Machine Learning Approaches to Code Similarity Measurement: A Systematic Review

Mrs.K.Gowri[1], Mr.E.Naveen[2], Mr.M.Vijay[3], Mr.M.Logeswaran[4]
[1]Assist prof, Dept of Artificial intelligence and data science
[2, 3, 4]Dept of Artificial intelligence and data science
[1, 2, 3, 4] Sir Issac Newton College Of Engineering and Technology

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 analysed 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 utilised 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 synthesises existing research but also identifies prevailing limitations and challenges, shedding light on potential avenues for future work.

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 asks encompassing a wide array of applications including, but not limited to, code clone detection [1], [2], [3], [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 MachineLearning (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],[22], [23], [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 utilisation 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 analyse the data from the included studies. The contribution of our SLR can be summarised as follows: • We have aggregated and synthesised 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 utilised 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 analyses 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 VII discusses the challenges and limitations in this field. Section VIII includes threats to Validity in this study.

II. BACKGROUND RELATED WORKS

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], [28], [29]. This approach evaluates elements such as code formatting, variable names, the organisation 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 organisation 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 recognise 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]. 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 utilised classification of clone types is rooted in the definitions of syntactic and semantic similarities between code segments [1], [39], [40], [41], [42], [43], [44]. Clone types 1 through 3 are categorised 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 and Roy [45]. In this schema, clones are categorised 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 categorised 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 categorised 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 unlabelled 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 minimising 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 unlabelled 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], [60], [61], [62], [63], [64],[65]or offering a broader perspective on the topic [25]. The earliest relevant survey by Roy and Cordy [65] categories 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 emphasising 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 centre 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], [67], [68], [69], [70], but these studies are often constrained to specific datasets or focus on limited sub-problems.Despite this variety of research, there remains a gap corning 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 sys- thematically categorised 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 r e s e a r c h q u e s t i o n s . F u r t h e r m o r e , o u r 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 pre-selected 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 recognised for their extensive collections of scholarly works in computer science and related inter- disciplinary fields. The following electronic databases wereincluded 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 utilising 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. 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.

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.

C) SELECTION PROCESS

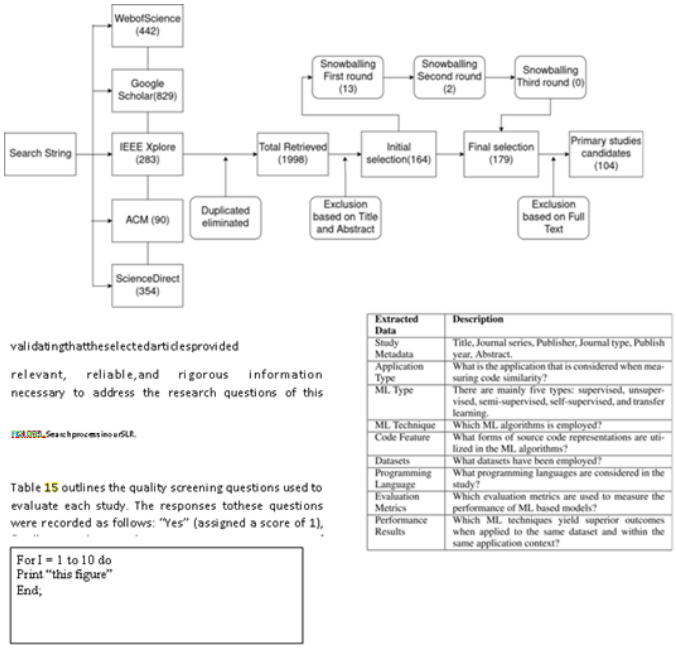
The selection process for this SLR, as illustrated in Figure 1, commenced with an initial search based on the sources and search string. This review encompasses studies published up to 2023, sourced from the initial dataset of the chosen digital libraries. As depicted in Figure 1, a total of 1998 articles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

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



validatingthattheselectedarticlesprovided

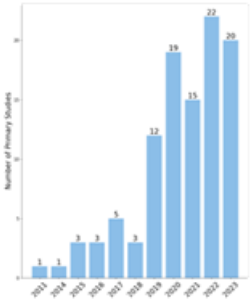
relevant, reliable,and rigorous information necessary to address the research questions of this

Table 15 outlines the quality screening questions used to evaluate each study. The responses tothese questions were recorded as follows: "Yes" (assigned a score of 1),

```
For I = 1 to 10 do
Print "this figure"
End;
```

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?

total quality score, each study was ranked according to the following classification:



Excellent: 7≤score ≤8
Good: 4 ≤score< 7
Poor: score < 3

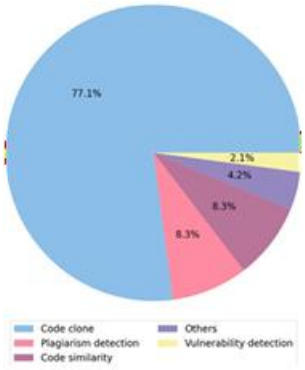
FIGURE 2. Number of primary studies in different years.

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.

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



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 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 categorise these studies according to their particular application focus. Moreover, Figure 4 presents the trajectory of ML utilisation across

various applications related to code similarity. Owing to the fact that some applications are only represented by one primary study, to improve the visualisation, 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 utilise 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 utilisation 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:** 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 sorcerer [156], this method improve

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]

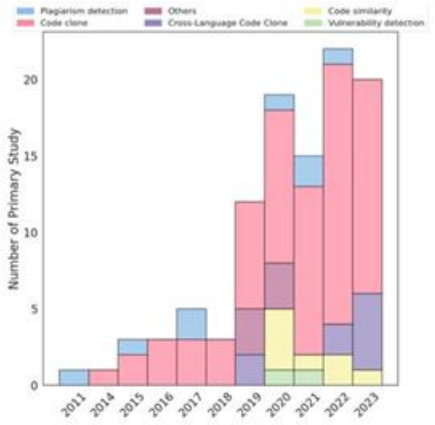


FIGURE 4. Applications of code similarity measurement over years.

Li et al. [40] introduced CCLearner: a tool that leverages deep neural networks to detect code clones. Cleaner 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 byte code, including byte code 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 obfuscate-tion 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 utilising 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 and Liu [94] introduce SCCD-GAN, an advanced model for detecting semantic code clones, utilising Graph Attention Networks [157] to analyse 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 detection task, showing competitive performance with other SOTA methods. Karthik and Rajdeep [99] introduce a Collaborative Code Clone Detection using a Deep Learning (CCCD- DL) model that utilises 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 utilises 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. Utilising a tailored GNN model, TAILOR outperforms existing methods in code clone detection and source code classification tasks, demonstrated by i ts 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 analyses 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 mod- els 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 Auto encoder (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 utilised 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 VALIDATON

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 endeavour 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 labelling 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 recognise 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 optimisation of software development practices and the maintenance of intellectual property integrity. Nafi 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. 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 utilising 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, utilising 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], utilising 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 utilises 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 re associate extracted features with segments of the original source code, thereby enhancing explainability. Both supervised and unsupervised ML algorithms are used to analyse 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 neighbours, 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 utilises static features extracted from the intermediate representation of programs during compilation. By employing unsupervised learning techniques, the system c lusters 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 plagiarised 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 utilisation 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. Sub- sequently, 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 unauthorised 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 pre-defined 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 utilises 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 a n i m p r e s s i v e a c c u r a c y r a t e o f 9 5 . 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 analyse 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 DETECTION

Ayinala et al. [149] propose RIDL, a novel approach for inspecting recurring code changes using DL. The method focuses on identifying and summarising systematic changes by learning patterns from code clones across multiple software versions. Utilising a trained CNN model on a dataset of 13,940 clones, RIDL demonstrates high efficiency in summarising 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. Utilising Gated Recurrent Units (GRU) and Generative Pre trained 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 emphasises 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 utilises deep semi-supervised learning for code similarity assessment. Their methodology integrates an auto-encoder-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 REPAIR

White et al. [151] explore the utilisation 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 prioritise 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 auto encoders 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 utilises Bilateral Neural Networks (Bi-NN) that encode both syntactic and semantic features of code from two different programming languages to recognise and classify algorithms implemented across these languages. The framework is evaluated on a dataset comprising thousands of Java and C +programs, demon-starting 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 0f 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.

EVALUATION METRICS CONSIDER TO PERFORMANCE OF THE ML TECHNIQUES

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 utilised across the reviewed studies, categorised into classification, regression, and time metrics. A detailed description of each of these metrics follows:

1. CLASSIFICARION METRICS

Accuracy (Acc)

Several primary studies reviewed herein utilise 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:

$$Acc = TP + TN /TP + TN + FP + FN$$

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:

$$Prec = TP/TP + FP$$

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:

$$Rec = TP/TP + FN$$

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:

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

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:

$$FNR = FN /TP + FN$$

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:

$$FPR = FP/FP + TN$$

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:

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

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:

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

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:

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

where:

n is the number of observations in the dataset,

- yi is the actual value of the i-th observation, yi is the predicted value for the i-th observation,
- |·|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:

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

where:

- n is the number of observations,
- yi is the actual value for the i-th observation,
  -yi is the predicted value for the i-th observation.

THRETS 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.

V. 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 utilised 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 prioritise 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 utilisation 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.

REFERENCE

[1] J. Liu, J. Zeng, X. Wang, and Z.Liang, ‘‘Learning graph-based code representations for source-level functional similarity detection,’’ in Proc. Int. Conf.Softw. Eng. (ICSE), Jan. 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 Proc. Int. Conf.Netw., Commun. Inf. Technol. (CNCIT), 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,’’ Inf. Softw.Technol., vol. 156, Apr. 2023, Art. no. 107130.

[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 Trans. Softw. Eng., vol. 49, no. 11, pp. 4846–4868, Nov. 2023.

[5] M. A. Fokam and R. Ajoodha, ‘‘Influence of contrastive learning onsource code plagiarism detection through recursive neural networks,’’in Proc. 3rd Int. Multidisciplinary Inf. Technol. Eng. Conf. (IMITEC),Nov. 2021, pp. 1–6.

[6] J. Yasaswi, S. Purini, and C. V. Jawahar, ‘‘Plagiarism detection in programming assignments using deep features,’’ in Proc. Asian Conf.Pattern Recognit. (ACPR), Nov. 2017, pp. 652–657.

[7] S. Parsa, M. Zakeri-Nasrabadi, M. Ekhtiarzadeh, and M. Ramezani,‘‘Method name recommendation based on source code metrics,’’ J.Comput. Lang., vol. 74, Jan. 2023, Art. no. 101177.

[8] S. Arshad, S. Abid, and S. Shamail, ‘‘CodeBERT for code clone detection:A replication study,’’ in Proc. IEEE 16th Int. Workshop Softw. Clones(IWSC), Oct. 2022, pp. 39–45.

[9] M. Hammad, Ö. Babur, H. A. Basit, and M. V. D. Brand, ‘‘DeepClone:Modeling clones to generate code predictions,’’ in Proc. Int. Conf. Softw.Syst. Reuse (ICSR), Jan. 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. 105485–105495, 2022.

[11] N. Tao, A. Ventresque, and T. Saber, ‘‘Multi-objective grammar-guidedgenetic programming with code

similarity measurement for programsynthesis,'' in Proc. IEEE Congr. Evol. Comput. (CEC), Jul. 2022,pp. 1–8.

[12] N. Tao, A. Ventresque, and T. Saber, ''Many-objective grammar-guidedgenetic programming with code similarity measurement for programsynthesis,'' in Proc. IEEE Latin Amer. Conf. Comput. Intell. (LA-CCI),Oct. 2023, pp. 1–6.

[13] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, ''Machine learningtechniques for code smell detection: A systematic literature review and meta-analysis,'' Inf. Softw. Technol., vol. 108, pp. 115–138, Apr. 2019.

[14] M. Kaur and D. Rattan, ''A systematic literature review on the use of machine learning in code clone research,'' Comput. Sci. Rev., vol. 47, Feb. 2023, Art. no. 100528.

[15] T. Young, D. Hazarika, S. Poria, and E. Cambria, ''Recent trends in deep learning based natural language processing,'' IEEE Comput. Intell. Mag., vol. 13, no. 3, pp. 55–75, Mar. 2018.

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

[17] C. A. Gomez-Uribe and N. Hunt, ''The Netflix recommender system: Algorithms, business Value, and innovation,'' ACM Trans. Manage. Inf. Syst., vol. 6, no. 4, pp. 1–19, Jan. 2016.

[18] X. Wang, X. He, M. Wang, F. Feng, and T. Chua, ''Neural graph collaborative filtering,'' in Proc. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr. (IR), Jul. 2019, pp. 165–174.

[19] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, ''A survey of deep learning techniques for autonomous driving,'' J. Field Robot., vol. 37, no. 3, pp. 362–386, Apr. 2020.

[20] C. Badué, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. F. Berriel, T. M. Paixão, F. Mutz, L. Veronese, T. Oliveira-Santos, and A. F. D. Souza, ''Self-driving cars: A survey,'' Expert Syst. Appl., vol. 165, Aug. 2020, Art. no. 113816.

[21] H. Habehh and S. Gohel, ''Machine learning in healthcare,'' Current Genomics, vol. 22, no. 4, pp. 291–300, Jul. 2021.

[22] S. Ahmed, M. M. Alshater, A. E. Ammari, and H. Hammami, ''Artificial intelligence and machine learning in finance: A bibliometric review,'' Res. Int. Bus. Finance, vol. 61, Oct. 2022, Art. no. 101646.

[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, Feb. 2024, Art. no. e0296858.

[24] F. Rosenblatt, ''The perceptron: A probabilistic model for information storage and organization in the brain,'' Psychol. Rev., vol. 65, no. 6, pp. 386–408, 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,'' J. Syst. Softw., vol. 204, Oct. 2023, Art. no. 111796.

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

[27] C. K. Roy, J. R. Cordy, and R. Koschke, ''Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,'' Sci. Comput. Program., vol. 74, no. 7, pp. 470–495, May 2009.

[28] B. S. Baker, ''On finding duplication and near-duplication in large software systems,'' in Proc. 2nd Work. Conf. Reverse Eng., Jul. 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 Trans. Softw. Eng., vol. 28, no. 7, pp. 654–670, Jul. 2002.

[30] M. Gabel and Z. Su, ''A study of the uniqueness of source code,'' in Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng., Nov. 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 Proc. 29th Int. Conf. Softw. Eng. (ICSE), May 2007, pp. 96–105.

[32] F.-H. Su, J. Bell, G. E. Kaiser, and S. Sethumadhavan, ''Identifying functionally similar code in complex codebases,'' in Proc. Int. Conf. Program Comprehension (ICPC), May 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 Proc. Int. Conf. Softw. Maintenance, Nov. 1998, pp. 368–377.

[34] U. Bandara and G. Wijayarathna, ''A machine learning based tool for source code plagiarism detection,'' Int. J. Mach. Learn. Comput., vol. 1, no. 4, pp. 337–343, 2011.

[35] G. Acampora and G. Cosma, ''A fuzzy-based approach to programming language independent source-code plagiarism detection,'' in Proc. IEEE Int. Conf. Fuzzy Syst. (FUZZ-IEEE), Aug. 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 Trans. Secur. Saf., vol. 7, no. 23, 2020, Art. no. 165275.