Questions posited, we have constructed a table to delineate desired elements from each selected primary study. This approach is intended to augment the effectiveness of the reading process. The specifics of these elements, along with their details, are illustrated in Table 3. All extracted data can be found at https://github.com/ZixianReid/Systematic_Review.

TABLE 2. Screening Questions for Quality Assessment

Screening Questions
Q1: Does the study explicitly focus on code similarity measurement?
Q2: Does the study apply machine learning or deep learning techniques?
Q3: Does the study specify the type of source code representation used in the machine learning model?
Q4: Does the study describe the dataset used, including its source, size, and preprocessing steps?
Q5: Does the study specify the training strategies or model fine-tuning processes used?
Q6: Does the study report validation methods and evaluation metrics clearly?
Q7: Does the study compare different machine learning algorithms or baselines?
Q8: Does the study provide reproducible results, including code, dataset, or detailed methodology?

IV. APPLICATION DOMAIN (RQ1)

In this section, we aim to answer RQ1 (related to the code similarity application domains).

The analysis of our identified primary studies reveals the variety of applications where ML techniques have been leveraged for code similarity measurement. The breakdown of these applications is depicted in Figure 3. According to this figure, the most frequent application for code similarity measurement using ML techniques is code clone detection, which constitutes 77% of the applications. Other significant applications include source code similarity assessment, code plagiarism detection, and vulnerability detection employing

TABLE 3. Data Extraction Form.

Extracted Data	Description
Study Metadata	Title, Journal series, Publisher, Journal type, Publish year, Abstract.
Application Type	What is the application that is considered when measuring code similarity?
ML Type	There are mainly five types: supervised, unsupervised, semi-supervised, self-supervised, and transfer learning.
ML Technique	Which ML algorithms is employed?
Code Feature	What forms of source code representations are utilized in the ML algorithms?
Datasets	What datasets have been employed?
Programming Language	What programming languages are considered in the study?
Evaluation Metrics	Which evaluation metrics are used to measure the performance of ML based models?
Performance Results	Which ML techniques yield superior outcomes when applied to the same dataset and within the same application context?

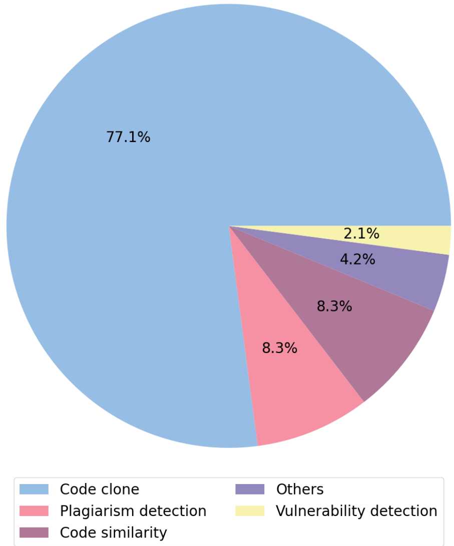

FIGURE 3. Applications of Code Similarity Measurement.

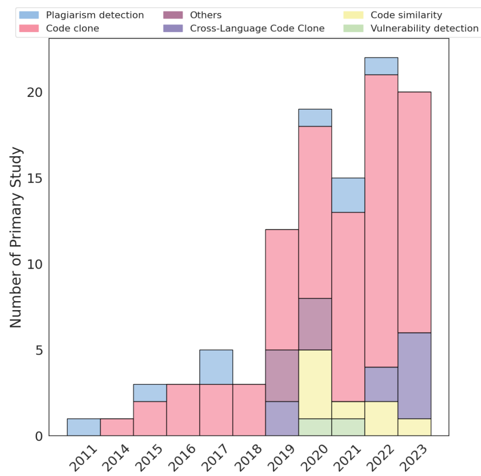

FIGURE 4. Applications of Code Similarity Measurement Over Years.

ML approaches. Additionally, there are fewer applications where ML-based source code similarity plays a crucial role, such as program repair, code prediction, and algorithm classification. It's noteworthy that while certain studies may concentrate on applying code similarity to a specific domain, they often explore general concepts of code clone detection or code similarity. In such instances, we categorize these studies according to their particular application focus. Moreover, Figure 4 presents the trajectory of ML utilization across various applications related to code similarity. Owing to the fact that some applications are only represented by one primary study, to improve the visualization, these have been aggregated as 'others'. It is observed that there has been a trend in research of cross-language code cloned detection and code similarity applications since 2019.

Table 4 presents a more comprehensive overview of these applications, with the columns of applications, their respective sub-applications, codes, counts, and citations. It is observed that the majority of studies utilize classification models to address code similarity measurement. Notably, only two cases employ a regression model, specifically in code clone validation (A3) and source code similarity (A6). These instances are separated in the citation column of this table.

In the following subsections, we will highlight the advancements and explorations in the application of ML methodologies with various code similarity applications.

A. CODE CLONE RELATED APPLICATIONS

Code clone is the most common application for code similarity measurement using ML techniques. In the following sub-sections, we will delve into sub-applications that span various facets of code cloning, where the utilization of ML has been observed, such as single-language code clone detection, cross-language code clone detection, and code clone validation.

1) Single-Language Code Clone Detection:

We observe that the majority application in code clone using machine learning algorithms is in single-language code clone detection. Here, we discuss the studies about it.

Li et al. [72] propose an ML-based method for code clone detection on web-based applications. The method uses tree-based techniques and characteristic vectors to detect code clones in large software projects. The tool has been evaluated on JDK and 7 web applications, showing better performance than Deckard [31].

Joshi et al. [74] introduce a method for identifying Type 1 and Type 2 function clones in software systems using clustering technique. The process includes three main steps:

TABLE 4. Different code similarity measurement applications where ML algorithms are applied. (*) for problems formulated as Regression problems, and Classification otherwise.

Paper	Sub-Application	Code	Count	Papers
Code Clone Detection	Single Language	A1	66	[1]–[3], [39]–[41], [43], [44], [72]–[120], [120]–[128]
	Cross-language	A2	10	[4], [129]–[136]
	Validation	A3	3	[137], [138], [139]*
	Application Clone	A4	1	[140]
Code Similarity	Software	A5	3	[141]–[143]
	Source Code	A6	5	[46], [48], [49], [144], [47]*
Plagiarism Detection	Source Code	A7	7	[5], [6], [34], [35], [145]–[147]
	Software	A8	1	[148]
Vulnerability Detection		A9	2	[36], [37]
Code Change Inspection		A10	1	[149]
Code Prediction		A11	1	[9]
Software Review		A12	1	[150]
Program Repair		A13	1	[151]
Issue-Commit Link Recovery		A14	1	[152]
Algorithm Classification		A15	1	[153]

feature extraction from functions within a software system, application of the DBSCAN [154] clustering algorithm to group similar functions, and identification of function clones from the clusters formed. The results indicate that the proposed method is effective in detecting function clones with a high precision value, suggesting the utility of DBSCAN [154] in clone detection tasks.

Sheneamer et al. [39] proposed an efficient metrics-based approach for detecting code clones by extracting features from abstract syntax trees and program dependency graphs. The method uses the XGBoost algorithm [54] with all features to improve clone detection accuracy for Type-3 and Type-4 clones. Compared with Nicad [155] and SourcererCC [156], this method improves both syntactic and semantic clone detection substantially.

Li et al. [40] introduced CCLearner: a tool that leverages deep neural networks to detect code clones. CCleaner learns from known method-level code clones and non-clones, training a classifier to detect clones based on token usage patterns. Based on the BigCloneBench [42] dataset, CCLearner demonstrates superior effectiveness in detecting various clone types with high precision and recall, outperforming traditional token-based and some tree-based approaches.

Sheneamer et al. [79] proposed a comprehensive approach to identify both semantic code clones and obfuscated code through ML. Leveraging features extracted from Java bytecode, including bytecode dependency graphs, program dependency graphs, and abstract syntax trees, the framework captures the semantic essence of code. It achieved remarkable accuracy, including a 100% success rate in certain obfuscation detection tasks.

Zhang et al. [80] introduce Go-Clone, a novel, learning-based clone detector designed specifically for the Go programming language. Go-Clone employs a deep neural network utilizing intermediate representation and labeled semantic flow graph. Evaluation of Go-Clone on a dataset constructed from 48 GitHub projects showed promising results, achieving an Area Under Curve (AUC) and Accuracy (ACC) of 89.6% and 83.80%, respectively.

Li et al. [90] introduce a novel approach for detecting semantic clones in software by leveraging a program control flow graph and Graph Attention Network (GAT) [157]. The core innovation lies in using event embedding trees to model statement execution semantics and GAT to understand the context of these executions within the code's control flow. Experimentation on the OJClone dataset [158] shows that their method surpasses existing open-source methods for Type-3 and Type-4 clone detection, providing a more nuanced approach to identifying code with similar functionalities but differing syntax.

Xu et al. [94] introduce SCCD-GAN, an advanced model for detecting semantic code clones, utilizing Graph Attention Networks [157] to analyze the similarity of code pairs with enhanced precision and lower false positive rates (FPR). By expanding on abstract syntax tree representations to include control and data flow information, and employing attention mechanisms to focus on key code features, the model aims to improve the accuracy of semantic clone detection. Implemented and evaluated on benchmark datasets like BigCloneBench [42] and Google Code Jam [159], SCCD-GAN outperforms existing methods in terms of precision and FPR, showcasing its effectiveness in identifying semantically similar yet syntactically distinct code fragments.

Keller et al. [116] explores a novel approach to generating code embeddings by leveraging visual patterns in source code. This method, called WySiWiM, uses visual representations of code fed into pre-trained image classification neural networks to benefit from transfer learning. The authors evaluate this embedding method in code clone detetion task, showing competitive performance with other SOTA methods.

Karthik et al. [99] introduce a Collaborative Code Clone Detection using a Deep Learning (CCCD-DL) model that utilizes lexical, syntactic, semantic, and structural features to identify all types of code clones efficiently. By leveraging a deep neural network (DNN) to process these features, the CCCD-DL model aims to overcome the limitations of distance-based clone detection methods, particularly in handling Large-Variance Code Clones (LV-CCs). Lexical features are extracted using LV-Mapper [160], while syntactic and semantic features are derived from Abstract Syntax Trees (AST) and Control Flow Graphs (CFG) respectively. Experimental results on benchmark datasets like BigCloneBench [42] and others such as Apache Maven and OpenNLP show that CCCD-DL outperforms existing methods in accuracy, precision, recall, processing time, and memory usage, highlighting its effectiveness in code clone detection across various clone types.

Patel et al. [102] introduce a novel approach for detecting code clones by combining holistic source code representations with Siamese Neural Networks (SNNs). The method first extracts and combines multiple intermediate representations of source code which contains Abstract Syntax Tree (AST) and Control Flow Graph (CFG) representations to generate a holistic embedding. Then, it utilizes these embeddings to train an Intermediate Merge Siamese Neural Network to detect functional code clones. This novel combination shows superior performance in detecting code clones over the OJClone dataset [158] compared to other existing methods, indicating the effectiveness of merging syntactic and semantic features with the advanced learning capabilities of SNNs.

Ding et al. [125] present BOOST, a self-supervised pre-trained model designed to improve code understanding by focusing on both structural and functional properties of source code. BOOST employs automated, structure-guided code transformation algorithms to generate functionally equivalent but textually different code, as well as textually similar but functionally distinct code. This approach helps the model learn to distinguish between code fragments with similar functionality (semantic clones) and those with different functionalities but similar syntax. The model is trained using a contrastive learning objective, which brings functionally equivalent code closer and pushes distinct code further apart. B OOST outperforms state-of-the-art models in code clone detection tasks, as demonstrated by its superior performance on datasets like POJ-104 and BigCloneBench.

Wang et al. [119] explores the performance of ChatGPT in detecting code clones. The study constructs a specific dataset covering multiple types of code data and evaluates ChatGPT's ability to detect clones in both source code and binary code. The findings indicate that while ChatGPT performs well in detecting simple code clones and explaining code semantics, it struggles with complex binary code scenarios. The research aims to guide developers in applying large intelligent models like ChatGPT to software engineering tasks

Zhang et al. [109] present a parallel DL-based model for code clone detection, leveraging temporal convolutional networks (TCNs) to address the inefficiencies of recurrent neural network (RNN) models in handling large datasets within constrained computational resources. By segmenting the abstract syntax tree (AST) of source code into statement sequences and applying TCNs for code representation, the proposed model significantly reduces time and memory consumption during clone detection.

Liu et al. [1] present TAILOR, a novel approach for detecting functionally similar code fragments through graph neural networks (GNNs), by leveraging a Code Property Graph (CPG) that encapsulates both syntactic and semantic features of source code. Utilizing a tailored GNN model, TAILOR outperforms existing methods in code clone detection and source code classification tasks, demonstrated by its performance on public datasets where it achieves up to 99.9% F-score in the clone detection task.

2) Cross-Language Code Clone Detection:

Cross-language code clone detection seeks to address the challenge of identifying functional similarities across code fragments written in disparate programming languages. This task is of significant importance for enhancing the robustness of source code within cross-platform applications. Our review reveals that the majority of primary studies within the domain of code clone detection have concentrated on singular programming languages. Nevertheless, in recent years, there has been a notable shift, with several researchers advancing proposals for DL-based methodologies aimed at augmenting the efficacy of cross-language code clone detection efforts.

Nafi et al. [129] introduce CLCDSA, a model for detecting cross-language code clones without requiring an intermediate representation of the source code. This model analyzes syntactic features and leverages API call similarity across different programming languages to identify clones. It operates in two phases: the first phase uses cosine similarity to compare features directly, while the second phase employs a Siamese Network to learn feature representations and identify clones. The authors further assemble an evaluative dataset derived from Atcoder and Google Code Jam [159]. Preliminary assessments of CLCDSA underscore its capability in identifying cross-language clones with remarkable accuracy, as evidenced by superior precision, recall, and F-measure scores, thereby outperforming pre-existing models within this sphere.

Ling et al. [131] present a novel approach to enhancing cross-language code clone detection (CCD) through the use of a Tree Autoencoder (TAE) architecture. The methodology leverages unsupervised learning to pretrain on abstract syntax trees (ASTs) from a large-scale dataset, followed by fine-tuning the trained encoder on a CCD task. The TAE incorporates a novel embedding method for AST nodes, including type and value embedding, and introduces training techniques such as "encode and decode by layers" and a node-level batch size design. The method achieved a notable improvement, with a 4% increase in F1 score for CCD, alleviating the data bottleneck issue and capturing node context information effectively.

Mehrotra et al. [4] introduce a cross-language code clone detection tool: RUBHUS, based on semi-supervised DL. This method incorporates of control and data flow with abstract syntax trees and enhances its ability to discern structural and semantic similarities beyond mere syntactic comparison. GNNs are utilized to extract code semantic information for code clone detection. This is evident from its superior performance metrics (precision, recall, F1-score) when compared to other SOTA tools.

3) Code Clone Validation:

Code clone validation constitutes a procedure aimed at ascertaining whether detected code clones are true clones that need to be considered for potential refactoring or merging. Typically, this process encompasses the analysis of intricate code structures, an endeavor that is both time-consuming and labor-intensive [42]. In response to the challenges inherent in this process, there have been scholarly efforts proposing the use of ML-based methodologies to automate code clone validation. These approaches seek to leverage the analytical capabilities of ML to streamline the validation process, thereby reducing the manual effort required and enhancing the efficiency and accuracy of clone detection initiatives.

Dumas et al. [137] present CloneCognition, an open-source ML tool designed to automate the validation of code clones detected by various tools. CloneCognition uses an Artificial Neural Network (ANN) classifier to predict whether a pair of code fragments, identified as potential clones by detection tools, are true clones based on patterns learned from manually validated clone sets. The tool achieved an accuracy of up to 87.4% in classifying clones, showcasing its potential to improve efficiency in the clone validation process significantly.

Mostaeen et al. [138] focuses on addressing the challenges posed by code clones in software maintenance through the development of a ML approach to automate the validation process of code clones. To automate this process, the authors propose a system that first builds a training dataset by manually validating clones identified by several clone detection tools across different systems. Then, it extracts several features from these clones to train a ML model. The proposed approach was evaluated using clones detected by several clone detectors in different software systems, achieving an accuracy of up to 87.4% compared to manual validation by multiple expert judges.

Sheneamer et al. [139] propose two novel schemes for labeling all types of code clones, including the challenging Type-IV (semantic) clones, specifically focusing on Java code. The first, an unsupervised approach, labels Type-IV clones and validates them with expert Java programmers. The second, a supervised scheme, classifies unknown samples based on labeled samples from the first approach. The performance of these schemes was evaluated on six well-known Java code clone corpora, demonstrating high quality in the produced code clones, as indicated by kappa agreement mean error and accuracy scores.

Tao et al. [134] present a new approach, named C4 for detecting cross-language code clones. The C4 model leverages the pre-trained CodeBERT model to convert programs into high-dimensional vector representations and fine-tunes the model using a contrastive learning objective to effectively recognize clone pairs and non-clone pairs. Experimental results demonstrate that C4 outperforms state-of-the-art baselines in terms of precision, recall, and F-measure.

Li et al. [133] introduce ZC3, a novel method for Zero-shot Cross-language Code Clone detection. To address the challenge of expensive and time-consuming data collection for code clones across different languages, the authors use contrastive snippet prediction to create an isomorphic representation space among various languages and employ domain-aware learning and cycle consistency learning to ensure the model generates aligned and distinct representations for different types of clones. Extensive experiments on four datasets demonstrate that ZC3 significantly outperforms state-of-the-art baselines in terms of MAP score.

Fang et al. [135] introduce a novel method for detecting code clones across different programming languages. This method, named TCCCD, leverages machine learning techniques to map programs written in various languages into a unified vector space using the pre-trained model UniXcoder. The model is then fine-tuned using triplet learning to enhance its effectiveness in cross-language clone detection. The authors conducted comparative experiments using the CLCDSA dataset, demonstrating that TCCCD significantly outperforms state-of-the-art baselines with precision, recall, and F1-measure scores of 0.96, 0.91, and 0.93, respectively.

B. CODE SIMILARITY

The application of code similarity measurement is aimed at evaluating segments of code to identify code similarities or differences. According to our analysis of primary studies that have been collected, there exist two categories of ML-based applications for code similarity: the first one contains the assessment of software similarity, while the second involves the comparison of source code similarity.

1) Software Similarity

Software similarity refers to the process of identifying software applications that execute similar functions or achieve similar outcomes. This identification process is instrumental in augmenting code reuse across different projects, facilitating software maintenance, and potentially uncovering instances of software plagiarism. To address the complexities and challenges inherent in accurately detecting software similarity, several scholars have proposed the adoption of sophisticated ML techniques. These advanced methodologies aim to significantly enhance the efficacy of software similarity detection, thereby contributing to the optimization of software development practices and the maintenance of intellectual property integrity.

Kawser et al. [141] present a novel model to detect similar software applications across different programming languages. It identifies semantic relationships among cross-language libraries by leveraging the doc2vec [161] model. According to experiments, this model can recommend cross-language functional similar code with average precision, recall, and F-measure scores of 0.28, and 0.85. 0.40 respectively.

Ullah et al. [142] introduce CroLSSim, a novel tool designed for detecting similar software applications across different languages. By integrating AST with methods description for semantic feature extraction, the tool leverages CNN combined with LSTM to classify software applications. The research evaluated the tool on a dataset comprising thousands of applications in Java, C#, and C++, demonstrating superior precision, recall, and F1 scores, thereby offering a robust solution for cross-language software similarity detection.

Sašo Karakatič et al. [143] introduce a tool to compare similarity between software systems through code2vec [162] neural network. It calculates the Hausdorff distance of different code embedding which AST constructs to estimate semantic software similarity. Several open-source Java systems are used for evaluation, showing the high efficiency of estimating semantic similarity by this tool.

2) Source Code Similarity

Aravind et al. [47] explore the application of GNN for estimating program similarity through CFGs. It introduces funcGNN, a novel DL based model trained to predict the Graph Edit Distance (GED) between pairs of programs by utilizing an effective embedding vector. It is noticeable that the code embeddings are completed through a combination of top-down and bottom-up graph embedding to CFGs. According to their experiment results, this model significantly outperforms traditional GED models in both accuracy and computational efficiency.

Zhang et al. [46] introduce a novel approach to determining the similarity of Scratch source codes. It proposes Siamese-based Bidirectional Long Short-Term Memory (BiLSTM) network that first abstracts Scratch blocks into a token-based code representation scheme. These tokens are then processed through a word embedding model for training. The model was evaluated using a dataset constructed from the Scratch official website, achieving over 90% accuracy and recall in similarity measurement, and 95% accuracy in code clustering tasks.

Xie et al. [144] introduce a novel approach for measuring the similarity of code snippets using a Siamese Neural Network (SNN) framework. This method aims to capture the semantic meaning of codes by mapping them into continuous space vectors, weighting by the Term Frequency-Inverse Document Frequency (TF-IDF) method. The assessment, utilizing the Open Judge system (OJS) dataset [158], illustrates the superiority of this model over methods that rely on single word embeddings.

Wu et al. [49] present a novel approach to computing code similarity using a Siamese network framework. This method involves embedding source code semantically through doc2vec [163], utilizing a Siamese network for feature extraction, and applying cosine distance for high-dimensional feature vector similarity calculations. The approach demonstrated improved precision, recall, and F1 scores over baseline methods, indicating its effectiveness in capturing code semantic information and enhancing code similarity measurement performance.

Boldini et al. [48] introduce a novel explainable method for detecting source code similarity, leveraging graph-based features and ML. The approach utilizes the LLVM Intermediate Representation (LLVM-IR) and Control-flow Graphs (CFG) to represent source code semantically. A notable aspect of this approach is the capability to reassociate extracted features with segments of the original source code, thereby enhancing explainability. Both supervised and unsupervised ML algorithms are used to analyze effectiveness and explainability for code similarity tasks.

C. PLAGIARISM DETECTION

Plagiarism detection within the realm of coding is typically defined as the process of identifying instances where code has been copied or closely mimicked proper attribution. This issue holds particular significance within academic contexts, as it impedes the ability of educators to accurately evaluate student performance and identify learners who may require additional support. The task of detecting code plagiarism presents considerable challenges, stemming from the wide variety of sources, the easy accessibility of source code, and the employment of transformation and obfuscation techniques by plagiarists. To address these challenges, several ML-based methodologies have been developed with the aim of enhancing the detection process. According to our survey, existing ML-enhanced code plagiarism detection methodologies bifurcate into two distinct categories: those that concentrate on source code and those that target software applications.

1) Source Code Plagiarism Detection

Bandara et al. [34] propose a new method for detecting source code plagiarism using ML techniques. The proposed system employs three ML algorithms—k-nearest neighbors, Naïve Bayes, and a meta-learning algorithm (AdaBoost)—to improve detection accuracy. The study found that no single algorithm could identify all instances of plagiarism satisfactorily, but a combination of the three algorithms, particularly using AdaBoost, enhanced performance. The results showed an 86.64% accuracy in classifying source code files among ten developers, demonstrating the potential of ML techniques in assisting the detection of source code plagiarism.

Yasaswi et al. [145] propose a method for plagiarism detection in programming assignments using the unsupervised clustering ML algorithm. Unlike traditional methods that rely on text-based analysis or syntactic features, this approach utilizes static features extracted from the intermediate representation of programs during compilation. By employing unsupervised learning techniques, the system clusters student submissions based on similarity, with preliminary results outperforming popular tools like MOSS [].

Yasaswi et al. [6] introduce an approach to detect plagiarism in source codes by employing deep features extracted using a character-level Recurrent Neural Network (char-RNN) pre-trained on the Linux Kernel source code. These deep features, being generic, do not require fine-tuning for each problem set, showcasing flexibility and robustness. The methodology demonstrated significant improvements over handcrafted features, achieving a 9.5% increase in F1-score for binary classification (copy/non-copy) and a 5% increase for three-way classification (copy/partial-copy/non-copy), illustrating the effectiveness of DL models in understanding the complexity of programming languages and detecting plagiarism with higher accuracy.

Fokam et al. [5] introduce an enhanced version of the Abstract Syntax Tree-based Neural Network (ASTNN) model [84] that incorporates contrastive learning for improved performance in source code plagiarism detection. The enhanced ASTNN model uses these embeddings to measure the similarity between source codes. By employing a contrastive loss function, the model aims to accurately map the input source codes into a representation space where similar codes are clustered together, thereby facilitating the detection of plagiarized codes. According to their experiment results, A significant improvement in the detection of code clones, with a +5% increase in the F1-score compared to the original ASTNN model is noticed.

2) Software Plagiarism Detection

Ullah et al. [148] propose a novel methodology to detect software plagiarism across multiple programming languages through the utilization of ML techniques. The approach includes preprocessing steps to convert source codes into a format suitable for ML analysis, followed by feature extraction using Principal Component Analysis (PCA) to retain crucial information without significant data loss. Subsequently, a multinomial logistic regression model (MLR) is applied for classification purposes. The research showcases the effectiveness of this methodology by achieving 84% accuracy in training data and 73% in testing data across five popular programming languages: C, C++, Java, C#, and Python.

D. VULNERABILITY DETECTION

Vulnerability detection encompasses the identification of weaknesses, defects, or security bugs within software programs. Such vulnerabilities may stem from a range of issues, including design flaws, substandard coding practices, or inadequate security assessments, thereby providing potential avenues for unauthorized system or network access. Thus, identifying and addressing vulnerabilities is crucial for enhancing software security. The integration of ML in vulnerability detection represents a burgeoning area of interest within the domain of software security. Nonetheless, there has been a relatively limited exploration of the combined use of code similarity metrics and ML techniques in this context. The rationale behind incorporating code similarity into vulnerability detection lies in its potential to identify predefined common vulnerabilities. By leveraging these known vulnerabilities as a benchmark, it is possible to enhance the effectiveness of detecting similar security flaws.

He et al. [36] Vul-Mirror, a few-shot learning model aimed at accurately identifying vulnerable code clones in software development. This novel approach is designed to automate the feature extraction of vulnerabilities and utilizes CNN to assess code similarity, significantly improving detection accuracy compared to existing methods. The authors also construct a dataset containing vulnerable code clones from different five operating systems, the results show that Vul-Mirror achieved an impressive accuracy rate of 95.7%, outperforming state-of-the-art methods.

Sun et al. [37] propose a novel approach for detecting software vulnerabilities based on code similarity. The methodology combines a Siamese network with BiLSTM and attention mechanisms to analyze pairs of code snippets (vulnerabilities and patches) and assess their similarity. By focusing on the similarity in the view of vulnerabilities, the model is designed to identify vulnerable code more accurately. Evaluation of datasets comprising vulnerabilities and patches from OpenSSL and Linux demonstrated that the VDSimilar model significantly outperforms existing methods, achieving about 97.17% in AUC value for OpenSSL vulnerabilities.

E. OTHERS

Based on the findings of our survey, we have identified some innovation applications that leverage ML techniques, in conjunction with code similarity measures to achieve specific tasks. We will provide a general description of them in the following section.

1) Code Change Inspection

Ayinala et al. [149] propose RIDL, a novel approach for inspecting recurring code changes using DL. The method focuses on identifying and summarizing systematic changes by learning patterns from code clones across multiple software versions. Utilizing a trained CNN model on a dataset of 13,940 clones, RIDL demonstrates high efficiency in summarizing recurring changes with 95.1% accuracy and detecting change anomalies with 93.1% accuracy in evaluations conducted on four open-source projects.

2) Code Prediction

Hammad et al. [9] develop DeepClone, a DL-based model to predict code tokens and complete method bodies by leveraging code clones. Utilizing Gated Recurrent Units (GRU) and Generative Pretrained Transformer 2 (GPT-2), and evaluated on the BigCloneBench dataset, the approach demonstrated the ability to predict code with high accuracy, notably achieving 95% accuracy in the top 10 suggestions. The research emphasizes the utility of code clones for enhancing code prediction, presenting a significant advancement in code generation and prediction applications.

3) Software Review

Guo et al. [150] addresses the challenge of insufficient code reviews in software development by proposing a novel review sharing approach that utilizes deep semi-supervised learning for code similarity assessment. Their methodology integrates an autoencoder-based model with a Convolutional Neural Network (CNN) to detect code clones, which facilitates the sharing of informative reviews from projects with abundant reviews to those with few or none. This approach is designed to improve code quality by leveraging existing reviews across similar code fragments. The experiments conducted demonstrate the effectiveness of their model in accurately detecting code clones, thereby enabling the sharing of relevant reviews.

4) Program Rrepair

White et al. [151] explore the utilization of DL for the automatic repair of software programs, facilitated by measurements of code similarity. The authors introduce DeepRepair, a method that employs code similarities derived from RNN to select and prioritize repair ingredients—code snippets that can potentially fix a defect. DeepRepair operates on the principle that similar code can contain the seeds for repairing defects and that these "seeds" can be identified and adapted through a combination of DL techniques, including the use of recursive autoencoders for learning code representations. The evaluation, conducted on six open-source Java projects comprising 374 buggy program revisions, demonstrates that DeepRepair can effectively identify repair ingredients, generating patches for defects that are not addressable by traditional, redundancy-based repair techniques.

5) Issue-commit Link Recovery

Xie et al. [152] introduce DeepLink, an innovative approach to enhancing the accuracy of issue-commit link recovery in software projects. By implementing a Recurrent Neural Network (RNN) architecture, the authors capture semantic information of code and related texts. Furthermore, they construct a code-related knowledge graph, facilitating the similarity relationships among code repositories. This methodology significantly enhances the efficacy of issue-commit link recovery processes. DeepLink significantly outperforms state-of-the-art techniques in experiments conducted on six real-world projects from the Apache Software Foundation.

6) Algorithm Classification

Bui et al. [153] introduces a novel framework for cross-language algorithm classification based on code similarity assessment. The approach utilizes Bilateral Neural Networks (Bi-NN) that encode both syntactic and semantic features of code from two different programming languages to recognize and classify algorithms implemented across these languages. The framework is evaluated on a dataset comprising thousands of Java and C++ programs, demonstrating promising classification results both within a single language and across languages. The use of dependency trees with tree-based convolutional neural networks (TBCNN) achieves the highest classification accuracy, highlighting the benefit of incorporating semantic information directly into code representations for improved algorithm classification accuracy.

Key Findings Of RQ1

- ML techniques play an important role across various areas of code similarity measurement, with a focus on detecting code clones in a single language. The adoption of DL for detecting cross-language code clones is a notable recent development.
- Beyond code clone detection, ML is also widely used for direct code similarity assessment and code plagiarism detection. However, there is significant potential for ML applications in less explored areas such as vulnerability detection, code prediction, and program repair.

V. ML DOMAIN: RQ2 AND RQ3

In this section, we aim to answer RQ2 (related to the types of ML techniques and RQ3 (related to the code representation in the ML techniques).

A. DEVISED ML TECHNIQUES (RQ2)

The integration of ML techniques into the field of software engineering has seen a remarkable surge in recent years. Our survey highlights the diverse application of ML in the realm of code similarity, with a wide range of algorithms being employed across various studies. Table 5 provides an overview of the different ML algorithms used. We categorize these ML algorithms into different types of learning methodologies. For each category, the specific ML algorithm name, the counts of its usage, and the applications in which it is employed, along with relevant papers, are provided.

In these selected primary studies, we found five different ML strategies: supervised learning, unsupervised learning, semi-supervised learning, self-supervised learning, transfer learning. Among them, supervised learning is the predominant type. We sub-category supervised learning into two categories: conventional ML algorithms and deep learning algorithms.

From table 5, we observe that a wide range of machine learning algorithms spanning both tradition approaches like SVMs, Decision Trees and etc, and DL models like GNN, RNN, CNN and etc. It is apparent that DL methods have become increasingly dominant in the field of code similarity measurement, overtaking traditional ML algorithms in terms of usage frequency. Besides, we depict the trend representing the the evolution of DL algorithm architectures over time, showing in Figure 5. Notably, CNNs and RNNs have maintained consistent usage over the years, whereas GNN-based networks have become increasingly prominent since 2018, reaching a peak in usage in 2023.

We observe several unsupervised learning algorithms, predominantly clustering-based, are employed in the assessment of code similarity. K-means clustering is the most prevalent algorithm, utilized extensively across various applications such as code clone detection, plagiarism detection, and program repair, as evidenced in studies by [72], [73], [151]. Additionally, the DBSCAN clustering algorithm is notably applied in software clone detection [74], organizing software functions into clusters based on their similarity across diverse software metrics.

TABLE 5. ML techniques used for code similarity measurement.

Learning Category	Sub-category	ML algorithm	Count	Citations
Supervised	Conventional	ANN	5	A1 [98], A3 [137], [138], A6 [48]
		Adaptive Boosting (Adaboost)	1	A1 [44]
		Bagging	2	A1 [39], [79]
		Decision Table	1	A3 [138]
		Decision Tree	3	A1 [44], [97], [98]
		Gradient Boosting Decision Tree (GDBT)	1	A1 [44]
		J48	3	A1 [39], [79], A3 [138]
		Logistic regression	3	A1 [98], A3 [138], A8 [148]
		Logit Boost	1	A1 [98]
		Naïve Bayes Classifier	5	A1 [39], [79], A3 [138], A6 [48], A7 [34]
		Random Committee	2	A1 [39], [79]
		Random Foreset	7	A1 [39], [44], [79], [96]–[98], A3 [138]
		Random Subspace	1	A1 [79]
		Random Tree	1	A3 [138]
		Rotation Forest	3	A1 [39], [79], [96]
		SVM	6	A1 [39], [75], [79], A6 [48], A7 [146],A14 [152]
		Stochastic Grad.D classifier	1	A3 [138]
		XGboost	4	A1 [39], [44], [96], A7 [146]
		k-Nearest Neighbors (kNN)	5	A1 [39], [44], [97], A6 [48], A7 [34]
	Deep Learning	Bi-directional Causal Convolutional Neural Network (BiC-CNN)	1	A1 [93]
		Bi-directional Long Short-Term Memory (BiLSTM)	2	A6 [46], A9 [37]
		CNN	9	A1 [96], [99], [102], [113], A5 [142], A6 [49], [144], A7 [147], A9 [36]
		DNN	9	A1 [40], [41], [78], [80], [88], [98]
		Graph Neural Networks (GNNs)	5	A1 [1], [101], A2 [4], [136], A6 [47]
		Gated Graph Neural Networks (GGNN)	3	A1 [91], [103], [122]
		Gated Recurrent Units (GRUs)	6	A1 [84], [101], [110], [122], A11 [9], A14 [152]
		Graph Attention Networks (GATs)	5	A1 [3], [90], [92], [107], [115]
		Graph Convolutional Network (GCN)	4	A1 [92], [111], [112], [122]
		Graph Matching Networks (GMN)	3	A1 [2], [91], [106]
		Graph-LSTM	1	A1 [104]
		LSTM	8	A1 [77], [82], [95], [107], [108], [122]
		Recurrent Neural Networks (RtNN)	8	A1 [3], [76], [84], [123], A4 [140], A5 [143], A7 [5], [6]
		Recursive Autoencoders (RAE)	2	A1 [83], [89]
		Recursive Neural Networks (RvNN)	2	A1 [76], [81]
		Temporal Convolutional Networks (TCN)	1	A1 [109]
		Transformer	2	A1 [100], [112]
		Tree-CNN	4	A1 [43], [164], A2 [132], A15 [153]
		Tree-LSTM	2	A1 [85], A2 [131]
Unsupervised	-	DBSCAN Clustering	1	A1 [74]
		Hierarchical Clustering	1	A6 [48]
		K-Means Clustering	3	A1 [72], [73], A13 [151]
		Unspecified	1	A3 [145]
Semi-supervised	-	Bidirectional Recurrent Neural Network Autoencoder	1	A1 [86]
		CNN	1	A12 [150]
		Generative Adversarial Networks (GAN)	1	A1 [94]
		Graph Attention Networks (GAT)	1	A1 [94]
		Graph Convolutional Network (GCN)	1	A1 [86]
		K-Nearest Neighbors (KNN)	1	A3 [141]
		LSTM	1	A2 [130]
		RSGNN (Residual Self-attention Graph Neural Network)	1	A1 [105]
Self-supervised	-	Tree-CNN	1	A1 [114]
		Transformer	2	A1 [125], [127]
Transfer	-	CodeBert	1	A11 [9]
		ELMo	1	A1 [124]
		GPT	2	A1 [119], A11 [9]
		GraphCodeBert	2	A1 [120], [121]
		ResNet	1	A1 [116]
		Xcode	1	A1 [124]
		UniXcode	2	A1 [128], A2 [135]

The realm of semi-supervised learning also finds its application in the domain of code similarity measurement. Sheneamer et al. [139] present a semi-supervised technique to label semantic code clones (Type-IV). Initially, an unsupervised approach uses Java ByteCode and Levenshtein similarity to tentatively label clones. Subsequently, multiple classifiers are trained are these labeled data to improve the code code detection accuracy. Perez et al. [130] employ a tree-based skip-gram model to learn token-level vector representations. These vectors are severe as input to a supervised LSTM-based neural network to predict code clones across Java and Python. Yuan et al. [86] introduce a novel semi-supervised technique for semantic code clone detection. Firstly, a bidirectional RNN autoencoder is used to model node features in CFGs. For global semantic clones, a Graph Convolutional Network (GCN) models the CFGs directly, capturing broader structural similarities. Nafi et al. [141] use a semi-supervised approach called CroLSim to detect similar software applications across different programming languages. The model uses distributed representations (Doc2Vec) to capture the semantics embed-

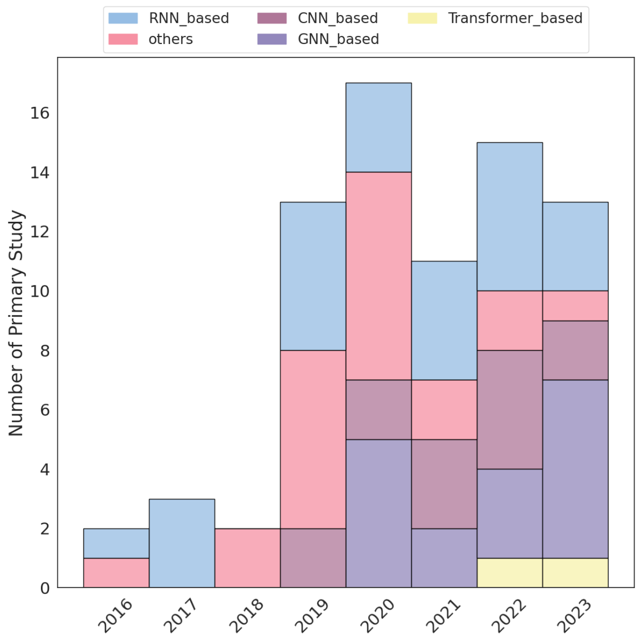

FIGURE 5. Studies using different deep learning architectures over years.

ded in software applications and KNN is used to cluster similar software applications based on their semantic similarities. Guo et al. [150] present a semi-supervised technique for code clone detection which facilitates review sharing. This model leverages a CNN and an autoencoder architecture to train on both labeled and unlabeled datasets, improving the detection of code clones. Xu et al. [94] propose a semi-supervised technique, SCCD-GAN for semantic code clone detection using GANs. This method initially uses unsupervised learning to handle large unlabelled datasets, enhancing feature extraction capabilities. It then employs GANs to refine the model's accuracy in identifying semantic clones.

We observe several self-supervised learning usage in 2021 and 2022. Bui et al. [114] leverage the ASTs of source code to predict syntax sub-tree. Each sub-tree is used as a label for the pretext task, which involves predicting the probability of that subtree appearing in a particular AST. This method allows the model to learn useful representations of code without the need for human labeling. Ding et al. [125] implements self-supervised learning through a novel model called BOOST, which focuses on the characteristics of source code. The key components include a Node-Type Masked Language Model to encode structural information, contrastive learning objective by generating positive samples and hard negative samples and structure-guided code transformation. Wang et al. [105] implement self-supervised learning by designing a novel hierarchical contrastive learning model to learn the relationships between nodes in an AST hierarchy. Specifically, the model uses the depth-first search (DFS) algorithm to traverse the AST nodes and construct pseudo-labels for the nodes automatically. The model then pre-trains by predicting the AST hierarchy through contrastive learning, which aims to minimize the distance between similar data representations and maximize the distance between dissimilar data representations. These extracted representations are applied to various downstream tasks including code clone detection.

The domain of code similarity measurement has not seen significant research efforts on transfer learning until the year 2020. However, the volume of research witnessed a decline in 2021, followed by a modest resurgence in 2022 and 2023. To the best of our knowledge, there are primarily two categories of pre-trained models utilized in this field. The first category comprises feature extraction models such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXCode (Guo et al., 2022), and Xcode (Lin et al., 2022). The second category includes text generation models, exemplified by GPT-2 (Hammad et al., 2020) and GPT-3.5 (Wang et al., 2023). Among the feature extraction models, CodeBERT is notably prevalent. This model employs a multi-layer bidirectional Transformer architecture, encompassing a total of 125 million parameters and is trained on a substantial dataset that spans six programming languages. Several studies have explored the application of CodeBERT in the realm of code clone detection. For instance, Sharma et al. (2022) harnessed the attention mechanism based on CodeBERT to detect code clones. Similarly, Abid et al. (2023) implemented CodeBERT for semantic code clone detection across Java and Python. Li et al. (2023) introduced a zero-shot technique utilizing CodeBERT for cross-language code clone detection, an approach further echoed by Tao et al. (2022) who applied a contrastive learning objective with CodeBERT for similar purposes. In terms of text generation models, use cases include the work of Muhammad et al. (2020), who fine-tuned GPT-2 with the BigCloneBench dataset to enhance code prediction performance by enabling the model to assimilate common coding practices. Additionally, Wang et al. (2023) assessed GPT-3's efficacy in code clone detection by inputting code pairs and evaluating their similarity through textual responses, thus determining their clonal relationship.

Figure 7 delineates the deployment of ML algorithms across various applications pertaining to code similarity. Supervised learning is predominant in all applications in code similarity measurement. We observe the usage of unsupervised learning usage in code clone detection (A1), source code similarity (A6), and source code plagiarism detection (A7), but remains low overall. We observe the usage of semi-supervised learning usage in code clone detection (A1), cross-language code clone detection (A2), code clone validation (A3), application clone (A5), and software review (A12), half of them located in code clone detection (A1). We observe one transfer learning usage in software review (A12) and one self-supervised learning technique in code clone detection (A1).

Key Findings Of RQ2

- Various types of ML techniques are employed in code similarity measurement, including supervised, unsupervised, semi-supervised, self-supervised, and transfer learning. However, supervised learning is the predominant type, with DL algorithms gaining considerable popularity in recent years.
- Only single instances of self-supervised and transfer learning have been identified in the realm of code similarity measurement, suggesting a substantial

research gap in this area.

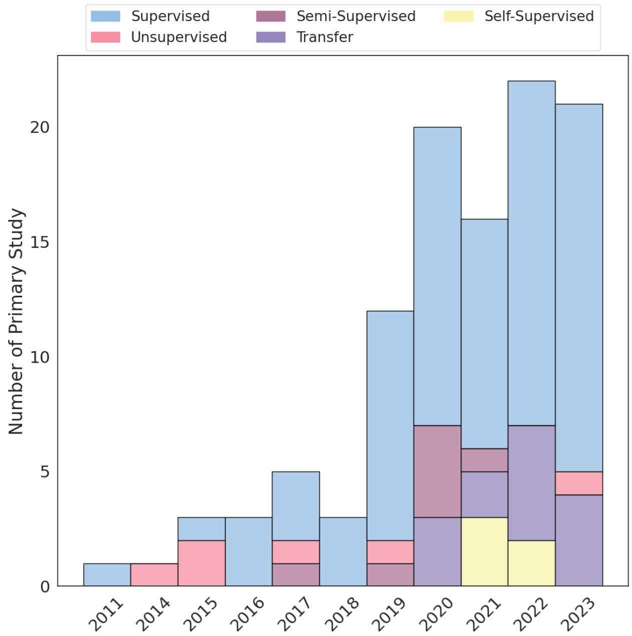

FIGURE 6. Studies using different ML categories over time.

B. SOURCE CODE REPRESENTATIONS IN ML TECHNIQUES (RQ3)

Effective and proper representation of code is crucial for leveraging ML techniques in applications related to code similarity, enabling the effective learning of the syntax and semantics of source code [105]. Diverse representations of source code are employed to exclude uninteresting elements and augment similarity across code fragments. These representations typically manifest in various forms, including token-based, tree-structured, and graph-structured formats. The selection of a specific form of code representation is strategically aligned with the requirements of different ML algorithms to optimize their applicability and effectiveness.

Table 6 presents an overview of the various code representations employed in the selected studies, detailing both their frequency of use and the associated citations. It is evident that the Abstract Syntax Tree (AST) emerges as the predominant code representation, being utilized in 64 instances. Among graph-based representations, the Control Flow Graph (CFG) is the most prevalent with 12 applications, followed by the Program Dependence Graph (PDG), which is used 8 times. The third code representations are token-based representations.

Furthermore, it has been observed that several researchers are exploring the integration of multiple code representations to achieve a more comprehensive depiction of code, thereby enhancing performance. For example, Liu et al. [1] combine the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG) to capture both syntactic and semantic dimensions of the source code, offering a holistic perspective for the analysis of program functionalities and consequently improving code clone detection performance. Similarly, Nafi et al. [129] combine AST with API, which are feature-based representations, to augment the scalability of cross-language code clone detection. Bui et al. [153] employ a combination of AST and token-based code representations to capture both structural and lexical features of code, thereby enriching the learning process.

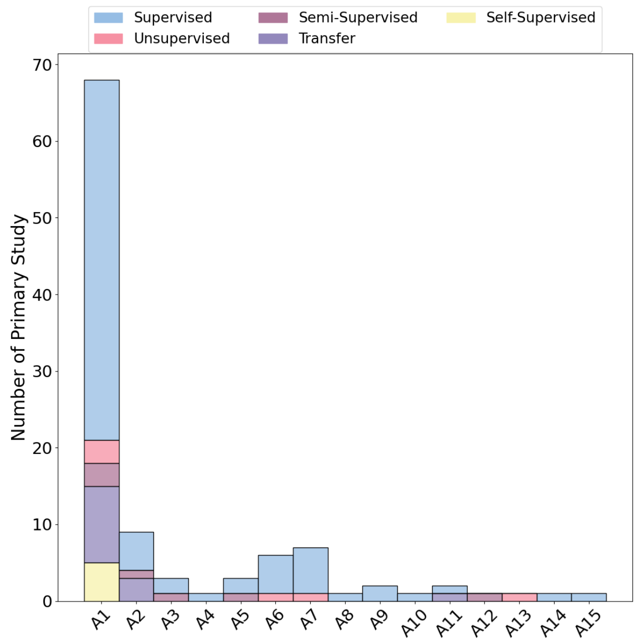

FIGURE 7. Studies using different ML categories over various code similarity applications.

Figure 8 illustrates the distribution of different code representation types across various code similarity applications. It is noted that single language code clone detection (A1) incorporates all types of code representations, with tree-based representation being the most prevalent, followed by graph-based representation. From both a quantity and application breadth perspective, tree-based representation is predominant across multiple applications (A1, A2, A3, A5, A7, A9, A10, A12, A13, A14, A15). In terms of application breadth, token-based representation ranks second, utilized in applications A1, A3, A4, A5, A6, A7, A11, and A15. From a quantity perspective, graph-based representation holds the second position, predominantly employed in the code clone detection area (A1, A3, A4).

Figure 9 illustrates the temporal trends in the usage of code representation types across selected studies. Since 2016, tree-based representations have predominated. However, an increase in the utilization of graph-based representations is observable from 2018 onward, culminating in a usage proportion in 2023 that is comparable to that of tree-based representations. This trend underscores a growing interest in the application of graph-based representations within code similarity studies. Further supporting this trend, Figure 5 illustrates a parallel increase in the adoption of GNN-based networks, mirroring the rise of graph-based code representations.

Key Findings Of RQ3

- Abstract Syntax Tree (AST) emerges as the most commonly used technique in code similarity assessments. However, there is an increasing use of graph-based representations like the Program Dependence Graph (PDG), Control Flow Graph (CFG), and Data Flow Graph (DFG).
- Many studies are exploring the integration of multiple code representations to capture different dimensions of code, including semantic, structural, and lexical aspects.

TABLE 6. Code representation used for code similarity measurement.

Code Representation Type	Code Representation	Count	Papers
Tree-based Representation	Abstract Syntax Tree (AST)	64	[1], [2], [4], [5], [36], [39], [40], [43], [44], [72], [76], [77], [79], [81]–[85], [87]–[105], [108], [109], [111], [112], [114], [123], [124], [126]–[132], [136], [139], [142], [143], [149]–[153], [164]
Graph-based Representation	Program Dependence Graph (PDG)	8	[3], [39], [79], [96], [104], [107], [113], [139]
	Bytecode Dependency Graph (BDG)	2	[79], [139]
	Control Flow Graph (CFG)	12	[1]–[3], [41], [47], [48], [82], [86], [88], [99], [102], [140]
	Data Flow Graph (DFG)	5	[1], [2], [41], [120], [122]
Token-based Representation	Token-based Representation	16	[8], [9], [35], [46], [76], [82], [117]–[119], [121], [133]–[135], [137], [148], [153]
	TF-IDF	4	[83], [140], [142], [144]
Metrics-based Representation	Metrics-based Representation	8	[34], [74], [75], [78], [99], [138], [145], [146]
Text-based Representation	Text-based Representation	5	[6], [49], [99], [110], [125]
Feature-based Representation	Application Programming Interface (API)	2	[129], [141]
	Intermediate Representation (IR)	2	[48], [106]
Image-based Representation	Image-based Representation	1	[116]

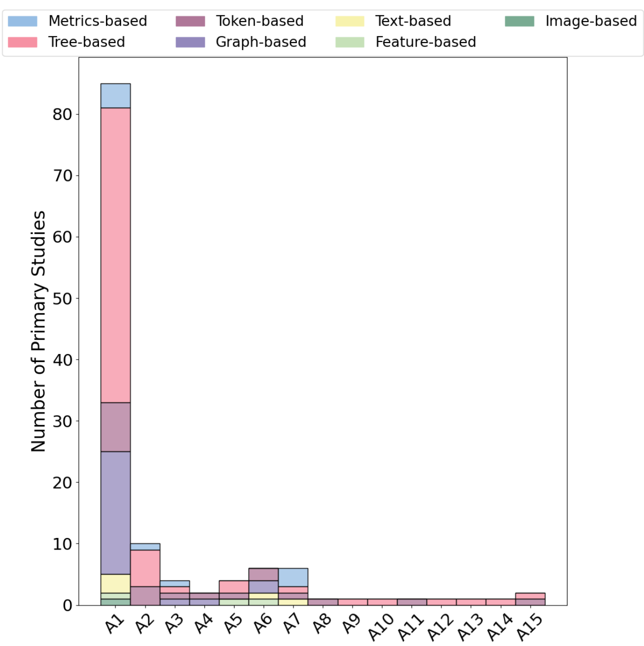

FIGURE 8. Studies using different code representations over various code similarity applications.

VI. EVALUATION DOMAIN: RQ4, RQ5, AND RQ6

In this section, we aim to answer RQ4 and RQ5 which are related to the datasets and metrics used to assess the performance of each ML technique in each code similarity application domain. This section also aims to answer RQ6 by identifying the most performing ML techniques in each application domain.

A. DATASETS EMPLOYED IN EACH CODE SIMILARITY APPLICATION DOMAIN (RQ4)

Table 7 delineates the datasets utilized in primary studies, detailing their applications, usage, frequency, and citations. Notably, 80 studies have employed public datasets to evaluate their machine-learning algorithms, while 24 studies have relied on proprietary, non-public datasets they developed. Among these public datasets, BigCloneBench [42] stands out as the most utilized, supporting research in diverse areas such as code clone detection, code clone validation, code prediction, and software review using ML algorithms. BigCloneBench serves as a significant big data benchmark for assessing clone detection methods across inter-project Java repositories, containing 6 million true clone pairs and 260,000 false clone pairs from approximately 25,000 software systems, totaling 365 million lines of code. The dataset was meticulously compiled by mining the IJaDataset for clones across ten common functionalities and subsequently verified through a manual validation process by expert judges to ascertain true and false positives.

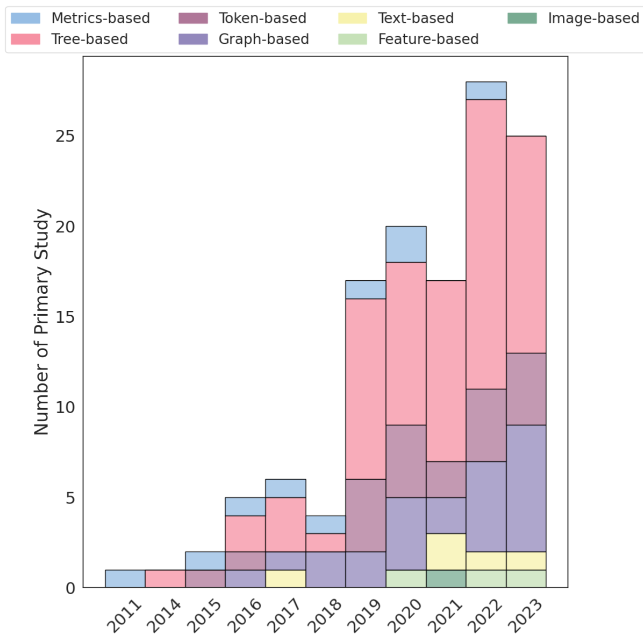

FIGURE 9. Distribution of selected studies in code representation type over years.

Other notable datasets include Google Code Jam (GCJ) [159] and OJClone [158]. The GCJ dataset, utilized for semantic and cross-language clone detection, comprises 1669 Java projects from 12 different competition problems. Conversely, the OJClone dataset contains solutions to 104 different programming challenges in C, and it has been employed in 15 code clone detection studies and 2 source code plagiarism detection tasks. Furthermore, the Atcoder dataset, curated by Perez et al. [130], is utilized extensively for cross-language code clone detection. Collected from a renowned competitive programming website, this dataset includes roughly 45,000 code fragments in Java and Python, annotated to indicate clone relationships. Studies focusing on less common applications such as vulnerability detection, code change inspection, issue-commit link recovery, and algorithm classification typically employ custom datasets.

Table 8 catalogs the publicly accessible code similarity datasets, detailing their names, download specifics, supported programming languages, and associated citations. Furthermore, Figure 10 illustrates the distribution of programming languages utilized in primary studies. Java predominates, being employed in 52.9% of the studies, followed by C, C++, and Python, which account for 21.7%, 9.6%, and 8.9% of the studies, respectively. Additionally, there is notable utilization of less common languages for code similarity measurement, including Go [80], PHP [72], Scratch [46] and C# [129], [141], [142], [148].

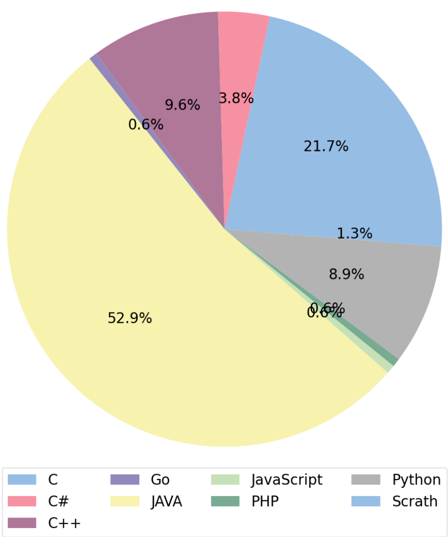

FIGURE 10. Distribution of programming languages used by collected primary studies.

Key Findings Of RQ4

- Public datasets are predominantly used to evaluate ML approaches in code similarity research, with BigCloneBench standing out as the most frequently used dataset across various applications and Atcoder being prominent in studies focusing on cross-language code clone detection.
- There is significant diversity in the programming languages used in ML-based research on code similarity, including PHP, Go, C#, C++, Scratch, Python, and C. However, Java is the most commonly used programming language.

B. EVALUATION METRICS CONSIDERED TO MEASURE PERFORMANCE OF ML TECHNIQUES (RQ5)

Table 9 delineates the evaluation metrics employed in the primary studies to assess the efficacy of ML algorithms. According to the data presented in this table, a total of ten distinct evaluation metrics have been utilized across the reviewed studies, categorized into classification, regression, and time metrics. A detailed description of each of these metrics follows:

1) Classification Metrics

- **Accuracy (Acc)**
  Several primary studies reviewed herein utilize accuracy as a performance metric; however, its application within machine learning contexts is fraught with complications. Particularly in scenarios involving highly unbalanced datasets, elevated accuracy levels may prove deceptive, offering a distorted view of the model's true efficacy. It is defined as the ratio of the number of correct predictions to the total number of predictions made. Formally, accuracy can be expressed by Eq 1:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

  where
  - True Positives (TP) are the number of instances where the model correctly predicts the positive class.
  - True Negatives (TN) are the number of instances where the model correctly predicts the negative class.
  - False Positives (FP) are the number of instances where the model incorrectly predicts the positive class.
  - False Negatives (FN) are the number of instances where the model incorrectly predicts the negative class.
- **Precision (Prec)**
  Precision is a metric used in classification tasks to evaluate the accuracy of the positive predictions made by a model. Precision is defined as the number of true positives divided by the number of true positives plus the number of false positives. Mathematically, precision can be expressed by Eq 2:

$$Prec = \frac{TP}{TP + FP} \tag{2}$$

- **Recall (Rec)**
  Recall is a metric used in classification tasks that measures the ability of a model. Specifically, it quantifies the proportion of actual positives that have been correctly identified by the model. The formula for recall is expressed in EQ 3:

$$Rec = \frac{TP}{TP + FN} \tag{3}$$

- **Mean Average Precision (MAP)**
  Mean Average Precision is a metric commonly used in information retrieval tasks and ranking systems. It measures the precision across multiple queries or instances, averaged over each relevant item retrieved. The formula for MAP is expressed in EQ 4:

$$MAP = \frac{1}{Q}\sum_{q=1}^{Q} AP(q) \tag{4}$$

TABLE 7. Datasets employed for each code similarity measurement application.

Application	Sub-Application	Code	Dataset	Count	Papers
Code Clone Detection	Single Language	A1	BigCloneBench [42]	50	[1]–[3], [39]–[41], [43], [44], [73], [77], [78], [81]–[84], [89], [91]–[101], [103]–[114], [116], [117], [120]–[123], [125], [127], [128], [164], [165]
			Google Code Jam (GCJ) [159]	11	[3], [41], [44], [91], [94], [97], [98], [101], [105], [107], [111]
			OJClone [158]	23	[1], [43], [77], [84], [85], [88], [90], [93], [100], [102]–[106], [112], [114]–[116], [122], [123], [125], [127], [128]
			SeSaMe dataset [37]	1	[107]
			SemanticCloneBench [166]	1	[118]
			Self-made	7	[74], [79], [80], [86], [99], [106], [113]
	Cross-language	A2	AtCoder [130]	9	[4], [129]–[136]
			CodeChef [4]	1	[4]
			Google Code Jam (GCJ) [159]	4	[129], [133]–[135]
			Self-made	1	[131]
	Validation	A3	BigCloneBench [42]	2	[137], [138]
			Self-made	1	[139]
	Application Clone	A4	Self-made	1	[140]
Code Similarity	Software	A5	Self-made	3	[141]–[143]
	Source Code	A6	CodeNet [167]	1	[48]
			OJClone [158]	1	[144]
			Self-made	2	[46], [47]
Plagiarism Detection	Source Code	A7	OJClone [158]	2	[5], [6]
			Programming Homework Dataset for Plagiarism Detection [168]	1	[146]
			Self-made	2	[6], [145]
	Software	A8	Self-made	1	[148]
Vulnerability Detection		A9	Self-made	2	[36], [37]
Code Change Inspection		A10	Self-made	1	[149]
Code Prediction		A11	BigCloneBench [42]	1	[9]
Software Review		A12	BigCloneBench [42]	1	[150]
Program Repair		A13	Defects4J [169]	1	[151]
Issue-Commit Link Recovery		A14	Self-made	1	[152]
Algorithm Classification		A15	Self-made	1	[153]

where Q is the total number of queries, and $AP(q)$ is the average precision for query q. Average precision itself is the average of the precision scores at each position in the ranking list where a relevant item is retrieved.

- **False Negative Rate (FNR)**
  The False Negative Rate (FNR), also known as the Miss Rate, measures the proportion of actual positive instances that are incorrectly classified as negative by the model. Formally, the FNR can be defined as follows:

$$FNR = \frac{FN}{TP + FN} \tag{5}$$

- **False Positive Rate (FPR)**
  The False Positive Rate (FPR), also known as the Fall-out Rate, quantifies the proportion of actual negative instances that are incorrectly classified as positive by the model. Formally, the FPR can be defined as follows:

$$FPR = \frac{FP}{FP + TN} \tag{6}$$

- **Area Under the Curve (AUC)**
  The AUC, or Area Under the Receiver Operating Characteristic (ROC) Curve, is widely used metric for evaluating the performance of classification models, particularly in terms of their ability to distinguish between classes. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. AUC provides a single measure of how well a model can distinguish between classes, with higher values indicating better performance. Formally, the AUC can be expressed as:

$$AUC = \int_{x=0}^{1} TPR(FPR^{-1}(x))dx \tag{7}$$

- **F1-Score/F-measure (F1)**
  The F1-Score, also known as the F-measure or the F1-Score, is a metric used to evaluate the performance of binary classification models, especially in situations where there are imbalanced classes or when the importance of precision and recall is equally weighted. It is the harmonic mean of precision and recall, providing a balance between them. Precision measures the model's accuracy in identifying positive instances among all instances it labeled as positive, while recall measures the model's ability to identify all positive instances in the dataset. The F1-Score reaches its best value at 1 (perfect

precision and recall) and its worst at 0. Formally, the F1-Score can be expressed in Eq 8:

$$F1 = 2 \cdot \frac{Prec \cdot Rec}{Prec + Rec} \tag{8}$$

2) Regression Metrics

- **Mean Absolute Error (MAE)**
  The Mean Absolute Error (MAE) is a widely used metric for evaluating the performance of regression models. It measures the average magnitude of the errors between the predicted values and the actual values, without considering their direction. The MAE is defined as follows:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{9}$$

  where: - n is the number of observations in the dataset,
  - y_i is the actual value of the i-th observation,
  - $\hat{y}_i$ is the predicted value for the i-th observation,
  - $|\cdot|$ denotes the absolute value.
- **Mean Squared Error (MSE)**
  The Mean Squared Error (MSE) is a fundamental metric used in regression analysis to evaluate the performance of a regression model. It measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value. The MSE is defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{10}$$

  where: - n is the number of observations,
  - y_i is the actual value for the i-th observation,
  - $\hat{y}_i$ is the predicted value for the i-th observation.

3) Time Metrics

- **Training Time (TT) and Inference Time (IT)**
  The time-related metrics employed in the selected primary studies include inference time and training time. Training time refers to the period it takes to develop an ML model using training data. Inference time, in some studies, also known as prediction time, is the time it takes for a trained model to make a prediction on new data.

The most frequently employed metrics in this field are Precision, Recall, and F1-Score, utilized 83, 83, and 75 times respectively. Accuracy follows these, with 22 instances of application. The AUC (Area Under Curve) metric, indicative of classification accuracy, is particularly noted for its utility in representing the efficacy of an algorithm—the greater the AUC value, the larger the area beneath the ROC curve, signifying higher classification accuracy [37]. This metric is extensively used to evaluate the performance of systems in code clone detection [80], [81], [83], [137], code clone validation [138], and vulnerability detection [37]. The MAE (Mean Absolute Error) metric, which quantifies the discrepancy between measured and true values, is applied in [139] for assessing the accuracy of labeling code clones using ML algorithms. Furthermore, MSE (Mean Squared Error) is utilized in [47] to compute the loss between the predicted scores and the ground truth in code similarity assessments. Some studies employ time performance metrics to evaluate the effectiveness of their ML algorithms. Notably, training time [41], [76], [88], [101], [104] and average inference time [41], [76], [79], [88], [99], [104] are most frequently reported. Additionally, the study cited in [40] evaluates time performance by calculating the time cost required to process 3.5 million lines of code.

Key Findings Of RQ5

- Classification-based metrics such as Precision, Recall, F1-Score, Accuracy, and AUC are most prevalent when assessing the performance of ML techniques on code similarity applications.
- Mean Absolute Error (MAE) is used for code clone validation, while Mean Squared Error (MSE) is applied in code similarity measurement evaluations.
- Several studies also highlight the efficiency of their algorithms by evaluating prediction time and training time as key performance indicators.

C. BEST PERFORMING ML TECHNIQUES (RQ6)

The comparison and evaluation of ML algorithms are challenging due to the diversity of applications, benchmarks, and the absence of standardized similarity measures [64]. To address this research question, we present the results from primary studies where one or more code similarity techniques have been applied to the same dataset. The findings are summarized in Table 10, Table 11, Table 12, and Table 13, with name and citation of the ML techniques, the values of evaluation parameters, and the datasets utilized for assessment. These tables categorize the different applications of ML algorithms. Some applications have not been included in the tables for performance comparison due to either a lack of comparisons across various algorithms and tools or insufficient sample sizes for those specific applications.

Table 10 presents a summary of tools and algorithms dedicated to code clone detection, as proposed or included for comparative analysis in selected scholarly works. both ML-based and traditional tools/algorithms for code clone detection task are included. Given the extensive body of primary research in this area, we have selectively included studies that employ the BigCloneBench, Google Code Jam, and OJClone datasets in our performance comparison table. We encompasses commonly utilized performance metrics included precision, recall, F1-score to summarize the models performance. However, not every metric is applied to every tool across all datasets. Performance metrics are given specific numeric values, and some cells are marked with a dash to indicate unavailable data. For more specific performance comparison for ML-based code clone detection tools/algorithms, we depict the comparative results of the studies utilizing the datasets BigCloneBench [42], OJClone [158], and GoogleCodeJam [159]. These results are illustrated in Figures 11. Figure 11 presents the performance metrics of various ML-based algorithms on three most common datasets: BigCloneBench, Google Code Jam, and OJClone. From up to down, each column represents the performance in each dataset in precision, recall and F1-score. From left to right, the columns are dedicated to each dataset respectively. The tools/algorithms are arrayed in descending order

TABLE 8. Public dataset proposed for code similarity measurement applications.

Dataset	Download Link	Programming Language	Papers
BigCloneBench (BCB)	https://github.com/clonebench/BigCloneBench	Java	[42]
Google Code Jam (GCJ)	https://github.com/Jur1cek/gcj-dataset	Java	[159]
OJClone/POJ	Expired	C	[158]
Atcoder for cross-language code clone detection	https://daniel.perez.sh/research/2019/cross-language-clones/	Java/Python	[130]
Defects4J	http://defects4j.org	JAVA	[169]
SeSaMe Dataset	https://github.com/FAU-Inf2/sesame	JAVA	[170]
Programming Homework Dataset for Plagiarism Detection	https://ieee-dataport.org/open-access/programming-homework-dataset-plagiarism-detection	C/C++	[168]
Siamese dataset	https://github.com/sunhao123456789/siamese_dataset	C	[37]
CodeChef for cross-language code clone detection	https://www.kaggle.com/datasets/arjoonn/codechef-competitive-programming	Java/C	[4]
CodeNet	https://developer.ibm.com/exchanges/data/all/project-codenet/	C++	[167]
SemanticCloneBench	https://drive.google.com/file/d/1KicfslV02p6GDPPBjZHNlmiXk-9IoGWl/view	C/CS/Java/Python	[166]

TABLE 9. Evaluation metrics of primary studies.

Metrics Type	Metrics	Count	Papers
Classification Metrics	Acc	24	[1], [35], [37], [39], [48], [75], [79], [80], [85], [99], [104], [108], [116], [118], [132], [137]–[140], [142], [146], [147], [149], [153]
	AUC	8	[37], [80], [81], [83], [124], [126], [137], [138]
	F1	75	[1]–[6], [9], [35]–[37], [39], [41], [43], [44], [46], [48], [49], [73], [77], [79], [82], [84]–[86], [88]–[98], [100]–[107], [109]–[117], [121], [123], [124], [127]–[132], [134]–[136], [138], [140]–[142], [144], [146], [148], [150], [152], [164], [165]
	FNR	2	[36], [37]
	FPR	3	[36], [37], [148]
	Prec	83	[1]–[6], [9], [35], [36], [39]–[41], [43], [44], [46], [48], [49], [73], [74], [76]–[79], [82]–[86], [88]–[107], [109]–[118], [121], [123]–[125], [127]–[132], [134]–[136], [138], [140]–[142], [144], [146], [148]–[150], [152], [164], [165]
	Rec	83	[1]–[6], [9], [35], [36], [39]–[41], [43], [44], [46], [48], [49], [73], [74], [77]–[79], [82]–[86], [88]–[118], [121], [123]–[125], [127]–[132], [134]–[136], [138], [140]–[142], [144], [146], [148]–[150], [152], [164], [165]
	MAP	4	[125]–[127], [133]
Regression Metrics	MAE	1	[139]
	MSE	1	[47]
Time Metrics	IT	7	[40], [41], [76], [79], [88], [99], [104]
	TT	5	[41], [76], [88], [101], [104]

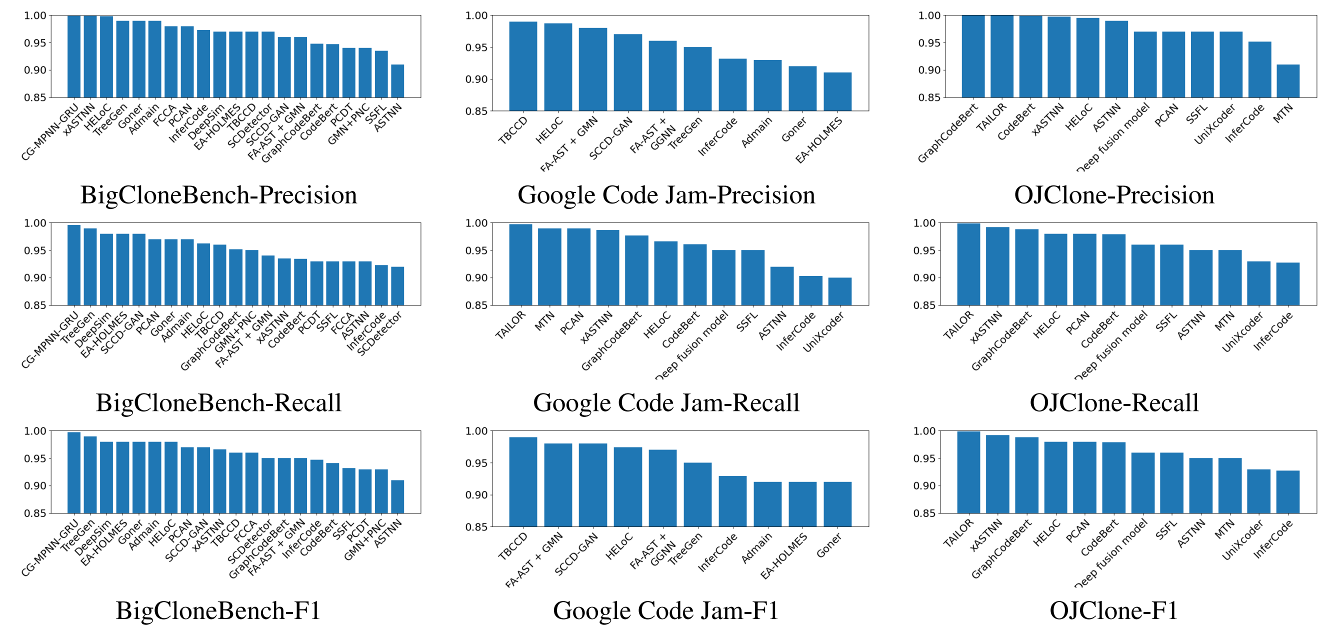

FIGURE 11. Performance of ML code clone detection techniques on different datasets.

TABLE 10. Performance Comparison of Code Clone Detection Tools/Algorithms.

Tools/Algorithms	BigCloneBench			Google Code Jam			OJClone		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
ASTNN [84]	0.91	0.93	0.91	0.89	0.92	0.88	0.99	0.92	0.95
Admain [97]	0.99	0.97	0.98	0.93	0.91	0.92	-	-	-
Bi-LSTM [85]	-	-	-	-	-	-	0.18	0.97	0.32
Boost [126]	0.942	0.946	-	-	-	-	-	-	-
CCLearner [40]	0.93	-	-	-	-	-	0.0651	0.8324	0.1207
CDLH [77]	0.92	0.74	0.82	0.46	0.70	0.55	0.47	0.93	0.57
CG-MPNN-GRU [101]	0.9991	0.9956	0.9973	0.8810	0.9902	0.9324	-	-	-
CNN [85]	-	-	-	-	-	-	0.29	0.43	0.34
CSME [90]	-	-	-	-	-	-	0.8036	0.7192	0.7583
CloneWork [171]	-	-	-	-	-	-	0.4604	0.3210	0.3890
Code-RNN [85]	-	-	-	-	-	-	0.26	0.97	0.41
Code2vec [144]	-	-	-	-	-	-	0.833	0.658	0.736
CodeBert [165]	0.947	0.934	0.941	-	-	-	0.999	0.961	0.979
DLC/RtvNN/RAE [76]	0.95	0.01	0.01	0.20	0.90	0.33	0.61	0.33	0.26
Deckard [31]	0.93	0.02	0.03	0.45	0.44	0.44	0.99	0.05	0.10
Deep fusion model [88]	-	-	-	-	-	-	0.97	0.95	0.96
DeepSim [41]	0.97	0.98	0.98	0.71	0.82	0.76	0.70	0.83	0.76
EA-HOLMES [107]	0.97	0.98	0.98	0.91	0.93	0.92	-	-	
FA-AST + GGNN [91]	0.85	0.90	0.88	0.96	0.98	0.97	-	-	-
FA-AST + GMN [91]	0.96	0.94	0.95	0.98	0.97	0.98	-	-	-
FCCA [82]	0.98	0.93	0.96	0.95	0.87	0.91	0.941	0.873	0.906
GGNN [107]	0.72	0.89	0.79	0.72	0.87	0.79	-	-	-
GMN+PNC [2]	0.94	0.95	0.93	-	-	-	-	-	-
Goner [44]	0.99	0.97	0.98	0.92	0.92	0.92	-	-	-
GraphCodeBert [172]	0.948	0.952	0.950	-	-	-	1.00	0.977	0.988
HELoC [105]	0.998	0.962	0.980	0.987	0.959	0.974	0.995	0.966	0.980
InferCode [114]	0.973	0.923	0.947	0.932	0.926	0.929	0.952	0.903	0.927
Jiang et al [104]	0.985	0.972	0.979	-	-	-	0.994	0.995	0.994
Jo et al. [164]	0.96	0.96	0.96	-	-	-	-	-	
LSTM [85]	-	-	-	-	-	-	0.19	0.95	0.31
MTN [85]	-	-	-	-	-	-	0.91	0.99	0.95
N-gram [144]	-	-	-	-	-	-	0.71	0.59	0.63
NiCad [155]	0.89	-	-	-	-	-	-	-	-
Oreo [78]	0.895	-	-	-	-	-	-	-	-
PCAN [103]	0.98	0.97	0.97	-	-	-	0.97	0.99	0.98
PCDT [109]	0.94	0.93	0.93	-	-	-	-	-	-
PDG+HOPE [173]	-	-	-	-	-	-	0.762	0.07	0.128
PGC+GGNN [174]	-	-	-	-	-	-	0.773	0.436	0.558
SCCD-GAN [94]	0.96	0.98	0.97	0.97	0.99	0.98	-	-	-
SCDetector [175]	0.97	0.92	0.95	0.90	0.87	0.89	0.95	0.89	0.92
SSFL [88]	0.935	0.930	0.932	-	-	-	0.97	0.95	0.96
Sia-RAE [89]	0.9925	-	-	-	-	-	-	-	-
SourcererCC [156]	0.92	0.02	0.06	0.42	0.11	0.17	0.13	0.71	0.23
TAILOR [1]	-	-	-	-	-	-	1.00	0.997	0.999
TBCCD [43]	0.97	0.96	0.96	0.99	0.99	0.99	-	-	-
TBCNN [158]	0.90	0.81	0.85	0.91	0.89	0.90	0.901	0.813	0.858
Tree-LSTM [85]	-	-	-	-	-	-	0.27	1.00	0.43
TreeGen [98]	0.99	0.99	0.99	0.95	0.95	0.95	-	-	-
UniXcoder [176]	-	-	-	-	-	-	0.97	0.90	0.93
WICE-SNN [144]	-	-	-	-	-	-	0.67	0.83	0.74
Word2vec [144]	-	-	-	-	-	-	0.79	0.42	0.55
xASTNN [123]	0.999	0.935	0.966	-	-	-	0.997	0.987	0.992
Zhang et al [93]	0.959	0.946	0.952	-	-	-	0.993	0.983	0.988
Zhang et al [112]	0.961	0.952	0.956	-	-	-	0.997	0.985	0.991

of performance, as indicated by the values beneath each corresponding bar.

In the BigCloneBench dataset, our analysis indicates that several machine learning-based tools and algorithms attained nearly flawless precision scores of 0.99. Noteworthy among these are CG-MPNN-GRU [101], xASTNN [123], HELoC [105], TreeGen [98], Goner [44], and Admain [97]. Regarding recall, CG-MPNN-GRU [101] and TreeGen [98] exhibited the highest scores, reaching 0.99. Similarly, for the F1-score, both CG-MPNN-GRU and TreeGen also recorded the highest figures at 0.99. For the Google Code Jam dataset, TBCCD [43] demonstrated superior performance across all metrics, achieving a score of 0.99 in each. HELoC [105] followed closely with a precision of 0.987. From the perspective of recall, SCCD-GAN [94] also achieved near-perfect performance with a score of 0.99. In terms of F1-score, both FA-AST + GMN [91] and SCCD-GAN were rated second, each achieving 0.98. In the OJClone dataset, GraphCodeBert [172] and TAILOR [1] reached perfect precision scores. For recall, TAILOR, MTN [85], and PCAN [103] achieved the highest marks at 0.99. In the F1-score metric, TAILOR, xASTNN, and GraphCodeBert led with scores of 0.999, 0.992, and 0.998, respectively. This comprehensive evaluation underscores the capabilities of these advanced algorithms in handling various aspects of code similarity and clone detection across distinct datasets.

Table 11 summarizes comparison studies related to cross-language code clone detection tasks which use the Atcoder dataset [130]. Perez et al. [130] firstly propose machine-learning-based approach to detect cross-language and introduce Atcoder dataset which would be widely used in this field. The approach is based on semi-supervised ML, combing with an unsupervised learning approach to learn token-level vector representations and an LSTM-based neural network to detect code clones. Nafi et al. [129] introduce the CLCDSA model for cross-language code clone detection, utilizing distinct syntactic features of source code and applying an action filter based on API call similarity. When compared with LICCA [177] and CLCMiner [178], CLCDSA demonstrates superior performance. Similarly, Ling et al. [131] propose a tree autoencoder (TAE) architecture that employs unsupervised learning to pretrain abstract syntax trees, followed by using a tree-based LSTM for detecting clones. This approach exhibits higher performance compared to the work of Perez et al. [130]. Yahya et al. [132] present CICD-I, a deep neural network-based method for detecting cross-language code clones utilizing InferCode. Compared with CLCDSA [129] and the work of Perez et al. [130], CICD-I outperforms the existing approaches by an average of 15%. Furthermore, Mehrotra et al. [4] introduce RUBHUS, a semi-supervised, DL-based tool for cross-language code clone detection. RUBHUS enriches the abstract syntax tree with control and data flow graphs. Its performance surpasses that of state-of-the-art single language tools such as TBCCD [43], FA-AST+GGNN [91], and FA-AST+GMN [91], particularly in terms of precision, recall, and F1-score. Tao et al. [134] introduce C4, a contrastive learning-based model for cross-language code clone detection. C4 leverages the pre-trained CodeBERT model to convert code snippets from different programming languages into high-dimensional vector representations. The model is fine-tuned using a contrastive learning objective to effectively distinguish clone pairs from non-clone pairs. Its performance surpasses that of state-of-the-art baselines, achieving precision, recall, and F1-scores of 0.94, 0.90, and 0.92, respectively. Fang et al. [135] introduce TCCCD, a triplet-based, machine learning approach for cross-language code clone detection. TCCCD leverages the pre-trained model UniXcoder to map programs written in different languages into the same vector space and fine-tunes the model using triplet learning. Its performance surpasses that of state-of-the-art tools such as CLCMiner, CLCDSA, and C4. Lin et al. [124] introduce XCode, a novel cross-language code representation method with large-scale pre-training. XCode utilizes abstract syntax trees and ELMo-enhanced variational autoencoders to pre-train source code language models, which are then distilled into a Shared Encoder-Decoder (SED) architecture. Its performance surpasses that of state-of-the-art single language tools such as BiLSTM, Tree-LSTM, and Code-RoBERTa.

TABLE 11. Performance comparison studies related to cross-language code clone detection using ML in the Atcoder dataset.

Algorithms/Tools compared	Performance		
	Precision	Recall	F1-Score
AST-GNN [136]	0.92	0.70	0.80
Nafi et al. [129]	0.19	0.90	0.32
LICCA [177]	0.14	0.32	0.194
C4 [134]	0.94	0.90	0.92
CLCMiner [178]	0.09	0.13	0.11
CLCDSA [129]	0.184	0.81	0.30
Randomly Initialized Encoder [131]	0.28	0.92	0.43
Pretrained Encoder [131]	0.31	0.93	0.47
CICD-I [132]	0.99	0.63	0.78
TBCCD [43]	0.71	0.00	0.00
TCCCD [135]	0.96	0.92	0.94
FA- AST+GGNN [91]	0.75	0.95	0.84
FA- AST+GMN [91]	0.60	0.66	0.63
RUBHUS [4]	0.90	0.91	0.90
XCode [124]	0.891	0.937	0.913

Table 12 presents a summary of comparative studies on detecting source code plagiarism using ML techniques. Acampora et al. [35] developed a method based on fuzzy clustering, optimized by a genetic algorithm, to cluster code tokens and identify plagiarized files. Through evaluation with their self-made data, their model is superior to ANFIS, SVM, RF, and LDA. We observe two OJClone dataset usage in source code plagiarism detection. Yasaswi et al. [6] employ a pre-trained character-level Recurrent Neural Network enhanced with deep features to capture non-contiguous interactions within code, aiming to detect source code plagiarism. They benchmark the effectiveness of deep features against Engels et al. [179], who use textual features, and Gorthi et al. [145], who employ source code metrics, reporting a 9.5% improvement in F1-score for binary classification and a 5% enhancement in three-way classification. Fokam et al. [5] integrate a contrastive learning paradigm into an Abstract Syntax Tree-based

TABLE 12. Performance Comparison studies related to source code plagiarism detection using ML.

Dataset	ML Techniques/Tools	Performance			
		Precision	Recall	F1-Score	Accuracy
Self-made [35]	SVM [35]	1.00	0.7	0.8235	0.75
	RF [35]	1.00	0.6	0.75	0.6667
	LDA [35]	1.00	0.4	0.5714	0.5
	PCA-GA-ANFIS [35]	0.9091	1.00	0.9524	0.9167
	SVD_ANFIS [35]	0.8333	1.00	0.8333	0.9091
OJClone [158]	Textual features [179]	0.413	0.490	0.423	-
	Souce-code metrics [145]	0.430	0.570	0.460	-
	Deep features [6]	0.470	0.653	0.513	-
	Deep features + Textual features + metrics [6]	0.490	0.660	0.543	-
	ASTNN [84]	0.972	0.899	0.935	-
	ASTNN-c [5]	0.975	0.992	0.983	-
Programming Homework Dataset [168]	Xgboost [146]	0.89	0.82	0.85	0.90
	SVM [146]	0.87	0.66	0.73	0.90

TABLE 13. Performance Comparison studies related to source code similarity using ML.

Dataset	ML Techniques/Tools	Performance				
		Precision	Recall	F1-Score	Accuracy	MSE(10^{-3})
Self-made [47]	QAP [180]	-	-	-	-	0.0
	VJ [181]	-	-	-	-	14.41
	Hungarian [182]	-	-	-	-	15.97
	HED [183]	-	-	-	-	8.67
	GCN [184]	-	-	-	-	4.58
	GraphSAGE [185]	-	-	-	-	3.61
	funcGNN [47]	-	-	-	-	1.94
Self-made [46]	fastText [46]	0.79	0.97	0.87	-	-
	CNN [46]	0.96	0.74	0.84	-	-
	LSTM [46]	0.92	0.74	0.82	-	-
	BiLSTM [46]	0.98	0.92	0.95	-	-
OJClone [158]	Word2Vec [163]	0.79	0.42	0.55	-	-
	N-gram [186]	0.71	0.59	0.63	-	-
	Code2Vec [162]	0.69	0.56	0.62	-	-
	WICE-SNN [144]	0.67	0.83	0.74	-	-
CodeNet [167]	KN [48]	0.79	0.76	0.76	0.76	-
	DT [48]	0.75	0.74	0.74	0.74	-
	RF [48]	0.87	0.87	0.86	0.86	-
	MLP [48]	0.85	0.85	0.84	0.84	-
	GNB [48]	0.69	0.69	0.58	0.59	

Neural Network (ASTNN) [84] to enhance performance in the source code plagiarism detection task. By comparing with the original ASTNN model [84], their approach demonstrates a 5% improvement in F1-score. Awale et al. [146] implement a set of comprehensive handcrafted features in source code for ML algorithms, using the public Programming Homework dataset for plagiarism detection [168]. They test their features with SVM and XGBoost, confirming their effectiveness.

Table 13 encapsulates the comparative studies on measuring source code similarity utilizing ML techniques. Nair et al. [47] introduce funcGNN, a Graph Neural Network (GNN) based model that utilizes labeled Control Flow Graphs (CFGs) to predict the graph edit distance (GED) between code pairs. In comparison to various ML models like QAP [180], VJ [181], Hungarian [182], HED [183], GCN [184], and GraphSAGE [185], funcGNN exhibits a lower error rate, enhanced scalability, and improved speed. Zhang et al. [46] propose a siamese-based bidirectional Long Short-Term Memory (BiLSTM) network to assess code similarity in Scratch programming language, using code pairs sourced from the official Scratch website. Their method is compared with fastText, CNN, and LSTM, showing superior performance. Xie et al. [144] develop a siamese neural network that employs multiple word embedding methods, including an unsupervised pre-trained model and Term Frequency-Inverse Document Frequency (TF-IDF), to semantically measure code similarity. They evaluate their model in public OJClone dataset. Compared with single word embedding methods like Word2Vec [163], N-gram [186], and Code2Vec [162], their approach significantly enhances detection performance. There is one study that utilizes CodeNet as its benchmark. Boldini et al. [48] concentrate on the explainability and generalization of ML algorithms for code similarity assessment. They utilize a graph-focused representation of code to demonstrate comparative performance across multiple ML algorithms including K-Nearest Neighbors (KNN), Decision Trees (DT), Random Forest (RF), Multi-Layer Perceptron (MLP), and Gaussian Naive Bayes (GNB).

Key Findings Of RQ6

- Code clone detection and cross-language code clone detection feature relatively unified standards, using public datasets and consistent evaluation metrics.
- For single language code clone detection, the best performance models vary based on different public datasets. Algorithms such as CG-MPNN-GRU and

TreeGen demonstrated near-perfect scores in precision, recall, and F1-score on the BigCloneBench dataset, while TBCCD exhibited outstanding performance across all metrics in the Google Code Jam dataset. Additionally, in the OJClone dataset, algorithms like GraphCodeBert and TAILOR achieved perfect precision scores, with TAILOR also reaching the highest marks in recall and F1-score.
- ML-based approaches using the Atcoder dataset indicate significant advancements, with methods like CLCDSA and RUBHUS showing superior performance compared to traditional tools. Novel methods like C4 and TCCCD introduce contrastive and triplet-based learning paradigms, enhancing cross-language detection capabilities.
- Other applications face challenges due to the lack of standardized public datasets for cross-study comparisons or an absence of general comparative studies.

VII. DISCUSSION

A. APPLICATION DOMAIN

The findings of this study highlight the extensive of ML techniques for code similarity across various software engineering applications, including code clone detection, software plagiarism detection, vulnerability detection, and program repair. However, the current landscape still reveals two major challenges (1) the lack of focus on real-world practices, such as code review tools or academic integrity system. (2) the practical efficiency of these techniques when deploy in real-world environments.

- **Limited focus on real-world applications:** Despite the variety of applications discussed in the literature—such as code clone detection, vulnerability detection, and program repair—some real-world applications remain largely ignored. One of the key areas for potential improvement is in automated code review tools. By integrating ML-based code similarity techniques, modern code review platforms could provide real-time feedback to developers, detecting redundant implementations and suggesting refactoring opportunities based on prior code submissions. Large-scale software projects and open-source ecosystems particularly require efficient and intelligent code review tools to handle the high volume of contributions. For example, enterprises managing vast codebases, such as Google, Microsoft, and Facebook, often have thousands of developers contributing code daily. Reviewing such massive amounts of code manually is impractical, and automated similarity detection could assist in identifying redundant or inefficient implementations, streamlining the review process, and ensuring consistency across projects. Another valuable application is in academic integrity and plagiarism detection systems. While code similarity detection-based plagiarism detection tools are already widely used, most existing research and applications have primarily focused on improving detection accuracy and performance, rather than applying in real academic integrity systems.
- **Practicality & efficiency concerns in real-world use:** Although numerous studies report high accuracy and superior performance of ML-based code similarity detection models, there is a lack of systematic empirical evaluation on their efficiency and scalability. According to our findings, while various ML architectures (RNNs, CNNs, GNNs) have shown strong performance in code similarity measurement, their high computational complexity makes them impractical for large-scale, real-time use. For instance, GNNs require pairwise comparisons of large graph structures, making them computationally expensive [187]. AST and CFG approaches involve recursive computations, leading to high memory consumption [?]. These inefficiencies limit real-world adoption, particularly in scenarios where millions of code fragments need to be compared rapidly.
In the future, we need focus more on efficiency and practical usability of code similarity measurement in real-world. Systematic empirical evaluation studies of efficiency and scalability like [188], [188]–[190] can be applied to assess the practical viability of ML-based code similarity measurement models. Building upon these evaluations, future work should explore lightweight and scalable architectures that reduce computational overhead

B. MACHINE LEARNING DOMAIN

1) Devised ML Techniques(RQ2)

Over recent years, ML techniques have significantly evolved in the domain of code similarity measurement, with a notable trend towards deep learning architectures. Among these, supervised learning remains the predominant approach, covering both traditional ML methods such as SVMs, Decision Trees, and Random Forests, as well as deep learning models like CNNs, RNNs, and Transformers.

Recent research has explored the integration of semi-supervised and self-supervised learning architectures to enhance accuracy while reducing reliance on labeled datasets. Among these approaches, contrastive learning has emerged as a particularly effective technique, especially in code similarity measurement tasks, where the objective is to identify functionally equivalent code pairs. This method operates by bringing semantically similar examples (positive pairs) closer together in the embedding space while pushing dissimilar examples (negative pairs) further apart, thereby improving the model's capacity to capture meaningful relationships between code snippets. Contrastive learning is particularly advantageous in code similarity measurement due to the scarcity of labeled datasets. By enabling models to learn robust representations from large-scale unlabeled code corpora, it enhances performance in tasks such as code similarity analysis and clone detection. Prior studies have primarily applied contrastive learning to code clone detection [105], [126], [134] and code plagiarism detection [5]. Given its effectiveness, further research is warranted to explore and expand its applications in code similarity measurement, thereby advancing the state of the art in this domain.

Alongside the dominance of supervised learning, transfer learning has emerged as a key trend, particularly with the rise of pre-trained models specialized for code similarity measurement. In recent years, transformer-based models such as CodeBERT [165], GraphCodeBERT [172], and UniXCode [176] have gained significant traction. These models have been trained on massive multi-language code corpora, allowing them to effectively capture both syntactic and semantic features of code. Additionally, large language models (LLMs) have recently emerged as a promising direction in code similarity measurement. Researchers have explored the capability of LLMs in code clone detection across multiple scenarios, including zero-shot learning [191], few-shot learning [192], and fine-tuning techniques [193]. While early empirical studies highlight the potential benefits of LLMs in code similarity tasks, they continue to face challenges in detecting semantic clones, where they often underperform compared to state-of-the-art models specifically designed for this task. Furthermore, current research on LLMs for code similarity measurement remains largely confined to code clone detection and is still in the empirical evaluation stage, rather than being developed into an efficient, standardized framework for broader code similarity applications. Future research can explore the use of reinforcement learning [194] and chain-of-thought reasoning [195] to enhance the performance of LLMs in code similarity analysis.

Another notable trend in the domain of code similarity is the increasing reliance on GNNs, driven by their ability to model the structured nature of source code. GNNs leverage graph-based representations such as ASTs, Control CFGs, and DFGs, etc., to capture both syntactic and semantic relationships within code. The rise of GNN-based models can be attributed to their superior performance in capturing long-range dependencies and hierarchical structures in code. Early studies employed models such as GCNs [136], GATs [104], and GGNNs [91]. One important trend in GNNs is the explicit modeling of interactions between code fragments to enhance similarity detection accuracy. Traditional approaches often process each code fragment independently before computing similarity scores, which may fail to capture fine-grained relationships between code structures. Recent studies, such as those by Zhang et al. [112] and Wang et al. [91], have addressed this limitation by introducing interaction-based architectures that jointly process code pairs, allowing models to directly compare and align corresponding structures. These interaction-based methods have demonstrated superior performance over independent encoding approaches. The trend toward GNN-based architectures is also reflected in hybrid models, where GNNs are combined with other representations to improve scalability and generalization. However, scaling GNN models to large codebases remains a significant challenge due to the computational cost of processing large graph structures. Unlike traditional deep learning models, which operate on fixed-size inputs, GNNs must process irregular, often massive graph representations of code. As software projects grow, the size of these graphs increases exponentially, leading to higher memory consumption, longer inference times, and increased complexity in message passing across graph nodes.

Future research should focus on efficient graph representation, hierarchical graph pooling, and distributed GNN training techniques to handle large-scale codebases while preserving structural information. Efficient graph processing methods can be explored to enhance the effectiveness and scalability of code graph representations. For instance, reinforcement learning can be used automatically learn optimal graph feature selection strategies [196]. Graph representations in ASTs can benefit from caching mechanisms for efficient processing [197]. Additionally, hybrid models combining GNNs with Transformers or LLMs may provide a balance between structural awareness and contextual richness, improving both efficiency and accuracy in code similarity tasks.

2) Source Code Representations in ML Techniques (RQ3)

The effectiveness of ML techniques in code similarity tasks heavily depends on the choice of source code representation. The survey reveals that various representation techniques have been explored, each with its own advantages and limitations.

ASTs are the most widely used code representations. The AST is beneficial because it captures the hierarchical syntactic structure of source code while abstracting away unimportant lexical details. This structure is well-suited for various deep learning models, we observe the usage of ASTs in RNN, GNN, Tree-CNN, transformer. However, they also present several limitations, particularly in terms of non-generality and computational cost. ASTs are tightly coupled with the syntax of a specific programming language, making them difficult to generalize across multiple languages. Thus, it is still challenging to access AST-based representation in cross-language code similarity measurement. Moreover, ASTs can become highly complex for large programs, leading to deep and unbalanced trees. Traversing, processing, and training models on such large trees introduce high computational overhead, slow down training and inference times. This problem is particularly evident in real-world scenarios such as large-scale code clone detection, real-time code analysis, and software maintenance in enterprise environments.

Token-based representations remain a popular approach, especially in NLP-inspired ML models such as Transformer-based architectures (e.g., CodeBERT, GraphCodeBERT, and GPT). This representation is beneficial for generalization and low computational overhead because they eliminate the need for complex syntactic or structural parsing, allowing models to operate directly on raw source code. Moreover, as the fast development of pre-trained models and LLMs, token-based representations can be easily adopted into these architectures due to their natural compatibility with sequence-based learning, enables easy fine-tuning across various code similarity applications. However, token-based approaches often struggle with capturing deep structural and semantic information of codes. Since token-based methods treat code as a linear sequence rather than a hierarchical structure, they fail to capture relationships between different code components, such as control flow, data dependencies, and function calls.

Recent advancements explore integrating multiple representations to obtain a more comprehensive depiction of source code. AST-based representations serve as a foundational component in hybrid approaches, often combined with graph-based or token-based representations to improve performance [1], [4], [91]. This combination enhances performance by leveraging the complementary strengths of different representations, promising avenue for capturing hierarchical and structural dependencies in source code [198]. Despite the promising advancements in hybrid approaches, challenges remain in scalability and efficiency, the concerns still remains in the additional computational cost incurred by combining multiple code representations. Currently, no extensive evaluation study has been conducted to systematically measure the trade-offs between improved performance and increased computational cost. Most existing research focuses on accuracy improvements, overlooking the practical feasibility of deploying hybrid models at scale, particularly in real-time applications such as code search, clone detection, and automated refactoring. Future studies should focus on optimizing computational efficiency by developing lightweight hybrid models that minimize memory overhead and inference time while preserving the advantages of multi-representation learning.

C. EVALUATION DOMAIN

1) Datasets Employed in Each Code Similarity Application Domain (RQ4)

The use of datasets in machine learning-based code similarity research is critical for evaluating model performance and ensuring reproducibility. However, the current landscape of dataset availability presents several challenges, including dataset bias, limited applicability to real-world scenarios, and the dominance of a few well-known datasets.

Most public datasets are used in code clone detection area. In sing-language code clone detection, the most frequent used dataset for ML-based tool/algorithms is BigCloneBench (BCB) [42]. BCB was constructed by mining a large inter-project Java dataset (250 million lines of code) for methods that implement specific target functionalities, which is considered as a good benchmark because its large scale, independence from tools, manual validation and real-world benchmark [45]. However, some researches [199], [200] have suggested that BCB is not suitable for ML tools developing because it was primarily designed to measure recall and contains mostly true clone labels, leading to a class imbalance with few false labels, which can hinder effective ML training. To address this, researchers have assumed that all unlabeled method pairs are false clones, but this approach only captures obvious false examples and misses subtler ones. Additionally, BCB is not a ground truth dataset and should not be used to infer unlabeled clones. Models trained and tested on BCB may perform well on the benchmark but struggle to generalize to real-world semantic similarity detection tasks. The dataset also lacks task-specific benchmarks designed explicitly for ML training, limiting its usefulness for advancing robust ML-based clone detection tools. Moreover, other widely used datasets for single-language code clone detection, such as Google Code Jam (GCJ) [159] and OJClone [158], are significantly smaller in size compared to BCB, raising concerns about the practicality of the proposed methods in real-world applications. Additionally, our observations suggest that SOTA models vary across different datasets, further questioning the generalization of these approaches.

For cross-language code clone detection, the Atcoder dataset [130], is prevalent in cross-language code clone detection, which sourced from the competitive programming website AtCoder. It includes accepted solutions from only beginner contests in Java and Python to ensure correct and syntactically similar implementations. The main problem with this dataset is that the data from competitive programming problems may not accurately reflect real-world code. Specifically, the code fragments in competitive programming often have less meaningful variable names and are highly optimized for well-defined tasks, which might not generalize well to more diverse and less structured real-world programming scenarios. Additionally, other datasets such as those from CodeChef and XLCoST are also used in this domain, but their sporadic adoption has led to a shortage of comprehensive benchmarks in the field. Apart from that, the majority of studies in other tasks have utilized custom datasets, which are often not publicly available. This lack of standardized, accessible datasets presents significant barriers to the reproducibility and validation of research findings. The inability to access these datasets hampers the community's efforts to rigorously test and compare the efficacy of proposed methods under consistent conditions.

While public datasets exist for clone detection, other applications of code similarity measurement—such as vulnerability detection, plagiarism detection, and code change inspection—often rely on proprietary, non-public datasets. This lack of standardized benchmarks significantly hinders the reproducibility of studies and the ability to compare results across different models.

Researchers have proposed large-scale AI-based benchmarks for code analysis, such as CodeNet [201] and CodeSearchNet [202]. CodeNet contains 14 million code samples across 50 programming languages, making it a strong candidate for training ML models. CodeSearchNet provides real-world code snippets and metadata, which can be leveraged to build more generalized models. However, these datasets were not specifically designed for code similarity measurement, and their applicability remains to be validated. For instance, while CodeNet claims that its problem-submission relationship corresponds to type-4 similarity and can be used for code search and clone detection, direct application to code similarity tasks is challenging. The dataset mainly consists of competitive programming submissions with diverse coding styles and problem constraints, lacks explicit ground-truth labels for similarity beyond problem-submission relations, and contains redundant or near-duplicate code that requires extensive preprocessing. As a result, substantial adaptation is necessary before these datasets can serve as standardized benchmarks for code similarity tasks.

2) Evaluation Metrics Considered to Measure Performance of ML Techniques (RQ5)

The evaluation of ML techniques for code similarity measurement is crucial for assess model performance and guiding improvements. However, the evaluation metrics predominantly relies on standard classification metics such as accuracy, precision, recall, and F1-score, which also implies that the current applications are still limited in detection-based tasks such as code clone detection, vulnerability detection etc. Additionally, while these metrics provide a basic understanding of model performance, their effectiveness in real-world scenarios remains uncertain. In many studies, models are assessed using a single or small ser of datasets and metrics without considering the broader implications of their deployment.

These challenges highlight the need for more comprehensive evaluation frameworks. Incorporating additional measures such as semantic similarity (e.g., BLEU, ROUGE, and CodeBLEU), ranking-based metrics (e.g., MAP for recommendation tasks), and efficiency-oriented evaluations (e.g., inference time for real-time applications) would provide a more holistic assessment of ML models. Without such considerations, the real-world applicability of these models remains in question, as their performance under controlled experimental settings may not reflect their robustness and reliability in practical software engineering workflows.

3) Best Performing ML Techniques (RQ6)

The evaluation of machine learning techniques for code similarity detection shows significant performance variations across different applications. However, challenges remain in standardizing evaluation methodologies and making fair comparisons due to dataset diversity. While most studies focus on code clone detection, there is a need for more comparative analyses in applications like software vulnerability detection, plagiarism detection, and automated code review. Future research should establish benchmark datasets and explore diverse ML techniques to improve generalizability and robustness across different code similarity tasks.

VIII. THREATS TO VALIDITY

Several factors may affect the validity and completeness of this systematic review. One primary concern is selection bias, which arises from the process of article selection and search completeness. Despite the use of a well-structured search strategy, including predefined search strings and multiple digital libraries, some relevant studies may have been unintentionally excluded. The accuracy of data extraction and quality assessment is another concern. the authors performed the quality check using predefined criteria and verified the extracted data. Despite these efforts, there remains a possibility of human error or subjective interpretation affecting the quality of the dataset used in this review.

IX. CONCLUSION

This paper presents a systematic review based on 104 primary studies concerning the use of ML in code similarity measurement. Our review systematically addresses comprehensive research questions across various subareas of these studies to delineate the current state of development, including 1) the application of current ML techniques to code similarity measurement; 2) ML techniques utilized in these studies; 3) source code representations in ML-based similarity assessments; 4) datasets employed in ML research on code similarity; 5) evaluation metrics used for assessing ML models in this context; 6) comparative studies within the field; and 7) challenges faced in the application of ML to code similarity.

We identify critical limitations, such as the scarcity of diverse datasets, the high computational cost of existing models, and the limited focus on real-world applications. Future research should prioritize lightweight architectures, hybrid models combining structural and contextual representations, and energy-efficient ML techniques to enhance the usability of these methods in large-scale software projects. Moreover, while LLMs have demonstrated effectiveness in various software development tasks, their potential to enhance code similarity measurement—especially in terms of scalability and handling multiple languages—remains largely unexplored.

Our analysis encapsulates the current research landscape and the utilization of ML algorithms in measuring code similarity. It is anticipated that these findings will spur further advancements in the field of code similarity measurement, fostering the development and application of innovative ML techniques.

ACKNOWLEDGEMENT

This research was supported by Science Foundation Ireland under grant numbers 18/CRT/6223 (SFI Centre for Research Training in Artificial Intelligence), and 13/RC/2094/P_2 (Lero SFI Centre for Software). For the purpose of Open Access, the author has applied a CC BY public copyright license to any Author Accepted Manuscript version arising from this submission.

APPENDIX A ABBREVIATIONS

The following abbreviations and acronyms are used in this paper:

Abbreviation	Full Form
ML	Machine Learning
DL	Deep Learning
SLR	Systematic Literature Review
AST	Abstract Syntax Tree
CFG	Control Flow Graph
PDG	Program Dependency Graph
GNN	Graph Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
BiLSTM	Bidirectional Long Short-Term Memory
TCN	Temporal Convolutional Network
GCN	Graph Convolutional Network
GAT	Graph Attention Network
GMN	Graph Matching Network
GRU	Gated Recurrent Units
RvNN	Recursive Neural Networks
RAE	Recursive Autoencoders
TBCNN	Tree-Based Convolutional Neural Network
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
ANN	Artificial Neural Network
SVM	Support Vector Machine
kNN	k-Nearest Neighbors
TF-IDF	Term Frequency-Inverse Document Frequency
GED	Graph Edit Distance
GPT	Generative Pre-trained Transformer
ELMo	Embeddings from Language Models
CodeBERT	A Pretrained Language Model for Programming Languages
GraphCodeBERT	A Graph-based Code Representation Learning Model
UniXcoder	A Pretrained Code Representation Model
BigCloneBench	A Popular Benchmark Dataset for Code Clone Detection
OJClone	Open Judge Clone Dataset
MOSS	Measure of Software Similarity (Plagiarism Detection Tool)
AUC	Area Under the Curve
ACC	Accuracy
FPR	False Positive Rate
ST3	Strongly Type-3 Clones
VST3	Very-Strongly Type-3 Clones
MT3	Moderately Type-3 Clones
WT3/4	Weakly Type-3 or Type-4 Clones

TABLE 14. List of Abbreviations and Their Full Forms

APPENDIX B QUALITY ASSESSMENT SCORE

TABLE 15: Quality Assessment Scores for Selected Studies

Reference	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Score	Rating
[112]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[111]	Y	Y	Y	p	p	Y	Y	N	6	Good
[4]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[110]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[48]	Y	Y	Y	Y	Y	Y	Y	N	8	Excellent
[113]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[44]	Y	Y	Y	Y	Y	Y	Y	N	8	Excellent
[108]	Y	Y	Y	Y	P	Y	P	N	6	Good
[132]	Y	Y	P	Y	N	Y	P	N	5	Good
[109]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[3]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[2]	Y	Y	Y	N	N	P	P	N	3	Good
[1]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[106]	Y	Y	P	Y	Y	P	Y	Y	7	Excellent
[105]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[104]	Y	Y	Y	Y	Y	P	Y	Y	7.5	Excellent
[103]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[107]	Y	Y	Y	P	P	Y	Y	N	6	Good
[147]	Y	Y	P	P	P	P	Y	N	5	Good
[101]	Y	Y	Y	Y	P	P	Y	N	6	Good
[102]	Y	Y	Y	P	Y	Y	Y	Y	7.5	Excellent
[131]	Y	Y	Y	Y	Y	P	Y	N	6.5	Good
[100]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[142]	Y	Y	Y	Y	Y	P	Y	N	6.5	Good
[143]	Y	Y	Y	Y	P	P	Y	N	6	Good
[99]	Y	Y	Y	P	Y	P	Y	N	6	Good
[97]	Y	Y	Y	Y	P	Y	Y	Y	7.5	Excellent
[98]	Y	Y	Y	Y	P	Y	Y	Y	8	Excellent
[140]	Y	Y	N	P	Y	Y	Y	N	5.5	Good
[95]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[164]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[96]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[94]	Y	Y	P	P	Y	Y	Y	N	6	Good
[5]	Y	Y	Y	Y	P	Y	Y	N	6.5	Good
[93]	Y	Y	Y	P	Y	Y	Y	Y	7.5	Excellent
[49]	Y	Y	Y	N	P	Y	Y	N	5.5	Good
[148]	Y	Y	Y	Y	P	P	Y	N	6	Good
[37]	Y	Y	Y	Y	P	Y	Y	N	6.5	Good
[92]	Y	Y	P	Y	P	P	Y	N	5.5	Good
[85]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[47]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[86]	Y	Y	Y	Y	P	Y	Y	N	6.5	Good
[149]	Y	Y	Y	N	P	Y	Y	N	6	Good
[87]	Y	Y	Y	N	Y	Y	P	N	5.5	Good
[9]	Y	Y	Y	Y	Y	Y	P	Y	7.5	Excellent
[138]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[141]	Y	Y	Y	N	Y	Y	Y	N	6	Good
[88]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[46]	Y	Y	P	N	P	Y	Y	N	5	Good
[89]	Y	Y	Y	Y	P	Y	Y	N	6.5	Good

Continued on next page

(Continued from previous page)

Reference	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Score	Rating
[90]	Y	Y	Y	Y	P	Y	Y	Y	6.5	Excellent
[146]	Y	Y	Y	N	P	Y	P	N	5.5	Good
[144]	Y	Y	Y	Y	Y	Y	P	N	6.5	Good
[36]	Y	Y	Y	N	Y	Y	Y	N	6	Good
[150]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[91]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[84]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[151]	Y	Y	Y	N	Y	Y	Y	N	6	Good
[83]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[129]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[82]	Y	Y	Y	Y	P	Y	Y	Y	7.5	Excellent
[152]	Y	Y	Y	P	Y	Y	P	N	6	Good
[130]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[81]	Y	Y	Y	Y	P	Y	Y	N	6.5	Good
[137]	Y	Y	Y	Y	P	Y	Y	N	6.5	Good
[137]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[43]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[80]	Y	Y	P	N	P	Y	N	N	4	Good
[41]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[78]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[79]	Y	Y	Y	O	Y	Y	N	N	5.5	Good
[139]	Y	Y	Y	P	Y	Y	Y	N	6.5	Good
[40]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[77]	Y	Y	Y	P	Y	P	Y	N	6	Good
[145]	Y	Y	P	P	Y	P	Y	N	5.5	Good
[6]	Y	Y	P	P	Y	Y	Y	N	6	Good
[39]	Y	Y	Y	p	P	Y	P	N	6	Good
[75]	Y	Y	P	N	P	Y	N	N	4	Good
[76]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[73]	Y	Y	P	Y	N	Y	P	N	5	Good
[74]	Y	Y	P	P	N	P	N	N	4	Good
[35]	Y	Y	Y	P	Y	Y	Y	N	6.5	Good
[72]	Y	Y	Y	N	Y	Y	Y	N	6	Good
[34]	Y	Y	Y	N	Y	Y	Y	N	6	Good
[172]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[114]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[116]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[125]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[126]	Y	Y	Y	p	P	Y	P	N	6	Good
[115]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[117]	Y	Y	Y	P	Y	Y	P	N	6	Good
[134]	Y	Y	Y	P	Y	Y	Y	Y	7.5	Excellent
[165]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[121]	Y	Y	Y	P	Y	Y	Y	N	6.5	Good
[124]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[127]	Y	Y	Y	P	P	Y	Y	N	6	Good
[118]	Y	Y	Y	P	N	Y	N	N	4.5	Good
[119]	Y	Y	Y	N	N	Y	N	N	4	Good
[133]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent
[135]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[136]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent

Continued on next page

(Continued from previous page)

Reference	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Score	Rating
[123]	Y	Y	Y	Y	Y	Y	Y	N	7	Excellent
[176]	Y	Y	Y	Y	Y	Y	Y	Y	8	Excellent

REFERENCES

[1] J. Liu, J. Zeng, X. Wang, and Z. Liang, "Learning graph-based code representations for source-level functional similarity detection," in *International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 345–357.

[2] L. Dai, "A study on the application of graph neural network in code clone detection: Improving the performance of code clone detection through graph neural networks and attention mechanisms," in *International Conference on Networks, Communications and Information Technology (CNCIT)*. ACM, 2023, pp. 172–176.

[3] D. Yu, Q. Yang, X. Chen, J. Chen, and Y. Xu, "Graph-based code semantics learning for efficient semantic code clone detection," *Information and Software Technology*, vol. 156, p. 107130, 2023.

[4] N. Mehrotra, A. Sharma, A. Jindal, and R. Purandare, "Improving cross-language code clone detection via code representation learning and graph neural networks," *IEEE Transactions on Software Engineering*, 2023.

[5] M. A. Fokam and R. Ajoodha, "Influence of contrastive learning on source code plagiarism detection through recursive neural networks," in *International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*. IEEE, 2021, pp. 1–6.

[6] J. Yasaswi, S. Purini, and C. Jawahar, "Plagiarism detection in programming assignments using deep features," in *Asian Conference on Pattern Recognition (ACPR)*. IEEE, 2017, pp. 652–657.

[7] S. Parsa, M. Zakeri-Nasrabadi, M. Ekhtiarzadeh, and M. Ramezani, "Method name recommendation based on source code metrics," *Journal of Computer Languages*, vol. 74, p. 101177, 2023.

[8] S. Arshad, S. Abid, and S. Shamail, "Codebert for code clone detection: A replication study," in *International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 39–45.

[9] M. Hammad, Ö. Babur, H. Abdul Basit, and M. v. d. Brand, "Deepclone: modeling clones to generate code predictions," in *International Conference on Software and Systems Reuse (ICSR)*. Springer, 2020, pp. 135–151.

[10] W. Wen, C. Shen, X. Lu, Z. Li, H. Wang, R. Zhang, and N. Zhu, "Cross-project software defect prediction based on class code similarity," *IEEE Access*, vol. 10, pp. 105 485–105 495, 2022.

[11] N. Tao, A. Ventresque, and T. Saber, "Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2022, pp. 1–8.

[12] ——, "Many-objective grammar-guided genetic programming with code similarity measurement for program synthesis," in *Latin American Conference on Computational Intelligence (LA-CCI)*. IEEE, 2023, pp. 1–6.

[13] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[14] M. Kaur and D. Rattan, "A systematic literature review on the use of machine learning in code clone research," *Computer Science Review*, vol. 47, p. 100528, 2023.

[15] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Computational Intelligence Magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[16] K. Ethayarajh and D. Jurafsky, "Utility is in the eye of the user: A critique of nlp leaderboards," *arXiv preprint arXiv:2009.13888*, 2020.

[17] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems*, vol. 6, no. 4, pp. 1–19, 2015.

[18] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *International ACM SIGIR Conference on Research and Development in Information Retrieval (IR)*. ACM, 2019, pp. 165–174.

[19] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of field robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[20] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixao, F. Mutz *et al.*, "Self-driving cars: A survey," *Expert systems with applications*, vol. 165, p. 113816, 2021.

[21] H. Habehh and S. Gohel, "Machine learning in healthcare," *Current genomics*, vol. 22, no. 4, p. 291, 2021.

[22] S. Ahmed, M. M. Alshater, A. El Ammari, and H. Hammami, "Artificial intelligence and machine learning in finance: A bibliometric review," *Research in International Business and Finance*, vol. 61, p. 101646, 2022.

[23] F. H. Quradaa, S. Shahzad, and R. S. Almoqbily, "A systematic literature review on the applications of recurrent neural networks in code clone research," *Plos one*, vol. 19, no. 2, p. e0296858, 2024.

[24] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[25] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, vol. 204, p. 111796, 2023.

[26] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[27] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[28] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Working Conference on Reverse Engineering*. IEEE, 1995, pp. 86–95.

[29] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[30] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 147–156.

[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 96–105.

[32] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[33] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *International Conference on Software Maintenance (ICSM)*. IEEE, 1998, pp. 368–377.

[34] U. Bandara and G. Wijayarathna, "A machine learning based tool for source code plagiarism detection," *International Journal of Machine Learning and Computing*, vol. 1, no. 4, p. 337, 2011.

[35] G. Acampora and G. Cosma, "A fuzzy-based approach to programming language independent source-code plagiarism detection," in *International conference on fuzzy systems (FUZZ-IEEE)*. IEEE, 2015, pp. 1–8.

[36] Y. He, W. Wang, H. Sun, and Y. Zhang, "Vul-mirror: a few-shot learning method for discovering vulnerable code clone," *EAI Endorsed Transactions on Security and Safety*, vol. 7, no. 23, 2020.

[37] H. Sun, L. Cui, L. Li, Z. Ding, Z. Hao, J. Cui, and P. Liu, "Vdsimilar: Vulnerability detection based on code similarity of vulnerabilities and patches," *Computers & Security*, vol. 110, p. 102417, 2021.

[38] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 602–614.

[39] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2016, pp. 1024–1028.

[40] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[41] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)*. ACM, 2018, pp. 141–151.

[42] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[43] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 70–80.

[44] Y. Wu, S. Feng, W. Suo, D. Zou, and H. Jin, "Goner: Building tree-based n-gram-like model for semantic code clone detection," *IEEE Transactions on Reliability*, 2023.

[45] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2016, pp. 596–600.
[46] L. Zhang, Z. Feng, W. Ren, and H. Luo, "Siamese-based bilstm network for scratch source code similarity measuring," in *International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2020, pp. 1800–1805.
[47] A. Nair, A. Roy, and K. Meinke, "funcgnn: A graph neural network approach to program similarity," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2020, pp. 1–11.
[48] G. Boldini, A. Diana, V. Arceri, V. Bonnici, and R. Bagnara, "A machine learning approach for source code similarity via graph-focused features," in *International Conference on Machine Learning, Optimization, and Data Science*. Springer, 2023, pp. 53–67.
[49] Y. Wu and W. Wang, "Code similarity detection based on siamese network," in *International Conference on Information Communication and Software Engineering (ICICSE)*. IEEE, 2021, pp. 47–51.
[50] M. Soori, B. Arezoo, and R. Dastres, "Artificial intelligence, machine learning and deep learning in advanced robotics, a review," *Cognitive Robotics*, 2023.
[51] D. Bishara, Y. Xie, W. K. Liu, and S. Li, "A state-of-the-art review on machine learning-based multiscale modeling, simulation, homogenization and design of materials," *Archives of computational methods in engineering*, vol. 30, no. 1, pp. 191–222, 2023.
[52] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data*. AMLBook New York, 2012, vol. 4.
[53] G. James, D. Witten, T. Hastie, R. Tibshirani *et al.*, *An introduction to statistical learning*. Springer, 2013, vol. 112.
[54] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2016, pp. 785–794.
[55] J. Oyelade, I. Isewon, O. Oladipupo, O. Emebo, Z. Omogbadegun, O. Aromolaran, E. Uwoghiren, D. Olaniyan, and O. Olawole, "Data clustering: Algorithms and its applications," in *International Conference on Computational Science and Its Applications (ICCSA)*. IEEE, 2019, pp. 71–81.
[56] S. Ayesha, M. K. Hanif, and R. Talib, "Overview and comparative study of dimensionality reduction techniques for high dimensional data," *Information Fusion*, vol. 59, pp. 44–58, 2020.
[57] X. Liu, F. Zhang, Z. Hou, L. Mian, Z. Wang, J. Zhang, and J. Tang, "Self-supervised learning: Generative or contrastive," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 857–876, 2021.
[58] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, pp. 1–40, 2016.
[59] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, "Deep learning application on code clone detection: A review of current knowledge," *Journal of Systems and Software*, vol. 184, p. 111141, 2022.
[60] C.-F. Chen, A. M. Zain, and K.-Q. Zhou, "Definition, approaches, and analysis of code duplication detection (2006–2020): a critical review," *Neural Computing and Applications*, vol. 34, no. 23, pp. 20 507–20 537, 2022.
[61] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.
[62] O. Karnalim, W. Chivers *et al.*, "Similarity detection techniques for academic source code plagiarism and collusion: a review," in *International Conference on Engineering, Technology and Education (TALE)*. IEEE, 2019, pp. 1–8.
[63] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: a systematic review," *ACM Transactions on Computing Education*, vol. 19, no. 3, pp. 1–37, 2019.
[64] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.
[65] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Technical Report 2007-541*, vol. 541, no. 115, pp. 64–68, 2007.
[66] E. Choi, N. Fuke, Y. Fujiwara, N. Yoshida, and K. Inoue, "Investigating the generalizability of deep learning-based clone detectors," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 181–185.
[67] W. Huang, G. Meng, C. Lin, Q. Yan, K. Chen, and Z. Ma, "Are our clone detectors good enough? an empirical study of code effects by obfuscation," *Cybersecurity*, vol. 6, no. 1, p. 14, 2023.
[68] A. Schäfer, W. Amme, and T. S. Heinze, "Experiments on code clone detection and machine learning," in *International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 46–52.
[69] S. M. Rabbani, N. A. Gulzar, S. Arshad, S. Abid, and S. Shamail, "A comparative analysis of clone detection techniques on semanticclonebench," in *International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 16–22.
[70] A. Capiluppi, D. Di Ruscio, J. Di Rocco, P. T. Nguyen, and N. Ajienka, "Detecting java software similarities by using different clustering techniques," *Information and Software Technology*, vol. 122, p. 106279, 2020.
[71] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *International conference on evaluation and assessment in software engineering (EASE)*. ACM, 2014, pp. 1–10.
[72] C. Li, J. Sun, and H. Chen, "An improved method for tree-based clone detection in web applications," in *International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*. IEEE, 2014, pp. 363–367.
[73] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 201–210.
[74] B. Joshi, P. Budhathoki, W. L. Woon, and D. Svetinovic, "Software clone detection using clustering approach," in *International Conference on Neural Information Processing (ICONIP)*. Springer, 2015, pp. 520–527.
[75] S. Jadon, "Code clones detection using machine learning technique: Support vector machine," in *International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2016, pp. 399–303.
[76] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *IEEE/ACM international conference on automated software engineering (ASE)*. ACM/IEEE, 2016, pp. 87–98.
[77] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *International Joint Conference on Artificial Intelligence*. IJCAI, 2017, pp. 3034–3040.
[78] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)*. ACM, 2018, pp. 354–365.
[79] A. Sheneamer, S. Roy, and J. Kalita, "A detection framework for semantic code clones and obfuscated code," *Expert Systems with Applications*, vol. 97, pp. 405–420, 2018.
[80] C. Wang, J. Gao, Y. Jiang, Z. Xing, H. Zhang, W. Yin, M. Gu, and J. Sun, "Go-clone: graph-embedding based clone detector for golang," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 374–377.
[81] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 95–104.
[82] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.
[83] J. Zeng, K. Ben, X. Li, and X. Zhang, "Fast code clone detection based on weighted recursive autoencoders," *IEEE Access*, vol. 7, pp. 125 062–125 078, 2019.
[84] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.
[85] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 4, pp. 1–23, 2020.
[86] Y. Yuan, W. Kong, G. Hou, Y. Hu, M. Watanabe, and A. Fukuda, "From local to global semantic clone detection," in *International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2020, pp. 13–24.
[87] H. Xue, Y. Mei, K. Gogineni, G. Venkataramani, and T. Lan, "Twin-finder: Integrated reasoning engine for pointer-related code clone detection," in *International Workshop on Software Clones (IWSC)*. IEEE, 2020, pp. 1–7.

[88] S. Khurshid, C. S. Păsăreanu, C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 516–527, 2020.

[89] C. Feng, T. Wang, Y. Yu, Y. Zhang, Y. Zhang, and H. Wang, "Sia-rae: a siamese network based on recursive autoencoder for effective clone detection," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 238–246.

[90] B. Li, C. Ye, S. Guan, and H. Zhou, "Semantic code clone detection via event embedding tree and gat network," in *International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 382–393.

[91] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[92] Z. Lu, R. Li, H. Hu, and W.-a. Zhou, "A code clone detection algorithm based on graph convolution network with ast tree edge," in *International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2021, pp. 1027–1032.

[93] A. Zhang, K. Liu, L. Fang, Q. Liu, X. Yun, and S. Ji, "Learn to align: a code alignment network for code clone detection," in *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 1–11.

[94] K. Xu and Y. Liu, "Sccd-gan: An enhanced semantic code clone detection model using gan," in *International Conference on Electronics and Communication Engineering (ICECE)*. IEEE, 2021, pp. 16–22.

[95] H. Liang and L. Ai, "Ast-path based compare-aggregate network for code clone detection," in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[96] A. Sheneamer, S. Roy, and J. Kalita, "An effective semantic code clone detection framework using pairwise feature fusion," *IEEE Access*, vol. 9, pp. 84 828–84 844, 2021.

[97] Y. Wu, S. Feng, D. Zou, and H. Jin, "Detecting semantic code clones by building ast-based markov chains model," in *International Conference on Automated Software Engineering*. IEEE, 2022, pp. 1–13.

[98] Y. Hu, D. Zou, J. Peng, Y. Wu, J. Shan, and H. Jin, "Treecen: Building tree graph for scalable semantic code clone detection," in *International Conference on Automated Software Engineering*. IEEE, 2022, pp. 1–12.

[99] S. Karthik and B. Rajdeepa, "A collaborative method for code clone detection using a deep learning model," *Advances in Engineering Software*, vol. 174, p. 103327, 2022.

[100] W. Hua and G. Liu, "Transformer-based networks over tree structures for code classification," *Applied Intelligence*, vol. 52, no. 8, pp. 8895–8909, 2022.

[101] H. Liu, H. Zhao, C. Han, and L. Hou, "Low-complexity code clone detection using graph-based neural networks," in *International Conference on Mobility, Sensing and Networking (MSN)*. IEEE, 2022, pp. 797–802.

[102] S. Patel and R. Sinha, "Combining holistic source code representation with siamese neural networks for detecting code clones," in *International Conference on Testing Software and Systems (IFIP)*. Springer, 2021, pp. 148–159.

[103] D. Xiao, D. Hang, L. Ai, S. Li, and H. Liang, "Path context augmented statement and network for learning programs," *Empirical Software Engineering*, vol. 27, pp. 1–26, 2022.

[104] Y. Jiang, X. Su, C. Treude, and T. Wang, "Hierarchical semantic-aware neural code representation," *Journal of Systems and Software*, vol. 191, p. 111355, 2022.

[105] X. Wang, Q. Wu, H. Zhang, C. Lyu, X. Jiang, Z. Zheng, L. Lyu, and S. Hu, "Heloc: Hierarchical contrastive learning of source code representation," in *International Conference on Program Comprehension*. IEEE, 2022, pp. 354–365.

[106] Z. Xue, Z. Jiang, C. Huang, R. Xu, X. Huang, and L. Hu, "Seed: Semantic graph based deep detection for type-4 clone," in *International Conference on Software and Software Reuse (ICSR)*. Springer, 2022, pp. 120–137.

[107] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3771–3789, 2021.

[108] X. Wu, W. Zhao, Z. Tan, X. Zhang, and W. Chen, "Research and implementation of code similarity detection technology based on deep learning," in *International Conference on Cloud Computing and Intelligent Systems (CCIS)*. IEEE, 2023, pp. 235–239.

[109] X. Zhang, J. Liu, and M. Shi, "A parallel deep learning-based code clone detection model," *Journal of Parallel and Distributed Computing*, vol. 181, p. 104747, 2023.

[110] B. Wan, S. Dong, J. Zhou, and Y. Qian, "Sjbcd: A java code clone detection method based on bytecode using siamese neural network," *Applied Sciences*, vol. 13, no. 17, p. 9580, 2023.

[111] H. Yang, Z. Li, and X. Guo, "A novel source code clone detection method based on dual-gcn and ivhfs," *Electronics*, vol. 12, no. 6, p. 1315, 2023.

[112] A. Zhang, L. Fang, C. Ge, P. Li, and Z. Liu, "Efficient transformer with code token learner for code clone detection," *Journal of Systems and Software*, vol. 197, p. 111557, 2023.

[113] Y. Li, C. Yu, and Y. Cui, "Tpcaps: a framework for code clone detection and localization based on improved capsnet," *Applied Intelligence*, vol. 53, no. 13, pp. 16 594–16 605, 2023.

[114] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[115] Z. Zhao, B. Yang, G. Li, H. Liu, and Z. Jin, "Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks," *Journal of Systems and Software*, vol. 184, p. 111108, 2022.

[116] P. Keller, A. K. Kaboré, L. Plein, J. Klein, Y. Le Traon, and T. F. Bissyande, "What you see is what it means! semantic representation learning of code based on visualization and transfer learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–34, 2021.

[117] R. Sharma, F. Chen, F. Fard, and D. Lo, "An exploratory study on code attention in bert," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 437–448.

[118] S. Abid, X. Cai, and L. Jiang, "Interpreting codebert for semantic code clone detection," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 229–238.

[119] P. Wang, L. Zhu, Q. Wang, O. Jaiteh, and C. Guo, "An empirical understanding of code clone detection by chatgpt," in *2023 6th International Conference on Data Science and Information Technology (DSIT)*. IEEE, 2023, pp. 78–83.

[120] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcode{bert}: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[121] M. Chochlov, G. A. Ahmed, J. V. Patten, G. Lu, W. Hou, D. Gregg, and J. Buckley, "Using a nearest-neighbour, bert-based approach for scalable clone detection," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 582–591.

[122] Y. Zhang, R. Wu, J. Liao, and L. Chen, "Structural adversarial attack for code representation models," in *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2023, pp. 392–413.

[123] Z. Xu, M. Zhou, X. Zhao, Y. Chen, X. Cheng, and H. Zhang, "xastnn: Improved code representations for industrial practice," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1727–1738.

[124] Z. Lin, G. Li, J. Zhang, Y. Deng, X. Zeng, Y. Zhang, and Y. Wan, "Xcode: Towards cross-language code representation with large-scale pre-training," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–44, 2022.

[125] Y. Ding, L. Buratti, S. Chakraborty, S. Pujar, A. Morari, and B. Ray, "Contrastive learning for source code with structural and functional properties," 2022. [Online]. Available: https://openreview.net/forum?id=7KgeqhkbZab

[126] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," 2021. [Online]. Available: https://openreview.net/forum?id=uV7hcsjqM-

[127] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Towards learning (dis)-similarity of source code from program contrasts," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 6300–6312. [Online]. Available: https://aclanthology.org/2022.acl-long.436

[128] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499

[129] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1026–1037.

[130] D. Perez and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 518–528.

[131] H. Ling, A. Zhang, C. Yin, D. Li, and M. Chang, "Improve representation for cross-language clone detection by pretrain using tree autoencoder," *Intelligent Automation & Soft Computing*, vol. 33, no. 3, pp. 1561–1577, 2022.

[132] M. A. Yahya and D.-K. Kim, "Clcd-i: cross-language clone detection by using deep learning with infercode," *Computers*, vol. 12, no. 1, p. 12, 2023.

[133] J. Li, C. Tao, Z. Jin, F. Liu, and G. Li, "Zc 3: Zero-shot cross-language code clone detection," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 875–887.

[134] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive cross-language code clone detection," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 413–424.

[135] Y. Fang, F. Zhou, Y. Xu, and Z. Liu, "Tcccd: Triplet-based cross-language code clone detection," *Applied Sciences*, vol. 13, no. 21, p. 12084, 2023.

[136] Z. Swilam, A. Hamdy, and A. Pester, "Cross-language code clone detection using abstract syntax tree and graph neural network," in *2023 International Conference on Computer and Applications (ICCA)*. IEEE, 2023, pp. 1–5.

[137] G. Mostaeen, J. Svajlenko, B. Roy, C. K. Roy, and K. A. Schneider, "Clonecognition: machine learning based code clone validation tool," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 1105–1109.

[138] G. Mostaeen, B. Roy, C. K. Roy, K. Schneider, and J. Svajlenko, "A machine learning based framework for code clone validation," *Journal of Systems and Software*, vol. 169, p. 110686, 2020.

[139] A. Sheneamer, H. Hazazi, S. Roy, and J. Kalita, "Schemes for labeling semantic code clones using machine learning," in *International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 981–985.

[140] F. Ullah, M. R. Naeem, L. Mostarda, and S. A. Shah, "Clone detection in 5g-enabled social iot system using graph semantics and deep learning model," *International Journal of Machine Learning and Cybernetics*, vol. 12, pp. 3115–3127, 2021.

[141] K. W. Nafi, B. Roy, C. K. Roy, and K. A. Schneider, "A universal cross language software similarity detector for open source software categorization," *Journal of Systems and Software*, vol. 162, p. 110491, 2020.

[142] F. Ullah, M. R. Naeem, H. Naeem, X. Cheng, and M. Alazab, "Crolssim: Cross-language software similarity detector using hybrid approach of lsa-based ast-mdrep features and cnn-lstm model," *International Journal of Intelligent Systems*, vol. 37, no. 9, pp. 5768–5795, 2022.

[143] S. Karakatič, A. Miloševič, and T. Heričko, "Software system comparison with semantic source code embeddings," *Empirical Software Engineering*, vol. 27, no. 3, p. 70, 2022.

[144] C. Xie, X. Wang, C. Qian, and M. Wang, "A source code similarity based on siamese neural network," *Applied Sciences*, vol. 10, no. 21, p. 7519, 2020.

[145] J. Yasaswi, S. Kailash, A. Chilupuri, S. Purini, and C. Jawahar, "Unsupervised learning based approach for plagiarism detection in programming assignments," in *Innovations in Software Engineering Conference (ISEC)*. ACM, 2017, pp. 117–121.

[146] N. Awale, M. Pandey, A. Dulal, and B. Timsina, "Plagiarism detection in programming assignments using machine learning," *Journal of artificial intelligence and capsule networks*, vol. 2, no. 3, pp. 177–184, 2020.

[147] D. Bernhauer, "Code visualization for plagiarism detection," in *International Conference on Social Networks Analysis, Management and Security (SNAMS)*. IEEE, 2022, pp. 1–6.

[148] F. Ullah, J. Wang, M. Farhan, M. Habib, and S. Khalid, "Software plagiarism detection in multiprogramming languages using machine learning approach," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 4, p. e5000, 2021.

[149] K. T. Ayinala, K. S. Cheng, K. Oh, T. Song, and M. Song, "Code inspection support for recurring changes with deep learning in evolving software," in *Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 931–942.

[150] C. Guo, H. Yang, D. Huang, J. Zhang, N. Dong, J. Xu, and J. Zhu, "Review sharing via deep semi-supervised code clone detection," *IEEE Access*, vol. 8, pp. 24 948–24 965, 2020.

[151] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[152] R. Xie, L. Chen, W. Ye, Z. Li, T. Hu, D. Du, and S. Zhang, "Deeplink: A code knowledge graph based deep learning approach for issue-commit link recovery," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 434–444.

[153] N. D. Bui, Y. Yu, and L. Jiang, "Bilateral dependency neural networks for cross-language algorithm classification," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 422–433.

[154] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Knowledge Discovery and Data Mining (KDD)*, vol. 96, no. 34. AAAI, 1996, pp. 226–231.

[155] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *International conference on program comprehension*. IEEE, 2008, pp. 172–181.

[156] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *International conference on software engineering*. ACM, 2016, pp. 1157–1168.

[157] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[158] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI conference on artificial intelligence*, vol. 30, no. 1. AAAI, 2016.

[159] 2016, google Code Jam https://code.google.com/codejam/contests.html.

[160] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C. K. Roy, "Lvmapper: A large-variance clone detector using sequencing alignment approach," *IEEE access*, vol. 8, pp. 27 986–27 997, 2020.

[161] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[162] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[163] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, vol. 26. Curran Associates, Inc., 2013.

[164] Y.-B. Jo, J. Lee, and C.-J. Yoo, "Two-pass technique for clone detection and type classification using tree-based convolution neural network," *Applied Sciences*, vol. 11, no. 14, p. 6613, 2021.

[165] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[166] F. Al-Omari, C. K. Roy, and T. Chen, "Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020, pp. 57–63.

[167] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, vol. 1035, 2021.

[168] V. Ljubovic, "Programming homework dataset for plagiarism detection," 2020. [Online]. Available: https://dx.doi.org/10.21227/71fw-ss32

[169] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *International symposium on software testing and analysis (ISSTA)*. ACM, 2014, pp. 437–440.

[170] M. Kamp, P. Kreutzer, and M. Philippsen, "Sesame: A data set of semantically similar java methods," in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 529–533.

[171] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *International Conference on Software Engineering Companion (ICSE)*. IEEE, 2017, pp. 27–30.

[172] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[173] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *International Conference on Mining Software Repositories (ICSE)*. ACM, 2018, pp. 542–553.

[174] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[175] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: Software functional clone detection based on semantic tokens analysis," in *International conference on automated software engineering (ASE)*. IEEE, 2020, pp. 821–833.

[176] 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.

[177] T. Vislavski, G. Rakić, N. Cardozo, and Z. Budimac, "Licca: A tool for cross-language clone detection," in *International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 512–516.

[178] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Clcminer: detecting cross-language clones without intermediates," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 2, pp. 273–284, 2017.

[179] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism detection using feature-based neural networks," in *Technical symposium on Computer science education (SIGCSE)*. ACM, 2007, pp. 34–38.

[180] S. Bougleux, L. Brun, V. Carletti, P. Foggia, B. Gaüzère, and M. Vento, "Graph edit distance as a quadratic assignment problem," *Pattern Recognition Letters*, vol. 87, pp. 38–46, 2017.

[181] S. Fankhauser, K. Riesen, and H. Bunke, "Speeding up graph edit distance computation through fast bipartite matching," in *International Workshop on Graph-Based Representations in Pattern Recognition (IAPR)*. Springer, 2011, pp. 102–111.

[182] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image and Vision computing*, vol. 27, no. 7, pp. 950–959, 2009.

[183] A. Fischer, K. Riesen, and H. Bunke, "Improved quadratic time approximation of graph edit distance by combining hausdorff matching and greedy assignment," *Pattern Recognition Letters*, vol. 87, pp. 55–62, 2017.

[184] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[185] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[186] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[187] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Simgnn: A neural network approach to fast graph similarity computation," in *Proceedings of the twelfth ACM international conference on web search and data mining*, 2019, pp. 384–392.

[188] P. Mohseni, N. Duffield, B. Mallick, and A. Hasanzadeh, "Adaptive conditional quantile neural processes," in *Uncertainty in Artificial Intelligence*. PMLR, 2023, pp. 1445–1455.

[189] M. Baradaran, A. Ansari, M. Sadrosadati, and H. Sarbazi-Azad, "Energy consumption analysis of instruction cache prefetching methods," in *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 2023, pp. 60–67.

[190] K. Chadli, G. Botterweck, and T. Saber, "The environmental cost of engineering machine learning-enabled systems: A mapping study," in *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024, pp. 200–207.

[191] Z. Zhang and T. Saber, "Assessing the code clone detection capability of large language models," in *2024 4th International Conference on Code Quality (ICCQ)*. IEEE, 2024, pp. 75–83.

[192] M. Khajezade, J. J. Wu, F. H. Fard, G. Rodríguez-Pérez, and M. S. Shehata, "Investigating the efficacy of large language models for code clone detection," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 161–165.

[193] R. Inoue and Y. Higo, "Improving accuracy of llm-based code clone detection u sing functionally equivalent methods," in *2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 2024, pp. 24–27.

[194] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[195] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[196] H. Imani, J. Anderson, S. Farid, A. Amirany, and T. El-Ghazawi, "Rlfl: A reinforcement learning aggregation approach for hybrid federated learning systems using full and ternary precision," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2024.

[197] A. T. Mughrabi, M. Baradaran, A. Samara, and K. Skadron, "Ecg: Expressing locality and prefetching for optimal caching in graph structures," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 520–525.

[198] P. Mohseni and N. Duffield, "Spectral convolutional conditional neural processes," *arXiv preprint arXiv:2404.13182*, 2024.

[199] J. Krinke and C. Ragkhitwetsagul, "Bigclonebench considered harmful for machine learning," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 1–7.

[200] J. Svajlenko and C. K. Roy, "Bigclonebench: A retrospective and roadmap," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 8–9.

[201] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[202] 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.

ZIXIAN ZHANG is a PhD student in the SFI Centre for Research Training in Artificial Intelligence (CRT-AI) the School of Computer Science at University of Galway, Ireland, under the supervision of Dr Takfarinas Saber. His main research topic is on the design and application of novel machine learning techniques for the assessment and measurement of code similarity.

TAKFARINAS SABER holds a PhD (2017) in Computer Science from University College Dublin, Ireland. He is currently a Lecturer in Computer Science at the University of Galway, Ireland. His area of expertise is in the optimisation of Complex Software Systems. He designs and applies novel Artificial Intelligence techniques from Operations Research, Machine Learning, and Evolutionary Computation/Learning to advance the Engineering and Testing of systems such as Smart Cities, Distributed Systems, and Wireless Communication Networks.