
Tushar Sharma¹, Maria Kechagia², Stefanos Georgiou³, Rohit Tiwari⁴, Indira Vats⁵, Hadi Moazen⁶, Federica Sarro²

¹Dalhousie University, Canada; ²University College London, United Kingdom; ³Queens University, Canada; ⁴DevOn, India; ⁵J.S.S. Academy of Technical Education, India; ⁶Sharif University of Technology, Iran

Abstract

The advancements in machine learning techniques have encouraged researchers to apply these techniques to a myriad of software engineering tasks that use source code analysis, such as testing and vulnerability detection. Such a large number of studies hinders the community from understanding the current research landscape. This paper aims to summarize the current knowledge in applied machine learning for source code analysis. We review studies belonging to twelve categories of software engineering tasks and corresponding machine learning techniques, tools, and datasets that have been applied to solve them. To do so, we conducted an extensive literature search and identified 494 studies. We summarize our observations and findings with the help of the identified studies. Our findings suggest that the use of machine learning techniques for source code analysis tasks is consistently increasing. We synthesize commonly used steps and the overall workflow for each task and summarize machine learning techniques employed. We identify a comprehensive list of available datasets and tools usable in this context. Finally, the paper discusses perceived challenges in this area, including the availability of standard datasets, reproducibility and replicability, and hardware resources.

Keywords: Machine learning for software engineering, source code analysis, deep learning, datasets, tools.

1. Introduction

In the last two decades, we have witnessed significant advancements in Machine Learning (ML), including Deep Learning (DL) techniques, specifically in the domain of image [237, 476], text [255, 4], and speech [418, 166, 165] processing. These advancements, coupled with a large amount of open-source code and associated artifacts, as well as the availability of accelerated hardware, have encouraged researchers and practitioners to use ML techniques to address software engineering problems [513, 561, 27, 248, 34].

The software engineering community has employed ML and DL techniques for a variety of applications such as software testing [275, 361, 564], source code representation [27, 191], source code quality analysis [34, 45], program synthesis [248, 540], code completion [288], refactoring [40], code summarization [295, 252, 24], and vulnerability analysis [440, 429, 501] that involve source code analysis. As the field of Machine Learning for Software Engineering (ML4SE) is expanding, the number of available resources, methods, and techniques as well as tools and datasets, is also increasing. This poses a challenge, to both researchers and practitioners, to fully comprehend the landscape of the available resources and infer the potential directions that the field is taking.
In this context, literature surveys play an important role in understanding existing research, finding gaps in research or practice, and exploring opportunities to improve the state of the art. By systematically examining existing literature, surveys may uncover hidden patterns, recurring themes, and promising research directions. Surveys also identify untapped opportunities and formulation of new hypotheses. A survey also serves as an educational tool, offering comprehensive coverage of the field to a newcomer.

In fact, there have been numerous recent attempts to summarize the application-specific knowledge in the form of surveys. For example, Allamanis et al. [27] present key methods to model source code using ML techniques. Shen and Chen [440] provide a summary of research methods associated with software vulnerability detection, software program repair, and software defect prediction. Durelli et al. [132] collect 48 primary studies focusing on software testing using machine learning. Alsolai and Roper [34] present a systematic review of 56 studies related to maintainability prediction using ML techniques. Recent surveys [487, 13, 45] summarize application of ML techniques on software code smells and technical debt identification. Similarly, literature reviews on program synthesis [248] and code summarization [348] have been attempted. We compare in Table 1 the aspects investigated in our survey with respect to existing surveys that review ML techniques for topics such as testing, vulnerabilities, and program comprehension with our survey. Existing studies, in general, kept their focus on only one category; due to that readers could not grasp existing literature belonging to various software engineering categories in a consistent form. In addition, existing surveys do not always provide datasets and tools in the field. Our survey, covers a wide range of software engineering activities; it summarizes a significantly large number of studies; it systematically examines available tools and datasets for ML that would support researchers in their studies in this field; it identifies perceived challenges in the field to encourage the community to explore ways to overcome them.

In this paper, we focus on the usage of ML, including DL, techniques for source code analysis.

Source code analysis involves tasks that take the source code as input, process it, and/or produce source code as output. Source code representation, code quality analysis, testing, code summarization, and program synthesis are applications that involve source code analysis. To the best of our knowledge, the software engineering literature lacks a survey covering a wide range of source code analysis applications using machine learning; this work is an attempt to fill this research gap.

In this survey, we aim to give a comprehensive, yet concise, overview of current knowledge on applied machine learning for source code analysis. We also aim to collate and consolidate available resources (in the form of datasets and tools) that researchers have used in previous studies on this topic. Additionally, we aim to identify and present challenges in this domain. We believe that our efforts to consolidate and summarize the techniques, resources, and challenges will help the community to not only understand the state-of-the-art better, but also to focus their efforts on tackling the identified challenges.

This survey makes the following contributions to the field:

- It presents a summary of the applied machine learning studies attempted in the source code analysis domain.
- It consolidates resources (such as datasets and tools) relevant for future studies in this domain.
- It provides a consolidated summary of the open challenges that require the attention of the researchers.

The rest of the paper is organized as follows. We present the followed methodology, including the literature search protocol and research questions, in Section 2. Section 2.3, Section 3, Section 4, and Section 5 provide the detailed results of our findings. We present threats to validity in Section 6, and conclude the paper in Section 7.
Table 1. Comparison Among Surveys. The “Category” column refers to the software engineering task the survey covers. The “Scope” column indicates the focus of the study; TML refers to traditional machine learning and DL refers to deep learning techniques. The “Data&Tools” column indicates if a survey reviews available datasets and tools for ml-based applications, the “Challenges” column shows whether the study identifies challenges in the field studied, the “Type” column refers to the type of literature survey, and the “#Studies” column refers to the number of studies included in a given survey. We use “–” to indicate that a field is not applicable to a certain study and NA for the number of studies column, where the study does not explicitly mention selection criteria and the number of selected studies.

<table>
<thead>
<tr>
<th>Category</th>
<th>Article</th>
<th>Scope</th>
<th>Data &amp; Tools</th>
<th>Challenges</th>
<th>Type</th>
<th>#Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Comprehension</td>
<td>Nazar et al. [348]</td>
<td>TML</td>
<td>Tools</td>
<td>No</td>
<td>Lit. survey</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Zhang et al. [560]</td>
<td>DL</td>
<td>Data</td>
<td>No</td>
<td>Lit. survey</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Song et al. [458]</td>
<td>TML &amp; DL</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>NA</td>
</tr>
<tr>
<td>Testing</td>
<td>Omri and Sinz [361]</td>
<td>DL</td>
<td>No</td>
<td>No</td>
<td>Lit. survey</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Durelli et al. [132]</td>
<td>TML &amp; DL</td>
<td>No</td>
<td>Yes</td>
<td>Mapping study</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>Hall and Bowes [181]</td>
<td>TML</td>
<td>Yes</td>
<td>Yes</td>
<td>Meta-analysis</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Zhang et al. [564]</td>
<td>TML &amp; DL</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Pandey et al. [368]</td>
<td>TML</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>Singh et al. [452]</td>
<td>TML</td>
<td>No</td>
<td>No</td>
<td>Lit. survey</td>
<td>NA</td>
</tr>
<tr>
<td>Vulnerability analysis</td>
<td>Li et al. [271]</td>
<td>DL</td>
<td>Yes</td>
<td>Yes</td>
<td>Meta-analysis</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Shen and Chen [440]</td>
<td>DL</td>
<td>No</td>
<td>Yes</td>
<td>Meta-analysis</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Ucci et al. [501]</td>
<td>TML</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Jie et al. [215]</td>
<td>TML</td>
<td>No</td>
<td>No</td>
<td>Lit. survey</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Hanif et al. [187]</td>
<td>TML &amp; DL</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>90</td>
</tr>
<tr>
<td>Quality assessment</td>
<td>Alsolai and Roper [34]</td>
<td>TML</td>
<td>No</td>
<td>No</td>
<td>Lit. survey</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Tsintzira et al. [487]</td>
<td>TML</td>
<td>Yes</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Azeem et al. [45]</td>
<td>TML</td>
<td>Yes</td>
<td>No</td>
<td>Lit. survey</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Caram et al. [77]</td>
<td>TML</td>
<td>No</td>
<td>No</td>
<td>Mapping study</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Lewowski and Madeyski [259]</td>
<td>TML</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>45</td>
</tr>
<tr>
<td>Prog. synthesis</td>
<td>Goues et al. [162]</td>
<td>TML &amp; DL</td>
<td>No</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Le et al. [248]</td>
<td>DL</td>
<td>Yes</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>NA</td>
</tr>
<tr>
<td>Prog. synthesis &amp; code</td>
<td>Allamanis et al. [27]</td>
<td>TML &amp; DL</td>
<td>Yes</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>39+48</td>
</tr>
<tr>
<td>representation</td>
<td>Yang et al. [544]</td>
<td>DL</td>
<td>Data</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>250</td>
</tr>
<tr>
<td>Source-code analysis</td>
<td>Our study</td>
<td>TML &amp; DL</td>
<td>Yes</td>
<td>Yes</td>
<td>Lit. survey</td>
<td>494</td>
</tr>
</tbody>
</table>
2. Methodology

First, we present the objectives of this study and the research questions derived from such objectives. Second, we describe the search protocol we followed to identify relevant studies. The protocol identifies detailed steps to collect the initial set of articles as well as the inclusion and exclusion criteria to obtain a filtered set of studies.

2.1 Research objectives

This study aims to achieve the following research objectives (ROs).

RO1. Identifying specific software engineering tasks involving source code that have been attempted using machine learning.

Our objective is to explore the extent to which machine learning has been applied to analyze and process source code for SE tasks. We aim to summarize how ML can help engineers tackle specific SE tasks.

RO2. Summarizing the machine learning techniques used for these tasks.

This objective explores the ML techniques commonly applied to source code for performing the software engineering tasks identified above. We attempt to synthesize a mapping of tasks (along with related sub-tasks) and corresponding ML techniques.

RO3. Providing a list of available datasets and tools.

With this goal, we aim to provide a consolidated summary of publicly available datasets and tools along with their purpose.

RO4. Identifying the challenges and perceived deficiencies in ML-enabled source code analysis and manipulation for software engineering.

With this objective, we aim to identify challenges, and opportunities arising when applying ML techniques to source code for SE tasks, as well as to understand the extent to which they have been addressed in the articles surveyed.

2.2 Literature search protocol

We identified 494 relevant studies through a four step literature search. Figure 1 summarizes the search process. We elaborate on each of these phases in the rest of this section.

![Figure 1. Overview of the search process](image)
2.2.1 Literature search—Phase 1

We split the phase 1 literature search into two rounds. In the first round, we carried out an extensive initial search on six well-known digital libraries—Google Scholar, SpringerLink, ACM Digital Library, ScienceDirect, IEEE Xplore, and Web of Science during Feb-Mar 2021. We formulated a set of search terms based on common tasks and software engineering activities related to source code analysis. Specifically, we used the following terms for the search: \textit{machine learning code}, \textit{machine learning code representation}, \textit{machine learning testing}, \textit{machine learning code synthesis}, \textit{machine learning smell identification}, \textit{machine learning security source code analysis}, \textit{machine learning software quality assessment}, \textit{machine learning code summarization}, \textit{machine learning program repair}, \textit{machine learning code completion}, and \textit{machine learning refactoring}. We searched minimum seven pages of search results for each search term manually; beyond seven pages, we continued the search unless we get two continuous search pages without any new and relevant articles. We adopted this mechanism to avoid missing any relevant articles in the context of our study.

In the second round of phase 1, we identified a set of frequently occurring keywords in the articles obtained from the first round for each category individually. To do that, we manually scanned the keywords mentioned in the articles belonging to each category, and noted the keywords that appeared at least three times. If the selected keywords are too generic, we first check whether adding \textit{machine learning} would improve the search results. For example, \textit{machine learning} and \textit{program generation} occurred multiple times in the \textit{program synthesis} category; we combined both of these terms to make one search string \textit{i.e., program generation using machine learning}. In other cases, we tried to reduce the scope of the search term by adding qualifying terms. Consider \textit{feature learning} as an example: it is so generic that would result in many unrelated results. We reduced the search scope by adding \textit{source code} in the search \textit{i.e., searching using feature learning in source code}. We carried out this additional round of literature search to augment our initial search terms and reduce the risk of missing relevant articles. The full list of search terms used in the second round of phase 1 can be found in our replication package [438]. Next, we defined inclusion and exclusion criteria to filter out irrelevant studies.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Category} & \textbf{Search terms} & \textbf{#Studies} \\
\hline
Vulnerability analysis & feature learning in source code & 9 \\
 & vulnerability prediction in source code using machine learning & 70 \\
 & deep learning-based vulnerability detection & 8 \\
 & malicious code detection with machine learning & 45 \\
\hline
Testing & word embedding in software testing & 2 \\
 & automated Software Testing with machine learning & 12 \\
 & optimal machine learning based random test generation & 1 \\
\hline
Refactoring & source code refactoring prediction with machine learning & 39 \\
 & automatic clone recommendation with machine learning & 14 \\
 & machine learning based refactoring detection tools & 16 \\
 & search-based refactoring with machine learning & 6 \\
\hline
Quality assessment & web service anti-pattern detection with machine learning & 25 \\
 & code smell prediction models & 34 \\
 & machine learning-based approach for code smells detection & 17 \\
 & software design flaw prediction & 37 \\
 & linguistic smell detection with machine learning & 2 \\
 & software defect prediction with machine learning & 66 \\
 & machine learning based software fault prediction & 35 \\
\hline
Program synthesis & automated program repair methods with machine learning & 45 \\
\hline
\end{tabular}
\caption{Search terms and corresponding relevant studies found in the second round of phase 1.}
\end{table}
### Inclusion criteria:

- Studies and surveys that discuss the application of machine learning (including DL) to source code to perform a software engineering task.
- Resources revealing the deficiencies or challenges in the current set of methods, tools, and practices.

### Exclusion criteria:

- Studies focusing on techniques other than ML applied on source code to address software engineering tasks e.g., code smell detection using metrics.
- Articles that are not peer-reviewed (such as articles available only on arXiv.org).
- Articles constituting a keynote, extended abstract, editorial, tutorial, poster, or panel discussion (due to insufficient details and limited length).
- Studies whose full text is not available, or is written in any other language than English.

We considered whether to include studies that do not directly analyze source code. Often, source code is analyzed to extract features, and machine learning techniques are applied to the extracted features. Furthermore, researchers in the field either create their own dataset (in that case, analyze/process source code) or use existing datasets. Removing studies that use a dataset will make this survey incomplete; hence, we decided to include such studies.

During the search, we documented studies that satisfy our search protocol in a spreadsheet including the required meta-data (such as title, bibtex record, and link of the source). The spreadsheet with all the articles from each phase can be found in our online replication package [438].

Each selected article went through a manual inspection of title, keywords, and abstract. The inspection applied the inclusion and exclusion criteria leading to inclusion or exclusion of the articles. In the end, we obtained 1,576 articles after completing Phase 1 of the search process.

#### 2.2.2 Literature search—Phase 2

We first identified a set of categories and sub-categories for common software engineering tasks. These tasks are commonly referred in recent publications [147, 27, 440, 45]. These categories

<table>
<thead>
<tr>
<th>Program generation with machine learning</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-oriented program repair with machine learning</td>
<td>15</td>
</tr>
<tr>
<td>Predicting patch correctness with machine learning</td>
<td>3</td>
</tr>
<tr>
<td>Multihunk program repair with machine learning</td>
<td>9</td>
</tr>
<tr>
<td>Autogenerated code with machine learning</td>
<td>6</td>
</tr>
<tr>
<td>Commits analysis with machine learning</td>
<td>34</td>
</tr>
<tr>
<td>Supplementary bug fixes with machine learning</td>
<td>9</td>
</tr>
<tr>
<td>Automatic source code summarization with machine learning</td>
<td>43</td>
</tr>
<tr>
<td>Automatic commit message generation with machine learning</td>
<td>19</td>
</tr>
<tr>
<td>Comments generation with machine learning</td>
<td>11</td>
</tr>
<tr>
<td>Security flaws detection in source code with machine learning</td>
<td>20</td>
</tr>
<tr>
<td>Intelligent source code security review with machine learning</td>
<td>2</td>
</tr>
<tr>
<td>Design pattern detection with machine learning</td>
<td>10</td>
</tr>
<tr>
<td>Human-machine-comprehensible software representation</td>
<td>1</td>
</tr>
<tr>
<td>Feature learning in source code</td>
<td>6</td>
</tr>
<tr>
<td>Missing software architectural tactics prediction with machine learning</td>
<td>1</td>
</tr>
<tr>
<td>Software system quality analysis with machine learning</td>
<td>6</td>
</tr>
<tr>
<td>Package-level tactic recommendation generation in source code</td>
<td>3</td>
</tr>
<tr>
<td>Identifier prediction in source code</td>
<td>13</td>
</tr>
<tr>
<td>Token prediction in source code</td>
<td>29</td>
</tr>
</tbody>
</table>
and sub-categories of common software engineering tasks can be found in Figure 3. Then, we manually assigned a category and sub-category, if applicable, to each selected article based on the (sub-)category to which an article contributes the most. The assignment was carried out by one of the authors and verified by two other authors. We computed Cohen’s Kappa [329] to measure the initial disagreement; we found a strong agreement among the authors with $\kappa = 0.87$. In case of disagreement, each author specified a key goal, operation, or experiment in the article, indicating the rationale of the category assignment for the article. This exercise resolved the majority of the disagreements. In the rest of the cases, we discussed the rationale identified by individual authors and voted to decide a category or sub-category to which the article contributes the most. In this phase, we also discarded duplicates or irrelevant studies not meeting our inclusion criteria after reading their title and abstract. After this phase, we were left with 1,098 studies.

2.2.3 Literature search—Phase 3

In the last decade, the use of ML has increased significantly. The research landscape involving source code and ML, which includes methods, applications, and required resources, has changed significantly in the last decade. To keep the survey focused on recent methods and applications, we focused on studies published after 2011. Also, we discarded papers that had not received enough attention from the community by filtering out all those having a `citation count < (2021 – publication year)’. We chose 2021 as the base year to not penalize studies that came out recently; hence, the studies that are published in 2021 do not need to have any citation to be included in this search. We obtain the citation count from digital libraries manually during Mar-May 2022. After applying this filter, we obtained 977 studies.

2.2.4 Literature search—Phase 4

In this phase, we discarded those studies that do not satisfy our inclusion criteria (such as when the article is too short or do not apply any ML technique to source code for SE tasks) after reading the whole article. The remaining 494 articles are the selected studies that we examine in detail. For each study, we extracted the core idea and contribution, the ML techniques, datasets and tools used as well as challenges and findings unveiled. Next, we present our observations corresponding to each research goal we pose.

2.3 Assigning articles to software engineering task categories

Towards achieving RO1, we tagged each selected article with one of the task categories based on the primary focus of the study. The categories represent common software engineering tasks that involve source code analysis. These categories are code completion, code representation, code review, code search, dataset mining, program comprehension, program synthesis, quality assessment, refactoring, testing, and vulnerability analysis. If a given article does not fall in any of these categories but is still relevant to our discussion as it offers overarching discussion on the topic; we put the study in the general category. Figure 2 presents a category-wise distribution of studies per year. It is evident that the topic is engaging the research community more and more and we observe, in general, a healthy upward trend. Interestingly, the number of studies in the scope dropped significantly in the year 2021.

Some of the categories are quite generic and hence further categorization is possible based on specific tasks. For each category, we identified sub-categories by grouping related studies together and assigning an intuitive name representing the set of the studies. For example, the testing category is further divided into defect prediction, and test data/case generation. We attempted to assign a sub-category to each study; if none of the sub-categories was appropriate for a study, we did not assign any sub-category to the study. One author of this paper assigned a sub-category to each study based on the topic to which that study contributed the most. The initial assignment was verified by two other authors of this paper, where disagreements were discussed and resolved to reach a consensus. Figure 3 presents the distribution of studies per year w.r.t. each category and...
To quantify the growth of each category, we compute the average increase in the number of articles from the last year for each category between the years 2012 and 2022. We observed that the program synthesis and vulnerability analysis categories grew most with approximately 44% and 50% average growth each year, respectively.
<table>
<thead>
<tr>
<th>Methodology</th>
<th>Model-based</th>
<th>Tree-based</th>
<th>Instance-based</th>
<th>Probabilistic-based</th>
<th>Rule-based</th>
<th>Learn-to-Rank</th>
<th>Clustering</th>
<th>Other</th>
<th>Evolutionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code representation</td>
<td>TML-SUP-MOD-SVR</td>
<td>Boosted Decision Trees</td>
<td>TML-SUP-IN-CWWEB</td>
<td>Bayes Net</td>
<td>Decision Table</td>
<td>Diverse Rank</td>
<td>Hierarchical Clustering</td>
<td>Fuzzy Logic</td>
<td>Gene Expression Programming</td>
</tr>
<tr>
<td>Code search</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-POLY</td>
<td></td>
<td></td>
<td></td>
<td>Latent Dir-Net Allocation</td>
<td>TML-UNSUP-MGP</td>
</tr>
<tr>
<td>Data set mining</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
<tr>
<td>Program comprehension</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-LDSVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MP</td>
</tr>
<tr>
<td>Program synthesis</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-LR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
<tr>
<td>Quality assessment</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-LDA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
<tr>
<td>Refactoring</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-LMSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
<tr>
<td>Ranking</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-LASSO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
<tr>
<td>Vulnerability analysis</td>
<td></td>
<td></td>
<td></td>
<td>TML-SUP-MOD-LSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TML-UNSUP-MGTE</td>
</tr>
</tbody>
</table>

**Table 3. Usage of ML techniques in the selected studies (Part 1)**

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Code representation</th>
<th>Code review</th>
<th>Code search</th>
<th>Dataset mining</th>
<th>Program comprehension</th>
<th>Program synthesis</th>
<th>Quality assessment</th>
<th>Refactoring</th>
<th>Testing</th>
<th>Vulnerability analysis</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reinforcement Learning</td>
<td>Dual Deep Q-Networks</td>
<td>RL-DDQN</td>
<td>0 0 0 0 0 0 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reinforcement Learning</td>
<td>RL-RL</td>
<td>0 0 0 0 0 0 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td>Hybrid</td>
<td>Adapative neuro fuzzy inference system</td>
<td>OTH-HYF-NFIS</td>
<td>0 0 0 0 0 0 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimization Techniques</td>
<td>Expectation Maximization</td>
<td>OTH-OPT-EM</td>
<td>0 0 0 0 0 0 1 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gradient Descent</td>
<td>OTH-OPT-GD</td>
<td>0 0 0 0 0 0 1 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stochastic Gradient Descent</td>
<td>OTH-OPT-SGD</td>
<td>0 0 0 0 0 0 2 0 2 0 2 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequential Minimal Optimization</td>
<td>OTH-OPT-SMO</td>
<td>0 0 0 0 0 0 5 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Particle Swarm Optimization</td>
<td>OTH-OPT-FSO</td>
<td>0 0 0 0 0 0 1 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Usage of ML techniques in the selected studies (Part-2)

3. Literature Survey Results

We document our observations per category and subcategory by providing a summary of the existing efforts to achieve RO2 of the study. Table 3 and Table 4 show the frequency of the various ML techniques per software engineering task category used in the selected studies. The tables also classify the ML techniques into a hierarchical classification based on the characteristics of the ML techniques. Specifically, the first level of classification divides ML techniques into traditional machine learning (mL), deep learning (DL), reinforcement learning (RL), and others (OTH) that include hybrid and optimization techniques. Furthermore, we identify sub-categories and ML techniques corresponding to each category. To generate these tables, we identified ML techniques used in
each study while summarizing the study. Given that a study may use multiple ML techniques, we developed a script to split the techniques and create a CSV file containing one ML technique and the corresponding paper category. We then compute a number of times for each ML technique for each software engineering task category to generate the tables. In these tables we refer to ML techniques with their commonly used acronym along with their category and sub-category. It is evident from these tables that SVM, RF, and DT are the most frequently used traditional ML techniques, whereas, the KNN family (including LSTM and GRU) is the most commonly used ML technique.

**Evolution of ML techniques use over time:** In addition, we segregate the identified ML techniques by their category (i.e., TML, DL, RL, and OTH) and year of publication. Figure 4 presents the summary of the analysis. We observe that majorly traditional ML and DL approaches are used in this field. We also observe that the use of DL approaches for source code analysis has significantly increased from 2016.

![Figure 4. Usage of ML techniques by categories per year](image)

**Venue and article categories:** We identified and manually curated the software engineering venue for each study discussed in our literature review. Figure 5 shows the venues for the considered categories. We show the most prominent venues per category. Each label includes a number indicating the number of articles published at the same venue in that category.

We observe that ICSE is the top venue, appearing in three categories. IEEE Access is the top journal for the considered categories. Machine learning conferences such as ICLR also appear as the top venues for the program synthesis category. The category program comprehension exhibits the highest concentration of articles to a relatively small list of top venues where approximately 50% of articles come from the top venues (with at least four studies). On the other hand, researchers publish articles related to testing, code completion, and vulnerability analysis in a rather diverse set of venues.

**Target programming languages:** We identified the target programming language of each study to observe the focus of researchers in the field by category. Figure 6 presents the result of the analysis. We observe that for most of the categories, Java dominates the field. For quality assessment category, studies also analyzed source code written in C/C++, apart from Java. Researchers analyzed Python programs also, apart from Java, for studies belonging to program comprehension and program synthesis. This analysis, on the one hand, shows that Java, C/C++, and Python are the most analyzed programming languages in this field; on the other hand, it points out the lack of studies targeting other prominent programming languages per category.

**Popular models:** As part of collecting metadata and summarizing studies, we identified the proposed model, if any, for each selected study. We considered novel proposed models only and not the name of the approach or method in this analysis. We also obtained the number of citations for the study. In Table 5, we present the most popular model, in no particular order, by using the number of citations as the metric to decide the popularity. We collected the number of citations at the end of August 2023 and included all the models with corresponding citations over 100.

In the rest of this section, we delve into each category and sub-category at a time, break down the entire workflow of a code analysis task into fine-grained steps, and summarize the method and ML techniques used. It is worth emphasizing that we structure the discussion around the cru-
cial steps for each category (e.g., model generation, data sampling, feature extraction, and model training).

### 3.1 Code representation

Raw source code cannot be fed directly to a DL model. Code representation is the fundamental activity to make source code compatible with DL models by preparing a numerical representation of the code to further solve a specific software engineering task. Code representation is the process of transforming the textual program source code into a numerical representation i.e., vectors that a DL model can accept and process \[227\]. Studies in this category emphasize that source code is a richer construct and hence should not be treated simply as a collection of tokens or text \[350, 27\]; the proposed techniques extensively utilize the syntax, structure, and semantics (such as type information from an AST). The activity transforms source code into a numerical representation making it easier to further use the code by ML models to solve specific tasks such as code pattern identification \[342, 480\], method name prediction \[32\], and comment classification \[514\].

In the training phase, a large number of repositories are processed to train a model which is then used in the inference phase. Source code is pre-processed to extract a source code model (such as an AST or a sequence of tokens) which is fed into a feature extractor responsible to mine the necessary features (for instance, AST paths and tree-based embeddings). Then, an ML model is
Figure 6. Target programming languages for each considered category

trained using the extracted features. The model produces a numerical (i.e., a vector) representation that can be used further for specific software engineering applications such as defect prediction, vulnerability detection, and code smells detection.

**Dataset preparation:** Code representation efforts start with preparing a source code model. The majority of the studies use the AST representation [350, 30, 563, 25, 91, 31, 32, 540, 67, 525, 84, 377, 376]. Some studies [439, 22, 44, 83, 574, 219, 352, 343, 134] parsed the source code as tokens and prepared a sequence of tokens in this step. Hoang et al. [194] generated tokens representing only the code changes. Furthermore, Sui et al. [465] compiled a program into llvm-ir. An inter-procedural value-flow graph (ivfg) used was built on top of the intermediate representation. Thaller et al. [480] used abstract semantic graphs as their code model. Nie et al. [353] used dataset offered by Jiang et al. [209] that offers a large number code snippets and comment pairs. Finally, Brauckmann et al. [66] and Tufano et al. [490] generated multiple source code models (AST, CFG, and byte code).

**Feature extraction:** Relevant features need to be extracted from the prepared source code model for further processing. The first category of studies, based on applied feature extraction mechanism, uses token-based features. Nguyen et al. [350] prepared vectors of syntactic context (referred to as syntaxemel), type context (sememes), and lexical tokens. Shedko et al. [439] generated a stream of tokens corresponding to function calls and control flow expressions. Karampatsis et al. [221] split tokens as subwords to enable subwords prediction. Path-based abstractions is the basis of the second category where the studies extract a path typically from an AST. Alon et al. [30] used paths between AST nodes. Kovalenko et al. [235] extracted path context representing two tokens.
Table 5. Popular models proposed in the selected studies.

<table>
<thead>
<tr>
<th>Model</th>
<th>#Citations</th>
<th>Model</th>
<th>#Citations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst2Vec [57]</td>
<td>234</td>
<td>Code-NN [204]</td>
<td>681</td>
</tr>
<tr>
<td>TBCNN [342]</td>
<td>695</td>
<td>Program as graph model [67]</td>
<td>159</td>
</tr>
<tr>
<td>SLAMC [352]</td>
<td>130</td>
<td>Coding criterion [377]</td>
<td>128</td>
</tr>
<tr>
<td>TransCoder [408]</td>
<td>115</td>
<td>TreeGen [468]</td>
<td>124</td>
</tr>
<tr>
<td>Codex [93]</td>
<td>897</td>
<td>AlphaCode [270]</td>
<td>317</td>
</tr>
</tbody>
</table>

in code and a structural connection along with paths between AST nodes. Alon et al. [31] encoded each AST path with its values as a vector and used the average of all of the \( k \) paths as the decoder's initial state where the value of \( k \) depends on the number of leaf nodes in the AST. The decoder then generated an output sequence while attending over the \( k \) encoded paths. Peng et al. [377] proposed \"coding criterion\" to capture similarity among symbols based on their usage using AST structural information. Peng et al. [376] used open-source parser Tree-Sitter to obtain AST for each method. They split code tokens into sub-tokens respective to naming conventions and generate path using AST nodes. The authors sets 32 as the maximum path length. Finally, Alon et al. [32] also used path-based features along with distributed representation of context where each of the path and leaf-values of a path-context is mapped to its corresponding real-valued vector representation.

Another set of studies belong to the category that used graph-based features. Chen et al. [91] created AST node identified by an API name and attached each node to the corresponding AST node belonging to the identifier. Thaller et al. [480] proposed feature maps; feature maps are human-interpreted, stacked, named subtrees extracted from abstract semantic graph. Brauckmann et al. [66] created a dataflow-enriched AST graph, where nodes are labeled as declarations, statements, and types as found in the Clang\footnote{https://clang.llvm.org/}. Cvitkovic et al. [115] augmented AST with semantic information by adding a graph-structured vocabulary cache. Finally, Zhang et al. [563] extracted small statement trees along with multi-way statement trees to capture the statement-level lexical and syntactical information. The final category of studies used DL [194, 490] to learn features automatically.

ML model training: The majority of the studies rely on the RNN-based DL model. Among them, some of the studies [514, 191, 525, 66, 31] employed LSTM-based models; while others [563, 194, 221, 540, 67] used CNN-based models. Among the other kinds of ML models, studies employed GNN-based [115, 528], DNN [350], conditional random fields [30], SVM [274, 394], CNN-based models [91, 342, 480], and transformer-based models [376]. Some of the studies rely on the combination of different DL models. For example, Tufano et al. [490] employed RNN-based model for learning embedding in the first stage which is given to an autoencoder-based model to encode arbitrarily long streams of embeddings.

A typical output of a code representation technique is the vector representation of the source code. The exact form of the output vector may differ based on the adopted mechanism. Often, the code vectors are application specific depending upon the nature of features extracted and training mechanism. For example, Code2Vec produces code vectors trained for method name prediction; however, the same mechanism can be used for other applications after tuning and selecting appropriate features. Kang et al. [220] carried out an empirical study to observe whether...
the embeddings generated by Code2Vec can be used in other contexts. Similarly, Pour et al. [385] used Code2Vec, Code2Seq, and CodeBERT to explore the robustness of code embedding models by retraining the models using the generated adversarial examples.

The semantics of the produced embeddings depend significantly on the selected features. Studies in this domain identify this aspect and hence swiftly focused to extract features that capture the relevant semantics; for example, path-based features encode the order among the tokens. The chosen ML model plays another important role to generate effective embeddings. Given the success of RNN with text processing tasks, due to its capability to identify sequence and pattern, RNN-based models dominate this category.

3.2 Testing

In this section, we point out the state-of-the-art regarding ML techniques applied to software testing. Testing is the process of identifying functional or non-functional bugs to improve the accuracy and reliability of a software. In this section, we offer a discussion on test cases generation by employing ML techniques.

3.2.1 Test data and test cases generation

A usual approach to have a ML model for generating test oracles involves capturing data from an application under test, pre-processing the captured data, extracting relevant features, using an ML algorithm, and evaluating the model.

**Dataset preparation:** Researchers developed a number of ways for capturing data from applications under test and pre-process them before feeding them to an ML model. Braga et al. [65] recorded traces for applications to capture usage data. They sanitized any irrelevant information collected from the programs recording components. AppFlow [197] captures human-event sequences from a smart-phone screen in order to identify tests. Similarly, Nguyen et al. [351] suggested Shinobi, a framework that uses a fast R-CNN model to identify input data fields from multiple web-sites. Utting et al. [505] captured user and system execution traces to help generating missing API tests. To automatically identify metamorphic relations, Nair et al. [345] suggested an approach that leverages ML techniques and test mutants. By using a variety of code transformation techniques, the authors’ approach can generate a synthetic dataset for training models to predict metamorphic relations.

**Feature extraction:** Some authors [65, 505] used execution traces as features. Kim et al. [230] suggested an approach that replaces SBST’s meta-heuristic algorithms with deep reinforcement learning to generate test cases based on branch coverage information. [164] used code quality metrics such as coupling, DIT, and NOF to generate test data; they use the test data generated to predict the code coverage in a continuous integration pipeline.

**ML model training:** Researchers used supervised and unsupervised ML algorithms to generate test data and cases. In some of the studies, the authors utilized more than one ML algorithm to achieve their goal. Specifically, several studies [65, 230, 505, 345] used traditional ML algorithms, such as Support Vector Machine, Naive Bayes, Decision Tree, Multilayer Perceptron, Random Forest, AdaBoost, Linear Regression. Nguyen et al. [351] used the DL algorithm Fast R-CNN. Similarly, [156] used LSTM to automate generating the input grammar data for fuzzing.

3.3 Program synthesis

This section summarizes the ML techniques used by automated program synthesis tools and techniques in the examined software engineering literature. Apart from a major sub-category program repair, we also discuss state-of-the-art corresponds to code generation and program translation sub-categories in this section.
3.3.1 Program repair

Automated Program Repair (APR) refers to techniques that attempt to automatically identify patches for a given bug (i.e., programming mistakes that can cause an unintended run-time behavior), which can be applied to software with a little or without human intervention [162]. Program repair typically consists of two phases. Initially, the repair tool uses fault localization to detect a bug in the software under examination, then, it generates patches using techniques such as search-based software engineering and logic rules that can possibly fix a given bug. To validate the generated patch, the (usually manual) evaluation of the semantic correctness of that patch follows.

According to Goues et al. [162], the techniques for constructing repair patches can be divided into three categories (heuristic repair, constraint-based repair, and learning-aided repair) if we consider the following two criteria: what types of patches are constructed and how the search is conducted. Here, we are interested in learning-aided repair, which leverages the availability of previously generated patches and bug fixes to generate patches. In particular, learning-aided-based repair tools use ML to learn patterns for patch generation.

Typically, at the pre-processing step, such methods take source code of the buggy revision as an input, and those revisions that fixes the buggy revision. The revision with the fixes includes a patch carried out manually that corrects the buggy revision and a test case that checks whether the bug has been fixed. Learning-aided-based repair is mainly based on the hypothesis that similar bugs will have similar fixes. Therefore, during the training phase, such techniques can use features such as similarity metrics to match bug patterns to similar fixes. Then, the generated patches rely on those learnt patterns. Next, we elaborate upon the individual steps involved in the process of program repair using ML techniques.

Dataset preparation: The majority of the studies extract buggy project revisions and manual fixes from buggy software projects. Most studies leverage source-code naturalness. For instance, Tufano et al. [492] extracted millions of bug-fixing pairs from GitHub, Amorim et al. [39] leveraged the naturalness obtained from a corpus of known fixes, and Chen et al. [97] used natural language structures from source code. Furthermore, many studies develop their own large-scale bug benchmarks. Ahmed et al. [10] leveraged 4,500 erroneous C programs, Gopinath et al. [161] used a suite of successful manual patches from open-source software repositories, Long and Rinard [297] used a set of successful manual patches from open-source software repositories, and Mashhadi and Hemmati [326] used the ManySStuBs4J dataset containing natural language description and code snippets to automatically generate code fixes. Le et al. [249] created an oracle for predicting which bugs should be delegated to developers for fixing and which should be fixed by repair tools. Jiang et al. [211] used a dataset containing more than 4 million methods extracted. White et al. [533] used Spoon, an open-source library for analyzing and transforming Java source code, to build a model for each buggy program revision. Pinconschi et al. [382] constructed a dataset containing vulnerability-fix pairs by aggregating five existing dataset (Mozilla Foundation Security Advisories, SecretPatch, NVD, Secbench, and Big-Vul). The dataset i.e., PatchBundle is publicly available on GitHub. Cambronero and Rinard [76] proposed a method to generate new supervised machine learning pipelines. To achieve the goal, the study trained using a collection of 500 supervised learning programs and their associated target datasets from Kaggle. Liu et al. [287] prepared their dataset by selecting 636 closed bug reports from the Linux kernel and Mozilla databases. Svyatkovskiy et al. [475] constructed their experimental dataset from the 2700 top-starred Python source code repositories on GitHub. CODIT [82] collects a new dataset—Code-ChangeData, consisting of 32,473 patches from 48 open-source GitHub projects collected from Travis Torrent.

Other studies use existing bug benchmarks, such as DEFECTS4J [218] and IntroClass [250], which already include buggy revisions and human fixes, to evaluate their approaches. For instance, Saha et al. [416], Lou et al. [299], Zhu et al. [582], Renzullo et al. [406], Wang et al. [518], and Chen

\(^2\)The term semantic correctness is a criterion for evaluating whether a generated patch is similar to the human fix for a given bug [291].
et al. [101] leveraged Defects4J for the evaluations of their approaches. Additionally, Dantas et al. [118] used the IntroClass benchmark and Majd et al. [313] conducted experiments using 119,989 C/C++ programs within Code4Bench. Wu et al. [534] used the DeepFix dataset that contains 46,500 correct C programs and 6,975 programs with errors for their graph-based DL approach for syntax error correction.

Some studies examine bugs in different programming languages. For instance, Svyatkovskiy et al. [474] used 1.2 billion lines of source code in Python, C#, JavaScript, and TypeScript programming languages. Also, Lutellier et al. [305] used six popular benchmarks of four programming languages (Java, C, Python, and JavaScript).

There are also studies that mostly focus on syntax errors. In particular, Gupta et al. [178] used 6,975 erroneous C programs with typographic errors, Santos et al. [421] used source code files with syntax errors, and Sakkas et al. [419] used a corpus of 4,500 ill-typed OCaml programs that lead to compile-time errors. Bhatia et al. [59] examined a corpus of syntactically correct submissions for a programming assignment. They used a dataset comprising of over 14,500 student submissions with syntax errors.

Finally, there is a number of studies that use programming assignment from students. For instance, Bhatia et al. [59], Gupta et al. [178], and Sakkas et al. [419] used a corpus of 4,500 ill-typed OCaml student programs.

Feature extraction: The majority of studies utilize similarity metrics to extract similar bug patterns and, respectively, correct bug fixes. These studies mostly employ word embeddings for code representation and abstraction. In particular, Amorim et al. [39], Svyatkovskiy et al. [474], Santos et al. [421], Jiang et al. [211], and Chen et al. [97], leveraged source-code naturalness and applied NLP-based metrics. Tian et al. [483] employed different representation learning approaches for code changes to derive embeddings for similarity computations. Similarly, White et al. [533] used Word2Vec to learn embeddings for each buggy program revision. Ahmed et al. [10] used similar metrics for fixing compile-time errors. Additionally, Saha et al. [416] leveraged a code similarity analysis, which compares both syntactic and semantic features, and the revision history of a software project under examination, from Defects4J, for fixing multi-hunk bugs, i.e., bugs that require applying a substantially similar patch to different locations. Furthermore, Wang et al. [518] investigated, using similarity metrics, how these machine-generated correct patches can be semantically equivalent to human patches, and how bug characteristics affect patch generation. Sakkas et al. [419] also applied similarity metrics. Svyatkovskiy et al. [475] extracted structured representation of code (for example, lexemes, AST, and dataflow) and learn directly a task over those representations.

There are several approaches that use logic-based metrics based on the relationships of the features used. Specifically, Van Thuy et al. [506] extracted twelve relations of statements and blocks for Bi-gram model using Big code to prune the search space, and make the patches generated by Prophet [297] more efficient and precise. Alrajeh et al. [33] identified counterexamples and witness traces using model checking for logic-based learning to perform repair process automatically. Cai et al. [74] used publicly available examples of faulty models written in the B formal specification language, and proposed B-repair, an approach that supports automated repair of such a formal specification. Cambronero and Rinard [76] extracted dynamic program traces through identification of relevant AST of the target library; the extracted traces help the employed machine learning model to generate pipelines for new datasets.

Many studies also extract and consider the context where the bugs are related to. For instance, Tufano et al. [492] extracted Bug-Fixing Pairs (bfps) from millions of bug fixes mined from GitHub (used as meaningful examples of such bug-fixes), where such a pair consists of a buggy code component and the corresponding fixed code. Then, they used those pairs as input to an Encoder-Decoder Natural Machine Translation (nmt) model. For the extraction of the pair, they used the GumTree Spoon ast Diff tool [140]. Additionally, Soto and Le Goues [459] constructed a corpus by
delimiting debugging regions in a provided dataset. Then, they recursively analyzed the differences between the Simplified Syntax Trees associated with EditEvent’s. Mesbah et al. [335] also generated AST diffs from the textual code changes and transformed them into a domain-specific language called Delta that encodes the changes that must be made to make the code compile. Then, they fed the compiler diagnostic information (as source) and the Delta changes that resolved the diagnostic (as target) into a Neural Machine Translation network for training. Furthermore, Li et al. [267] used the prior bug fixes and the surrounding code contexts of the fixes for code transformation learning. Saha et al. [415] developed a ML model that relies on four features derived from a program’s context, i.e., the source-code surrounding the potential repair location, and the bug report. Similarly, Mashhadi and Hemmati [326] used a combination of natural language text and corresponding code snippet to generated an aggregated sequence representation for the downstream task. Finally, Bader et al. [46] utilized a ranking technique that also considers the context of a code change, and selects the most appropriate fix for a given bug. Vasic et al. [507] used results from localization of variable-misuse bugs. Wu et al. [534] developed an approach, CGB, for syntax-error correction that treats the code as a mixture of the token sequences and graphs. Lin et al. [276] and Zhu et al. [582] used AST paths to generate code embeddings to predict the correctness of a patch. Chakraborty et al. [82] represent the patches in a parse tree form and extract the necessary information (e.g., grammar rules, tokens, and token-types) from them. They used GumTree, a tree-based code differencing tool, to identify the edited AST nodes. To collect the edit context, their proposal, CODIT, converts the ASTS to their parse tree representation and extracts corresponding grammar rules, tokens, and token types.

**ML model training:** In the following, we present the main categories of ML techniques found in the examined papers.

**Neural Machine Translation:** This category includes papers that apply neural machine translation (NMT) for enhancing automated program repair. Such approaches can, for instance, include techniques that use examples of bug fixing for one programming language to fix similar bugs for other programming language. Lutellier et al. [305] developed the repair tool called CoCoNuT that uses ensemble learning on the combination of CNNs and a new context-aware NMT. Additionally, Tufano et al. [492] used NMT techniques (Encoder-Decoder model) for learning bug-fixing patches for real defects, and generated repair patches. Mesbah et al. [335] introduced DeepDelta, which used NMT for learning to repair compilation errors. Jiang et al. [211] proposed Cure, a NMT-based approach to automatically fix bugs. Pinconshi et al. [382] used SequenceR, a sequence-to-sequence model, to patch security faults in C programs. Zhu et al. [582] proposed a tool Recorder, a syntax-guided edit decoder that takes encoded information and produces placeholders by selecting non-terminal nodes based on their probabilities. Chakraborty et al. [82] developed a technique called CODIT that automates code changes for bug fixing using tree-based neural machine translation. In particular, they proposed a tree-based neural machine translation model, an extension of OpenNMT, to learn the probability distribution of changes in code.

**Natural Language Processing:** In this category, we include papers that combine natural language processing (NLP) techniques, embeddings, similarity scores, and ML for automated program repair. Tian et al. [483] carried out an empirical study to investigate different representation learning approaches for code changes to derive embeddings, which are amenable to similarity computations. This study uses BERT transformer-based embeddings. Furthermore, Amorim et al. [39] applied, a word embedding model (Word2Vec), to facilitate the evaluation of repair processes, by considering the naturalness obtained from known bug fixes. Van Thuy et al. [506] have also applied word representations, and extracted relations of statements and blocks for a Bi-gram model using Big code, to improve the existing learning-aid-based repair tool Prophet [297]. Gupta et al. [178] used word embeddings and reinforcement learning to fix erroneous C student programs with typographic errors.
Tian et al. [483] applied a ML predictor with BERT transformer-based embeddings associated with logistic regression to learn code representations in order to learn deep features that can encode the properties of patch correctness. Saha et al. [416] used similarity analysis for repairing bugs that may require applying a substantially similar patch at a number of locations. Additionally, Wang et al. [518] used also similarity metrics to compare the differences among machine-generated and human patches. Santos et al. [421] used n-grams and NNs to detect and correct syntax errors.

Logic-based rules: Alrajeh et al. [33] combined model checking and logic-based learning to support automated program repair. Cai et al. [74] also combined model-checking and ML for program repair. Shim et al. [444] used inductive program synthesis (DEEPER CODE), by creating a simple Domain Specific Language (DSL), and ML to generate computer programs that satisfies user requirements and specification. Sakkas et al. [419] combined type rules and ML (i.e., multi-class classification, DNNs, and MLP) for repairing compile errors.

Probabilistic predictions: Here, we list papers that use probabilistic learning and ML approaches such as association rules, Decision Tree, and Support Vector Machine to predict bug locations and fixes for automated program repair. Long and Rinard [297] introduced a repair tool called PROPHET, which uses a set of successful manual patches from open-source software repositories, to learn a probabilistic model of correct code, and generate patches. Soto and Le Goues [459] conducted a granular analysis using different statement kinds to identify those statements that are more likely to be modified than others during bug fixing. For this, they used simplified syntax trees and association rules. Gopinath et al. [161] presented a data-driven approach for fixing of bugs in database statements. For predicting the correct behavior for defect-inducing data, this study uses Support Vector Machine and Decision Tree. Saha et al. [415] developed the ELIXIR repair approach that uses Logistic Regression models and similarity-score metrics. Bader et al. [46] developed a repair approach called GETAFIX that uses hierarchical clustering to summarize fix patterns into a hierarchy ranging from general to specific patterns. Xiong et al. [537] introduced L2S that uses ML to estimate conditional probabilities for the candidates at each search step, and search algorithms to find the best possible solutions. Gopinath et al. [160] used Support Vector Machine and ID3 with path exploration to repair bugs in complex data structures. Le et al. [249] conducted an empirical study on the capabilities of program repair tools, and applied Random Forest to predict whether using genetic programming search in APR can lead to a repair within a desired time limit. Aleti and Martinez [16] used the most significant features as inputs to Random Forest, Support Vector Machine, Decision Tree, and multi-layer perceptron models.

Recurrent neural networks: DL approaches such as RNNs (e.g., LSTM and Transformer) have been used for synthesizing new code statements by learning patterns from a previous list of code statement, i.e., this techniques can be used to mainly predict the next statement. Such approaches often leverage word embeddings. Dantas et al. [118] combined Doc2Vec and LSTM, to capture dependencies between source code statements, and improve the fault-localization step of program repair. Ahmed et al. [10] developed a repair approach (TILDER) for fixing compilation errors using RNNs. Recently, Li et al. [267] introduced DLFix, which is a context-based code transformation learning for automated program repair. DLFix uses RNNs and treats automated program repair as code transformation learning, by learning patterns from prior bug fixes and the surrounding code contexts of those fixes. Syvatkovskiy et al. [474] presented INTELLICODE that uses a Transformer model that predicts sequences of code tokens of arbitrary types, and generates entire lines of syntactically correct code. Chen et al. [97] used the LSTM for synthesizing if–then constructs. Similarly, Vasic et al. [507] applied the LSTM in multi-headed pointer networks for jointly learning to localize and repair variable misuse bugs. Bhatia et al. [59] combined neural networks, and in particular RNNs, with constraint-based reasoning to repair syntax errors in buggy programs. Chen et al. [101] applied LSTM for sequence-to-sequence learning achieving end-to-end program repair through the SEQUENCE repair tool they developed. Majd et al. [313] developed SLDEEP, statement-level software defect prediction, which uses LSTM on static code features.
Apart from above-mentioned techniques, White et al. [533] developed DeepRepair, a recursive unsupervised deep learning-based approach, that automatically creates a representation of source code that accounts for the structure and semantics of lexical elements. The neural network language model is trained from the file-level corpus using embeddings.

### 3.3.2 Code generation

An automated code generation approach takes specification, typically in the form of natural language prompts, and generates executable code based on the specification [551, 395, 474]. We elaborate on the studies that involve generating source code using ML techniques.

**Dataset preparation:** Yin and Neubig [552] proposed a transition-based neural semantic parser, namely \texttt{tranx}, which generates formal meaning representation from natural language text. They used multiple datasets for their study—dataset proposed by Dong and Lapata [128] containing 880 geography-related questions, \texttt{Django} dataset [358], as well as \texttt{WikiSQL} dataset [576]. Similarly, Sun et al. [468] and Shin et al. [446] used the \texttt{HearthStone} dataset [283] for Python code generation; in addition, Shin et al. [446] used the \texttt{Spider} [557] dataset for training. Liang et al. [272] used the semantic parsing dataset \texttt{WebQuestionsSP} [550] consisting of question-answer pairs for training and 1,639 for testing. Bielik et al. [60] used the \texttt{Linux Kernel} dataset [222], and the \texttt{Hutter Prize} \texttt{Wikipedia} dataset. Devlin et al. [122] evaluated their architecture on 205 real-world \texttt{Flash-Fill} instances [170]. Xiong et al. [537] used training data stemming from two \texttt{Defects4J} projects and their related JDK packages. Wei et al. [530] conducted experiments on Java and Python projects collected from \texttt{GitHub} used by previous work (such as by Hu et al. [198], Hu et al. [199], Wan et al. [511]).

Some studies curated datasets for their experiments. For example, Chen et al. [93] created \texttt{HumanEval}, a dataset containing 164 programming problems crafted manually for evaluation. Similarly, Li et al. [270] first used a curated set of public \texttt{GitHub} repositories implemented in several popular languages such as C++, C#, Java, Go, and Python for pre-training. They created a dataset, \texttt{CodeContests}, for fine-tuning. The dataset includes problems, solutions, and test cases scraped from the Codeforces platform. Furthermore, \texttt{IntelliCode} [474] is trained on 1.2 billion lines of source code written in the Python, C#, JavaScript and TypeScript programming languages. Allahmanis et al. [28] evaluated their models on a large dataset of 2.9 million lines of code. Cai et al. [75] used a training set that contains 200 traces for addition, 100 traces for bubble sort, 6 traces for topological sort, and 4 traces for quicksort. Devlin et al. [121] used programming examples that involve induction, such as I/O examples. Shu and Zhang [449] used training data to generate programs at various levels of complexity according to 45 predefined tasks (e.g., Split, Join, Select). Murali et al. [344] used a corpus of about 150,000 API-manipulating Android methods. Shin et al. [447] propose a new approach to generate desirable distribution for the target datasets for program induction and synthesis tasks.

**Feature extraction:** Studies in this category extensively used \texttt{AST} during the feature extraction step. \texttt{tranx} [552] maps natural language text into an \texttt{AST} using a series of tree-construction actions. Similarly, Sun et al. [468] parsed a program as an \texttt{AST} and decomposed the program into several context-free grammar rules. Also, the study by Yin and Neubig [551] transformed statements to \texttt{AST}s. These \texttt{AST}s are generated for all well-formed programs using parsers provided by the programming language under examination. Furthermore, Rabinovich et al. [395] developed a model that used a modular decoder, whose sub-models are composed using natively generated \texttt{AST}s. Each sub-model is associated with a specific construct in the \texttt{AST} grammar, and, then, it is invoked when that construct is required in the output tree.

Some studies in the category used examples of input and output to learn code generation. \textit{Euphony} [257] learns good representation using easily obtainable solutions for given programs. \textit{DeepCoder} [474] observes inputs and outputs, by leveraging information from interpreters. Then,
DeepCoder searches for a program that matches the input-output examples. Similarly, Chen et al. [99] developed a neural program synthesis from input-output examples. Shu and Zhang [449] extracted features from string transformations, i.e., input-output strings, and use the learned features to induce correct programs. Devlin et al. [122] used I/O programming examples and developed a DSL for synthesizing related programs.

Finally, the rest of the studies used tokens from source code as their features. For example, Chen et al. [97] and Li et al. [270] extracted tokens from source code. Allamanis et al. [28] extracted features that refer to program semantics such as variable names. Xiong et al. [537] extracted several features, including context, variable, expression, and position features, from the source code to train their ML models. Devlin et al. [121] focused on extracting features from programs that involve induction. Murali et al. [344] extracted low-level features (e.g., API calls). Liang et al. [272] also used tokens and graphs extracted from the data sets used. Shin et al. [446] considered idioms (new named operators) from programs in an extended grammar. Bielik et al. [60] leveraged language features, using datasets of ngrams in their experiments. Maddison and Tarlow [310] considered features of variables and structural language features. Cummins et al. [113] used language features to synthesize human-like written programs. Shin et al. [447] used different features related to I/O operations e.g., program size, control-flow ratio, and so on. Chen et al. [98] extracted features from programming-language arguments. Wei et al. [530] leveraged the power of code summarization and code generation. The input of code summarization is the output of code generation; the approach applies the relations between these tasks and proposes a dual training framework to train these tasks simultaneously using probability and attention weights along with dual constraints.

ML model training: A majority of the studies in this category relies on the RNN-based encoder-decoder architecture. Trans [552] implemented a transition system that generates an AST from a sequence of tree-constructing actions. The system is based on a LSTM-based encoder-decoder model where the encoder encodes the input tokens into its corresponding vector representation and the decoder generates the probabilities of tree-constructing actions. Also, Yin and Neubig [551] proposed a data-driven syntax-based neural network model for generation of code in general-purpose programming languages such as Python. Cai et al. [75] implemented recursion in the Neural Programmer-Interpreter framework that uses an LSTM controller on four tasks: grade-school addition, bubble sort, topological sort, and quicksort. Bielik et al. [60] designed a language TChar for character-level language modeling, and program synthesis using LSTM. Cummins et al. [113] applied LSTM to synthesize compilable, executable benchmarks. Chen et al. [98] used reinforcement learning to predict arguments (e.g., CALL, REDUCE). Devlin et al. [122] presented a novel variant of the attentional RNN architecture, which allows for encoding of a variable size set of input-output examples. Wei et al. [530] used Seq2Seq, Bi-LSTM, LSTM-based models to exploit the code summarization and code generation for automatic software development. Furthermore, Rabinovich et al. [395] introduced Abstract Syntax Networks (ASNs), an extension of the standard encoder-decoder framework.

Some of the studies employed transformer-based models. Sun et al. [468] proposed TreeGen for code generation. They implemented an AST reader to combine the grammar rules with AST and mitigated the long-dependency problem with the help of the attention mechanism used in Transformers. Similarly, Li et al. [270] implemented a transformer architecture for AlphaCode. Chen et al. [93] proposed Codex that is a GPT model fine-tuned on publicly available code from GitHub containing up to 12B parameters on code. IntelliCode by Svyatkovskiy et al. [474] is a multilingual code completion tool that predicts sequences of code tokens of arbitrary types. IntelliCode is also able to generate entire lines of syntactically correct code. It uses a generative transformer model.

Euphony [257] targets a standard formulation, syntax-guided synthesis, by extending the grammar of given programs. To do so, Euphony uses a probabilistic model dictating the likelihood of each program. DeepCoder [47] leverages gradient-based optimization and integrates neural network architectures with search-based techniques. Szydlo et al. [477] investigated the concept of
source code generation of machine learning models as well as the generation algorithms for commonly used ML methods. Chen et al. [99] introduced a technique that is based on execution-guided synthesis and uses a synthesizer ensemble. This approach leverages semantic information to ensemble multiple neural program synthesizers. Chen et al. [97] used latent attention to compute token weights. They found that latent attention performs better in capturing the sentence structure. Allamanis et al. [28] used DL models to learn semantics from programs. They used the code’s graph structure and learned program representations over the generated graphs. Xiong et al. [537] applied the gradient boosting tree algorithm to train their models. Devlin et al. [121] used the transfer learning and k-shot learning approach for cross-task knowledge transfer to improve program induction in limited-data scenarios. Shu and Zhang [449] proposed NPBE (Neural Programming by Example) that teaches a DNN to compose a set of predefined atomic operations for string manipulations. Murali et al. [344] trained a neural generator on program sketches to generate source code in a strongly typed, Java-like programming language. Liang et al. [272] introduced the Neural Symbolic Machine (NSM), based on a sequence-to-sequence neural network induction, and apply it to semantic parsing. Shin et al. [446] employed non-parametric Bayesian inference to mine the code idioms that frequently occur in a given corpus and trained a neural generative model to optionally emit named idioms instead of the original code fragments. Maddison and Tarlow [310] used models that are based on probabilistic context-free grammars (PCFGs) and a neuro-probabilistic language, which are extended to incorporate additional source code-specific structures.

### 3.3.3 Program translation

In this section, we list studies that use ML that can be used, for instance, for translating source code from one programming language to another by learning source-code patterns. Le et al. [248] presented a survey on DL techniques including machine translation algorithms and applications. Oda et al. [357] used statistical machine translation (SMT) and proposed a method to automatically generate pseudo-code from source code for source-code comprehension. To evaluate their approach they conducted experiments, and generated English or Japanese pseudo-code from Python statements using SMT. Then, they found that the generated pseudo-code is mostly accurate, and it can facilitate code understanding. Roziere et al. [408] applied unsupervised machine translation to create a transcompiler in a fully unsupervised way. TransCoder uses beam search decoding to generate multiple translations. Phan and Jannesari [380] proposed PrefixMap, a code suggestion tool for all types of code tokens in the Java programming language. Their approach uses statistical machine translation that outperforms NMT. They used three corpora for their experiments—a large-scale corpus of English-German translation in NLP [304], the Conala corpus [553], which contains Python software documentation as 116,000 English sentences, and the MSR 2013 corpus [23].

### 3.4 Quality assessment

The quality assessment category has sub-categories code smell detection, clone detection, and quality assessment/prediction. In this section, we elaborate upon the state-of-the-art related to each of these categories within our scope.

#### 3.4.1 Code smell detection

Code smells impair the code quality and make the software difficult to extend and maintain [435]. Extensive literature is available on detecting smells automatically [435]; ML techniques have been used to classify smelly snippets from non-smelly code. First, source code is pre-processed to extract individual samples (such as a class, file, or method). These samples are classified into positive and negative samples. Afterwards, relevant features are identified from the source code and those features are then fed into an ML model for training. The trained model classifies a source code sample into a smelly or non-smelly code.
Dataset preparation: The process of identifying code smells requires a dataset as a ground truth for training an ML model. Each sample of the training dataset must be tagged appropriately as smelly sample (along with target smell types) or non-smelly sample. Many authors built their datasets tagged manually with annotations. For example, Fakhoury et al. [139] developed a manually validated oracle containing 1,700 instances of linguistic smells. Pecorelli et al. [375] created a dataset of 8.5 thousand samples of smells from 13 open-source projects. Some authors [11, 336, 110, 206, 180] employed existing datasets (Landfill and Qualitas) in their studies. Tummalapalli et al. [500, 497, 499] used 226 WSDL files from the tera-PROMISE dataset. Oliveira et al. [360] relied on historical data and mined smell instances from history where the smells were refactored.

Some efforts such as one by Sharma et al. [437] used CodeSplit [434, 433] first to split source code files into individual classes and methods. Then, they used existing smell detection tools [436, 432] to identify smells in the subject systems. They used the output of both of these tasks to identify and segregate positive and negative samples. Similarly, Kaur and Kaur [226] used smells identified by Dr Java, EMMA, and FindBugs as their gold-set. Alazba and Aljamaan [14] and Dewangan et al. [124] used the dataset manually labelled instances detected by four code smell detector tools (i.e., iPlasma, PMD, Fluid Tool, Anti-Pattern Scanner, and Marinescu’s detection rule). The dataset labelled six code smells collected from 74 software systems. Zhang and Dong [569] proposed a large dataset BrainCode consisting 270,000 samples from 20 real-world applications. The study used iPlasma to identify smells in the subject systems.

Liu et al. [290] adopted an usual mechanism to identify their positive and negative samples. They assumed that popular well-known open-source projects are well-written and hence all of the classes/methods of these projects are by default considered free from smells. To obtain positive samples, they carried out reverse refactoring e.g., moving a method from a class to another class to create an instance of feature envy smell.

Feature extraction: The majority of the articles [52, 223, 240, 174, 8, 360, 390, 149, 42, 148, 481, 111, 38, 114, 336, 290, 179, 495, 110, 500, 417, 497, 499, 226, 176, 124, 14, 206, 569, 173] in this category use object-oriented metrics as features. These metrics include class-level metrics (such as lines of code, lack of cohesion among methods, number of methods, fan-in and fan-out) and method-level metrics (such as parameter count, lines of code, cyclomatic complexity, and depth of nested conditional). We observed that some of the attempts use a relatively small number of metrics (Thongkum and Mekruksavanich [481] and Agnihotri and Chug [8] used 10 and 16 metrics, respectively). However, some of the authors chose to experiment with a large number of metrics. For example, Amorim et al. [38] employed 62, Mhawish and Gupta [336] utilized 82, and Arcelli Fontana and Zanoni [42] used 63 class-level metrics and 84 method-level metrics.

Some efforts diverge from the mainstream usage of using metrics as features and used alternative features. Lujan et al. [303] used warnings generated from existing static analysis tools as features. Similarly, Ochodek et al. [356] analyzed individual lines in source code to extract textual properties such as regex and keywords to formulate a set of vocabulary based features (such as bag of words). Tummalapalli et al. [498] and Gupta et al. [175] used distributed word representation techniques such as Term frequency-inverse Document Frequency (TFIDF), Continuous Bag Of Words (CBW), Global Vectors for Word Representation (GloVe), and Skip Gram. Similarly, Hadj-Kacem and Bouassida [180] generated AST first and obtain the corresponding vector representation to train a model for smell detection. Furthermore, Sharma et al. [437] hypothesized that DL methods can infer the features by themselves and hence explicit feature extraction is not required. They did not process the source code to extract features and feed the tokenized code to ML models.

ML model training: The type of ML models usage can be divided into three categories.

Traditional ML models: In the first category, we can put studies that use one or more traditional ML.
models. These models include Decision Tree, Support Vector Machine, Random Forest, Naive Bayes, Logistic Regression, Linear Regression, Polynomial Regression, Bagging, and Multilayer Perceptron. The majority of studies [303, 240, 174, 8, 360, 390, 149, 148, 374, 481, 111, 127, 114, 495, 110, 498, 499, 226, 124, 14, 175, 206, 180, 173] in this category compared the performance of various ML models. Some of the authors experimented with individual ML models; for example, Kaur et al. [223] and Amorim et al. [38] used Support Vector Machine and Decision Tree, respectively, for smell detection.

**Ensemble methods:** The second category of studies employed ensemble methods to detect smells. Barbez et al. [52] and Tummalapalli et al. [496] experimented with ensemble techniques such as majority training ensemble and best training ensemble. Saidani et al. [417] used the Ensemble Classifier Chain (ECC) model that transforms multi-label problems into several single-label problems to find the optimal detection rules for each anti-pattern type.

**DL-based models:** Studies that use DL form the third category. Sharma et al. [437] used CNN, RNN (LSTM), and autoencoders-based ML models. Hadj-Kacem and Bouassida [179] employed autoencoder-based DL model to first reduce the dimensionality of data and Artificial Neural Network to classify the samples into smelly and non-smelly instances. Liu et al. [290] deployed four different DL models based on CNN and RNN. It is common to use other kinds of layers (such as embeddings, dense, and dropout) along with CNN and RNN. Gupta et al. [176] used eight DL models and Zhang and Dong [569] proposed Metric-Attention-based Residual network (MAR) to detect brain class/method. MARS used metric-attention mechanism to calculate the weight of code metrics and detect code smells.

**Discussion:** A typical ML model trained to classify samples into either smelly or non-smelly samples. The majority of the studies focused on a relatively small set of known code smells—god class [52, 303, 223, 174, 8, 360, 149, 167, 42, 111, 78, 179], feature envy [52, 223, 8, 149, 42, 148, 111, 437, 179], long method [223, 174, 149, 167, 42, 148, 111, 45, 179], data class [223, 360, 149, 167, 42, 148], complex class [303, 174, 360]. Results of these efforts vary significantly; F1 score of the ML models vary between 0.3 to 0.99. Among the investigated ML models, authors widely report that Decision Tree [45, 148, 13, 174] and Random Forest [45, 148, 240, 42, 336] perform the best. Other methods that have been reported better than other ML models in their respective studies are Support Vector Machine [496], Boosting [302], and autoencoders [437].

Traditional ML techniques are the prominent choice in this category because these techniques work well with fixed size, fixed column meaning vectors. Code quality metrics capture the features relevant to the identification of smells, and they have fixed size, fixed column meaning vectors. However, such vectors do not capture subjectivity inherent in the context and hence some studies rely on alternative features such as embeddings generated by AST representations to feed DL models such as RNN.

### 3.4.2 Code clone detection

Code clone detection is the process of identifying duplicate code blocks in a given software system. Software engineering researchers have proposed not only methods to detect code clones automatically, but also verify whether the reported clones from existing tools are false-positives or not using ML techniques. Studies in this category prepare a dataset containing source code samples classified as clones or non-clones. Then, they apply feature extraction techniques to identify relevant features that are fed into ML models for training and evaluation. The trained models identify clones among the sample pairs.

**Dataset preparation:** Manual annotation is a common way to prepare a dataset for applying ML to identify code clones [340, 341, 532]. Mostaeno et al. [340] used a set of tools (NiCad, Deckard, iClones, CCFinderX and SourcererCC) to first identify a list of code clones; they then manually validated each of the identified clone set. Yang et al. [542] used existing code clone detection tools to generate their training set. Some authors (such as Bandara and Wijayarathna [49] and Hammad et al. [183]) relied on existing code-clone datasets. Zhang and Khoo [562] used NiCad to detect all clone groups from each version of the software. The study mapped the clones from a consecu-
tive version and used the mapping to predict clone consistency at both the clone-creating and the clone-changing time. Bui et al. [72] deployed an interesting mechanism to prepare their code-clone dataset. They crawled through GitHub repositories to find different implementations of sorting algorithms; they collected 3,500 samples from this process.

**Feature extraction:** The majority of the studies relied on the textual properties of the source code as features. Bandara and Wijayarathna [49] identified features such as the number of characters and words, identifier count, identifier character count, and underscore count using the antlr tool. Some studies [340, 341, 339] utilized line similarity and token similarity. Yanget al. [542] and Hammad et al. [183] computed tf-idf along with other metrics such as position of clones in the file. Cesare et al. [79] extracted 30 package-level features including the number of files, hashes of the files, and common filenames as they detected code clones at the package level. Zhang and Khoo [562] obtained a set of code attributes (e.g., lines of code and the number of parameters), context attribute set (e.g., method name similarity, and sum of parameter similarity). Similarly, Sheneamer and Kalita [441] obtained metrics such as the number of constructors, number of field access, and super-constructor invocation from the program ast. They also employed program dependence graph features such as decl_assign and control_decl. Along the similar lines, Zhao and Huang [571] used cfg and dfg (Data Flow Graph) for clone detection. Some of the studies [72, 532, 142] relied on dl methods to encode the required features automatically without specifying an explicit set of features.

**ML model training:**

**Traditional ml models:** The majority of studies [341, 49, 339, 441, 562] experimented with a number of ml approaches. For example, Mostaeen et al. [341] used Bayes Network, Logistic Regression, and Decision Tree; Bandara and Wijayarathna [49] employed Naive Bayes, K Nearest Neighbors, AdaBoost. Similarly, Sheneamer and Kalita [441] compared the performance of Support Vector Machine, Linear Discriminant Analysis, Instance-Based Learner, Lazy K-means, Decision Tree, Naive Bayes, Multilayer Perceptron, and Logit Boost.

**dl-based models:** dl models such as ann [340, 339], dnn [142, 571], and rnn with Reverse neural network [532] are also employed extensively. Bui et al. [71] and Bui et al. [72] combined neural networks for ml models’ training. Specifically, Bui et al. [71] built a Bilateral neural network on top of two underlying sub-networks, each of which encodes syntax and semantics of code in one language. Bui et al. [72] constructed BitBCNN—a combination layer of sub-networks to encode similarities and differences among code structures in different languages. Hammad et al. [183] proposed a Clone-Advisor, a dnn model trained by fine-tuning GPT-2 over the BigCloneBench code clone dataset, for predicting code tokens and clone methods.

### 3.4.3 Defect prediction

To pinpoint bugs in software, researchers used various ml approaches. The first step of this process is to identify the positive and negative samples from a dataset where samples could be a type of source code entity such as classes, modules, files, and methods. Next, features are extracted from the source code and fed into an ml model for training. Finally, the trained model can classify different code snippets as buggy or benign based on the encoded knowledge. To this end, we discuss the collected studies based on (1) data labeling, (2) features extract, and (3) ml model training.

**Dataset preparation:** To train an ml model for predicting defects in source code a labeled dataset is required. For this purpose, researchers have used some well-known and publicly available datasets. For instance, a large number of studies [80, 157, 316, 454, 85, 58, 320, 453, 81, 517, 106, 265, 125, 386, 307, 229, 90, 116, 520, 442, 129, 455, 568, 73, 126, 423, 521, 281, 404, 263, 224, 359, 246, 457, 366, 318, 393, 323, 470, 137, 365, 554, 469, 120, 12, 15] used the promise dataset [424]. Some studies used other datasets in addition to promise dataset. For example, Liang et al. [273]
used Apache projects and Qiao et al. [393] used MIS dataset [306]. Xiao et al. [535] utilized a Continuous Integration (CI) dataset and Pradel and Sen [387] generated a synthetic dataset. Apart from using the existing datasets, some other studies prepared their own datasets by utilizing various GitHus projects [314, 190, 455, 7, 315, 372, 491] including Apache [266, 64, 117, 141, 364, 460, 317, 105, 400], Eclipse [583, 117] and Mozilla [311, 233] projects, or industrial data[64].

Feature extraction: The most common features to train a defect prediction model are the source code metrics introduced by Halstead [182], Chidamber and Kemerer [103], and McCabe [328]. Most of the examined studies [80, 157, 316, 454, 85, 320, 517, 106, 314, 315, 307, 229, 73, 86, 233, 427, 141, 224, 217, 359, 246, 41, 21, 457, 522, 318, 393, 323, 469, 554, 470, 120, 105, 137, 400, 12, 364, 460, 388, 317, 15, 372, 488] used a large number of metrics such as Lines of Code, Number of CI cycles, with a CI environment, to train defect prediction models. Some authors [365, 456] combined detected code smells with code quality metrics. Furthermore, Felix and Lee [144] used defect metrics such as defect density and defect velocity along with traditional code smells.

In addition to the above, some authors [81, 125, 58, 386] suggested the use of dimensional space reduction techniques—such as Principal Component Analysis (PCA)—to limit the number of features. Pandey and Gupta [367] used Sequential Forward Search (SFS) to extract relevant source code metrics. Most authors [129] suggested a sampling-based approach to extract source code metrics to train defect prediction models. Kaur et al. [225] suggested an approach to fetch entropy of change metrics. Bowes et al. [64] introduced a novel set of metrics constructed in terms of mutants and the test cases that cover and detect them.

Other authors [387, 568] used embeddings to train models. Such studies, first generate ASTS [266, 141, 263, 366, 273], a variation of ASTS such as simplified ASTS [281, 88], or AST-diff [521, 491] for a selected method or file could be considered. Then, embeddings are generated either using the token vector corresponding to each node in the generated tree or extracting a set of paths from an AST. Singh et al. [455] proposed a method named Transfer Learning Code Vectorizer that generates features from source code by using a pre-trained code representation ML model. Another approach for detecting defects is capturing the syntax and multiple levels of semantics in the source code as suggested by Dam et al. [116]. To do so, the authors trained a tree-base LSTM model by using source code files as feature vectors. Subsequently, the trained model receives an AST as input and predicts if a file is clear from bugs or not.

Wang et al. [520] employed the Deep Belief Network algorithm (DBN) to learn semantic features from token vectors, which are fetched from applications’ ASTS. Shi et al. [442] used a DNN model to automate the features extraction from the source code. Xiao et al. [535] collected the testing history information of all previous CI cycles, within a CI environment, to train defect predict models. Likewise to the above study, Madhavan and Whitehead [311] and Aggarwal [7] used the changes among various versions of a software as features to train defect prediction models.

In contrast to the above studies, Chen et al. [90] suggested the DTL-DP, a framework to predict defects without the need of features extraction tools. Specifically, DTL-DP visualizes the programs as images and extracts features out of them by using a self-attention mechanism [508]. Afterwards, it utilizes transfer learning to reduce the sample distribution differences between the projects by feeding them to a model.

ML model training: In the following, we present the main categories of ML techniques found in the examined papers.

Traditional ML models: To train models, most of the studies [80, 157, 316, 454, 85, 58, 320, 453, 81, 106, 125, 386, 314, 315, 184, 367, 129, 455, 229, 225, 73, 520, 393, 323, 469, 554, 470, 120, 105, 400, 364, 460, 456, 388, 317, 15, 372, 224, 359, 246, 144, 318, 457, 21, 404] used traditional ML algorithms such as Decision Tree, Random Forest, Support Vector Machine, and AdaBoost. Similarly, Jing et al. [217], Wang et al. [522] used Cost Sensitive Discriminative Learning. In addition, other authors [265, 517, 307] proposed changes to traditional ML algorithms to train their mod-
els. Specifically, Wang and Yao [517] suggested a dynamic version of AdaBoost.NC that adjusts its parameters automatically during training. Similarly, Li et al. [265] proposed ACoForest, an active semi-supervised learning method to sample the most useful modules to train defect prediction models. Ma et al. [307] introduced Transfer Naive Bayes, an approach to facilitate transfer learning from cross-company data information and weighting training data.

**dl-based models:** In contrast to the above studies, researchers [90, 116, 387, 266, 427] used dl models such as cnn and rnn-based models for defect prediction. Specifically, Chen et al. [90], Al Qasem et al. [12], Li et al. [263], Pan et al. [366] used cnn-based models to predict bugs. rnn-based methods [116, 491, 88, 273, 141, 281] are also frequently used where variations of lstm are used to defect prediction. Moreover, by using dl approaches, authors achieved improved accuracy for defect prediction and they pointed out bugs in real-world applications [387, 266].

3.4.4 Quality assessment/prediction

Studies in this category assess or predict issues related to various quality attributes such as reliability, maintainability, and run-time performance. The process starts with dataset pre-processing and labeling to obtain labeled data samples. Feature extraction techniques are applied on the processed samples. The extracted features are then fed into an ml model for training. The trained model assesses or predicts the quality issues in the analyzed source code.

**Dataset preparation:** Heo et al. [193] generated data to train an ml model in pursuit to balance soundness and relevance in static analysis by selectively allowing unsoundness only when it is likely to reduce false alarms. Similarly, Alik hashashne et al. [20] used the Understand tool to detect various metrics, and employed them on the Juliet test suite for C++. Reddivari and Raman [402] extracted a subset of data belonging to open source projects such as Ant, Tomcat, and Jedit to predict reliability and maintainability using ml techniques. Malhotra and Chug [321] also prepared a custom dataset using two proprietary software systems as their subjects to predict maintainability of a class.

**Feature extraction:** Heo et al. [193] extracted 37 low-level code features for loop (such as number of null, array accesses, and number of exits) and library call constructs (such as parameter count and whether the call is within a loop). Some studies [20, 402, 321] used source code metrics as features.

**ML model training:** Alik hashashne et al. [20] employed Random Forest, Support Vector Machine, K Nearest Neighbors, and Decision Tree to classify static code analysis tool warnings as true positives, false positives, or false negatives. Reddivari and Raman [402] predicted reliability and maintainability using the similar set of ml techniques. Anomaly-detection techniques such as One-class Support Vector Machine have been used by Heo et al. [193]. They applied their method on taint analysis and buffer overflow detection to improve the recall of static analysis. Whereas, some other studies [20] aimed to rank and classify static analysis warnings.

3.5 Code completion

Code auto-completion is a state-of-the-art integral feature of modern source-code editors and IDES [69]. The latest generation of auto-completion methods uses NLP and advanced ml models, trained on publicly available software repositories, to suggest source-code completions, given the current context of the software-projects under examination.

**Dataset preparation:** The majority of the studies mined a large number of repositories to construct their own datasets. Specifically, Gopalakrishnan et al. [158] examined 116,000 open-source systems to identify correlations between the latent topics in source code and the usage of architectural developer tactics (such as authentication and load-balancing). Han et al. [185], Han et al. [186] trained and tested their system by sampling 4,919 source code lines from open-source projects. Raychev et al. [401] used large codebases from GitHub to make predictions for JavaScript
and Python code completion. Svyatkovskiy et al. [473] used 2,700 Python open-source software
GitHub repositories for the evaluation of their novel approach, Pythia.

The rest of the approaches employed existing benchmarks and datasets. Rahman et al. [398]
trained their proposed model using the data extracted from Aizu Online Judge (AOJ) system. Liu et al.
[289], Liu et al. [288] performed experiments on three real-world datasets to evaluate the effectiveness
of their model when compared with the state-of-the-art approaches. Li et al. [264] conducted
experiments on two datasets to demonstrate the effectiveness of their approach consisting of an
attention mechanism and a pointer mixture network on code completion tasks. Schuster et al.
[426] used a public archive of GitHub from 2020 [1].

**Feature extraction:** Studies in this category extract source code information in variety of forms.
Gopalakrishnan et al. [158] extracted relationships between topical concepts in the source code
and the use of specific architectural developer tactics in that code. Liu et al. [289], Liu et al. [288]
introduced a self-attentional neural architecture for code completion with multi-task learning. To
achieve this, they extracted the hierarchical source code structural information from the programs
considered. Also, they captured the long-term dependency in the input programs, and derived
knowledge sharing between related tasks. Li et al. [264] used locally repeated terms in program
source code to predict out-of-vocabulary (OoV) words that restrict the code completion. Chen and
Wan [92] proposed a tree-to-sequence (Tree2Seq) model that captures the structure information
of source code to generate comments for source code. Raychev et al. [401] used ASTs and performed
prediction of a program element on a dynamically computed context. Svyatkovskiy et al.
[473] introduced a novel approach for code completion called Pythia, which exploits state-of-the-art
large-scale DL models trained on code contexts extracted from ASTs.

**ML model training:** The studies can be classified based on the used ML technique for code com-
pletion.

**Recurrent Neural Networks:** For code completion, researchers mainly try to predict the next token.
Therefore, most approaches use RNNs. In particular, Terada and Watanobe [479] used LSTM for
code completion to facilitate programming education. Rahman et al. [398] also used LSTM. Wang
et al. [519] used an LSTM-based neural network combined with several techniques such as Word
Embedding models and Multi-head Attention Mechanism to complete programming code. Zhong
et al. [575] applied several DL techniques, including LSTM, Attention Mechanism (AM), and Sparse
Point Network (SPN) for JavaScript code suggestions.

Apart from LSTM, researchers have used RNN with different approaches to perform code sug-
gestions. Li et al. [264] applied neural language models, which involve attention mechanism for RNN,
by learning from large codebases to facilitate effective code completion for dynamically-typed pro-
gramming languages. Hussain et al. [202] presented CodeGRU that uses GRU for capturing source
codes contextual, syntactical, and structural dependencies. Yang et al. [545] presented REP to im-
prove language modeling for code completion. Their approach uses learning of general token rep-
etition of source code with optimized memory, and it outperforms LSTM. Schumacher et al. [425]
combined neural and classical ML including RNNs, to improve code recommendations.

**Probabilistic Models:** Earlier approaches for code completion used statistical learning for recom-
mending code elements. In particular, Gopalakrishnan et al. [158] developed a recommender sys-
tem using prediction models including neural networks for latent topics. Han et al. [185], Han et al.
[186] applied Hidden Markov Models to improve the efficiency of code-writing by supporting code
completion of multiple keywords based on non-predefined abbreviated input. Proksch et al. [391]
used Bayesian Networks for intelligent code completion. Raychev et al. [401] utilized a probabilistic
model for code in any programming language with Decision Tree. Svyatkovskiy et al. [473] proposed
Pythia that employs a Markov Chain language model. Their approach can generate ranked lists of
methods and API recommendations, which can be used by developers while writing programs.

**Other techniques:** Recently, new approaches have been developed for code completion based on
multi-task learning, code representations, and NMT. For instance, Liu et al. [289], Liu et al. [288] applied Multi-Task Learning (MTL) for suggesting code elements. Lee et al. [256] developed MergeLog-1023, a DL-based merged network that uses code representations for automated logging decisions. Chen and Wan [92] applied Tree2Seq model with NMT techniques for code comment generation.

### 3.6 Program Comprehension

Program comprehension techniques attempt to understand the theory of comprehension process of developers as well as the tools, techniques, and processes that influence the comprehension activity [463]. We summarized, in the rest of the section, program comprehension studies into four sub-categories i.e., code summarization, program classification, change analysis, and entity identification/recommendation.

#### 3.6.1 Code summarization

Code summarization techniques attempt to provide a consolidated summary of the source code entity (typically a method). A variety of attempts has been made in this direction. The majority of the studies [94, 252, 285, 9, 443, 548, 198, 260, 516, 253, 549, 523, 565, 204, 268, 580, 188, 581] produces a summary for a small block (such as a method). This category also includes studies that summarize small code fragments [347], code folding within IDEs [510], commit message generation [212, 295, 214, 213, 96, 526], and title generation for online posts from code [151].

**Dataset preparation:** The majority of the studies [26, 94, 252, 285, 9, 198, 95, 260, 516, 511, 523, 96, 581] in this category prepares pairs of code snippets and their corresponding natural language description. Specifically, Chen and Zhou [94] used more than 66 thousand pairs of C# code and natural language description where source code is tokenized using a modified version of the ANTLR parser. Ahmad et al. [9] conducted their experiments on a dataset containing Java and Python snippets; sequences of both the code and summary tokens are represented by a sequence of vectors. Hu et al. [198] and Li et al. [260] prepared a large dataset from 9,714 GitHub projects. Similarly, Wang et al. [516] mined code snippets and corresponding javadoc comments for their experiment. Chen et al. [95] created their dataset from 12 popular open-source Java libraries with more than 10 thousand stars. They considered method bodies as their inputs and method names along with method comments as prediction targets. Psarras et al. [392] prepared their dataset by using Weka, SystemML, DL4J, Mahout, Neuroph, and Spark as their subject systems. The authors retained names and types of methods, and local and class variables. Choi et al. [104] collected and refined more than 114 thousand pairs of methods and corresponding code annotations from 100 open-source Java projects. Iyer et al. [204] mined StackOverflow and extracted title and code snippet from posts that contain exactly one code snippet. Similarly, Gao et al. [151] used a dump of StackOverflow dataset. They tokenized code snippets with respect to each programming language for pre-processing. The common steps in preprocessing identifiers include making them lower case, splitting the camel-cased and underline identifiers into sub-tokens, and normalizing the code with special tokens such as "VAR" and "NUMBER". Nazar et al. [347] used human annotators to summarize 127 code fragments retrieved from Eclipse and NetBeans official frequently asked questions. Yang et al. [546] built a dataset with over 300K pairs of method and comment to evaluate their approach. Chen et al. [96] used dataset provided by Hu et al. [198] and manually categorized comments into six intention categories for 20,000 code-comment pairs. Wang et al. [526] created a Python dataset that contains 128 thousand code-comment pairs. Zhou et al. [579] crawled over 6700 Java projects from Github to extract their methods and the corresponding javadoc comments to create their dataset.

Jiang [213] used 18 popular Java projects from GitHub to prepare a dataset with approximately 50 thousand commits to generate commit messages automatically. Liu et al. [292] processed 56 popular open-source projects and selected approximately 160K commits after filtering out the irrelevant commits. Liu et al. [296] used RepoRepear to identify Java repositories to process. They
collected pull-request meta data by using GitHub APIs. After preprocessing the collected information, they trained a model to generate pull request description automatically. Wang et al. [515] prepared a dataset of 107K commits by mining 10K open-source repositories to generate context-aware commit messages.

Apart from source code, some of the studies used additional information generated from source code. For example, LeClair et al. [252] used AST along with code and their corresponding summaries belonging to more than 2 million Java methods. Likewise, Shido et al. [443] and Zhang et al. [565] also generated ASTs of the collected code samples. Liu et al. [285] utilized call dependencies along with source code and corresponding comments from more than a thousand GitHub repositories. LeClair et al. [253] employed AST along with adjacency matrix of AST edges.

Some of the studies used existing datasets such as StaQC [547] and the dataset created by Jiang et al. [212]. Specifically, Liu et al. [295], Jiang and McMillan [214] utilized a dataset of commits provided by Jiang et al. [212] that contains two million commits from one thousand popular Java projects. Yao et al. [548] and Ye et al. [549] used StaQC dataset [547]; it contains more than 119 thousand pairs of question title and code snippet related to SQL mined from StackOverflow. Xie et al. [536] utilized two existing datasets—one each for Java [251] and Python [53]. Bansal et al. [51] evaluated their code summarization technique using a Java dataset of 2.1M Java methods from 28K projects created by LeClair and McMillan [251]. Li et al. [268] also used the Java dataset of 2.1M methods LeClair and McMillan [251] to predict the inconsistent names from the implementation of the methods. Similarly, Haque et al. [188], LeClair et al. [254], Haque et al. [189] relied on the Java dataset by LeClair and McMillan [251] for summarizing methods. Zhou et al. [580] combined multiple datasets for their experiment. The first dataset [198] contains over 87 thousand Java methods. The other datasets contained 2.1M Java methods [251] and 500 thousand Java methods respectively.

Efforts in the direction of automatic code folding also utilize techniques similar to code summarization. Viuginov and Filchenkov [510] collected projects developed using IntelliJ platform. They identified the foldable and FoldingDescription elements from workspace.xml belonging to 335 JavaScript and 304 Python repositories.

**Feature extraction:** Studies investigated different techniques for code and feature representations. In the simplest form, Jiang et al. [212] tokenized their code and text. Jiang and McMillan [214] extracted commit messages starting from ``verb + object'' and computed TFIDF for each word. Haque et al. [189] extracted top-40 most-common action words from the dataset of 2.1m Java methods provided by LeClair and McMillan [251]. Psarras et al. [392] used comments as well as source code elements such as method name, variables, and method definition to prepare bag-of-words representation for each class. Liu et al. [285] represented the extracted call dependency features as a sequence of tokens.

Some of the studies extracted explicit features from code or AST. For example, Viuginov and Filchenkov [510] used 17 languages as independent and 8 languages as dependent features. These features include AST features such as depth of code blocks’ root node, number of AST nodes, and number of lines in the block. Hu et al. [198] and Li et al. [260] transformed AST into Structure-Based Traversal (SST). Yang et al. [546] developed a DL approach, MMTrans, for code summarization that learns the representation of source code from the two heterogeneous modalities of the AST, i.e., AST sequences and graphs. Zhou et al. [580] extracted AST and prepared tokenized code sequences and tokenized AST to feed to semantic and structural encoders respectively. Zhou et al. [581, 579] tokenized source code and parse them into AST. Lin et al. [277] proposed block-wise AST splitting method; they split the code of a method based on the blocks in the dominator tree of the Control Flow Graph, and generated a split AST for each block. Liu et al. [292] worked with AST diff between commits as input to generate a commit summary. Lu et al. [301] used Eclipse JDT to parse code snippets at method-level into AST and extracted API sequences and corresponding comments to generate comments for API-based snippets. Huang et al. [201] proposed a statement-based AST...
traversal algorithm to generate the code token sequence preserving the semantic, syntactic and structural information in the code snippet.

The most common way of representing features in this category is to encode the features in the form of embeddings or feature vectors. Specifically, LeClair et al. [252] used embeddings layer for code, text, as well as for AST. Similarly, Choi et al. [104] transformed each of the tokenized source code into a vector of fixed length through an embedding layer. Wang et al. [516] extracted the functional keyword from the code and perform positional encoding. Yao et al. [548] used a code retrieval pre-trained model with natural language query and code snippet and annotated each code snippet with the help of a trained model. Ye et al. [549] utilized two separate embedding layers to convert input sequences, belonging to both text and code, into high-dimensional vectors. Furthermore, some authors encode source code models using various techniques. For instance, Chen et al. [95] represented every input code snippet as a series of ASTs where each path is seen as a sequence of embedding vectors associated with all the path nodes. LeClair et al. [253] used a single embedding layer for both the source code and AST node inputs to exploit a large overlap in vocabulary. Wang et al. [523] prepared a large-scale corpus of training data where each code sample is represented by three sequences—code (in text form), AST, and CFG. These sequences are encoded into vector forms using work2vec. Studies also explored other mechanisms to encode features. For example, Liu et al. [295] extracted commit diffs and represented them as bag of words. The corresponding model ignores grammar and word order, but keeps term frequencies. The vector obtained from the model is referred to as diff vector. Zhang et al. [565] parsed code snippets into ASTs and calculated their similarity using ASTs. Allamanis et al. [26] and Ahmad et al. [9] employed attention-based mechanism to encode tokens. Li et al. [268] used GloVe, a word embedding technique, to obtain the vector representation of the context; the study included method callers and callee as well as other methods in the enclosing class as the context for a method. Similarly, Li et al. [262] calculated edit vectors based on the lexical and semantic differences between input code and the similar code.

**ML model training:** The ML techniques used by the studies in this category can be divided into the following four categories.

**Encoder-decoder models:** The majority of the studies used attention-based Encoder-Decoder models to generate code summaries for code snippets. We further classify the studies in three categories based on their ML implementation.

A large portion of the studies use sequence-to-sequence based approaches. For instance, Gao et al. [151] proposed an end-to-end sequence-to-sequence system enhanced with an attention mechanism to perform better content selection. A code snippet is transformed by a source-code encoder into a vector representation; the decoder reads the code embeddings to generate the target question titles. Jiang et al. [212] trained an LSTM algorithm to `translate` from diffs to commit messages. Iyer et al. [204] used an attention-based neural network to model the conditional distribution of a natural language summary. Their approach uses an LSTM model guided by attention on the source code snippet to generate a summary of one word at a time. Choi et al. [104] transformed input source code into a context vector by detecting local structural features with CNNs. Also, attention mechanism is used with encoder CNNs to identify interesting locations within the source code. Similarly, Jiang [213], Haque et al. [188], Liu et al. [296], Lu et al. [301], Takahashi et al. [478] employed LSTM-based Encoder-Decoder model to generate summaries. Their last module decoder generates source code summary. Ahmad et al. [9] proposed to use Transformer to generate a natural language summary given a piece of source code. For both encoder and decoder, the Transformer consists of stacked multi-head attention and parameterized linear transformation layers. LeClair et al. [252] used attention mechanism to not only attend words in the output summary to words in the code word representation but also to attend the summary words to parts of the AST. The concatenated context vector is used to predict the summary of one word at a time. Xie et al. [536] designed a novel multi-task learning (MLT) approach for code summarization through mining the
relationship between method-code summaries and method names. Li et al. [268] used RNN-based encoder-decoder model to generate a code representation of a method and check whether the current method name is inconsistent with the predicted name based on the semantic representation. Haque et al. [189] compared five seq2seq-like approaches (attendgru, ast-attendgru, ast-attendgru-fc, graph2seq, and code2seq) to explore the role of action word identification in code summarization. Wang et al. [515] proposed a new approach, named CoRec, to translate git diffs, using attentional Encoder-Decoder model, that include both code changes and non-code changes into commit messages. Zhou et al. [578] presented ContextCC that uses a Seq2Seq Neural Network model with an attention mechanism to generate comments for Java methods.

Other studies relied on tree-based approaches. For example, Yang et al. [546] developed a multi-modal transformer-based code summarization approach for smart contracts. Bansal et al. [51] introduced a project-level encoder DL model for code summarization. Chen et al. [95], Hu et al. [198] employed LSTM-based Encoder-Decoder model to generate summaries.

Rest of the studies employed retrieval-based techniques. Zhang et al. [565] proposed Rencos in which they first trained an attentional Encoder-Decoder model to obtain an encoder for all code samples and a decoder for generating natural language summaries. Second, the approach retrieves the most similar code snippets from the training set for each input code snippet. Rencos uses the trained model to encode the input and retrieves two code snippets as context vectors. It then decodes them simultaneously to adjust the conditional probability of the next word using the similarity values from the retrieved two code snippets. Li et al. [262] implemented their retrieve-and-edit approach by using LSTM-based models.

Extended encoder-decoder models: Many studies extended the traditional Encoder-Decoder mechanism in a variety of ways. Among them, sequence-to-sequence based approaches include an approach proposed by Liu et al. [285]; they introduced CallNN that utilizes call dependency information. They employed two encoders, one for the source code and another for the call dependency sequence. The generated output from the two encoders are integrated and used in a decoder for the target natural language summarization. Wang et al. [516] implemented a three step approach. In the first step, functional reinforcer extracts the most critical function-indicated tokens from source code which are fed into the second module code encoder along with source code. The output of the code encoder is given to a decoder that generates the target sequence by sequentially predicting the probability of words one by one. LeClair et al. [253] proposed to use GNN-based encoder to encode AST of each method and RNN-based encoder to model the method as a sequence. They used an attention mechanism to learn important tokens in the code and corresponding AST. Finally, the decoder generates a sequence of tokens based on the encoder output. Zhou et al. [580] used two encoders, semantic and structural, to generate summaries for Java methods. Their method combined text features with structure information of code snippets to train encoders with multiple graph attention layers.

Li et al. [260] presented a tree-based approach Hybrid-DeepCon model containing two encoders for code and AST along with a decoder to generate sequences of natural language annotations. Shido et al. [443] extended Tree-LSTM and proposed Multi-way Tree-LSTM as their encoder. The rational behind the extension is that the proposed approach not only can handle an arbitrary number of ordered children, but also factor-in interactions among children. Zhou et al. [581] trained two separate Encoder-Decoder models, one for source code sequence and another for AST via adversarial training, where each model is guided by a well-designed discriminator that learns to evaluate its outputs. Lin et al. [277] used a transformer to generate high-quality code summaries. The learned syntax encoding is combined with code encoding, and fed into the transformer.

Rest of the approaches adopted retrieval-based approaches. Ye et al. [549] employed dual learning mechanism by using Bi-LSTM. In one direction, the model is trained for code summarization task that takes code sequence as input and summarized into a sequence of text. On the other hand, the code generation task takes the text sequence and generate code sequence. They reused the
outcome of both tasks to improve performance of the other task. Liu et al. [292] proposed a new approach ATOM that uses the diff between commits as input. The approach used BiLSTM module to generate a new message by using \textit{diff-diff} to retrieve the most relevant commit message.

**Reinforcement learning models:** Some of the studies exploited reinforcement learning techniques for code summary generation. In particular, Yao et al. [548] proposed code annotation for code retrieval method that generates an natural language annotation for a code snippet so that the generated annotation can be used for code retrieval. They used \textit{Advanced Actor-Critic} model for annotation mechanism and LSTM based model for code retrieval. Wan et al. [511] and Wang et al. [523] used deep reinforcement learning model for training using annotated code samples. The trained model is an \textit{Actor} network that generates comments for input code snippets. The \textit{Critic} module evaluates whether the generated word is a good fit or not. Wang et al. [526] used a hierarchical attention network for comment generation. The study incorporated multiple code features, including type-augmented abstract syntax trees and program control flows, along with plain code sequences. The extracted features are injected into an actor-critic network. Huang et al. [201] proposed a composite learning model, which combines the actor-critic algorithm of reinforcement learning with the encoder-decoder algorithm, to generate block comments.

**Other techniques:** Jiang and McMillan [214] used \textit{Naive Bayes} to classify the diff files into the verb groups. For automated code folding, Viuginov and Filchenkov [510] used \textit{Random Forest} and \textit{Decision Tree} to classify whether a code block needs to be folded. Similarly, Nazar et al. [347] used \textit{Support Vector Machine} and \textit{Naive Bayes} classifiers to generate summaries from the extracted features. Chen et al. [96] compared six \textit{ml} techniques to demonstrate that comment category prediction can boost code summarization to reach better results. Etemadi and Monperrus [138] compared NNGen, SimpleNNGen, and EXC-NNGen to explore the origin of nearest diffs selected by the neural network.

### 3.6.2 Program classification

Studies targeting this category classify software artifacts based on programming language [504], application domain [504], and type of commits (such as buggy and adaptive) [207, 334]. We summarize these efforts below from dataset preparation, feature extraction, and \textit{ml} model training perspective.

**Dataset preparation:** Ma et al. [308] identified more than 91 thousand open-source repositories from GitHub as subject systems. They created an oracle by manually classifying software artifacts from 383 sample projects. Shimonaka et al. [445] conducted experiments on source code generated by four kinds of code generators to evaluate their technique that identify auto-generated code automatically by using \textit{ml} techniques. Ji et al. [207] and Meqdadi et al. [334] analyzed the GitHub commit history. Ugurel et al. [504] relied on C and C++ projects from Ibiblio and the Sourceforge archives. Levin and Yehudai [258] used eleven popular open-source projects and annotated 1151 commits manually to train a model that can classify commits into maintenance activities. Similarly, Mariano et al. [325] and Mariano et al. [324] classify commits by maintenance activities; they identify a large number of open-source GitHub repositories. Along the similar lines, Meng et al. [333] classified commits messages into categories such as bug fix and feature addition and Li et al. [261] predicted the impact of single commit on the program. They used popular a small set (specifically, 5 and 10 respectively) of Java projects as their dataset. Furthermore, Sabetta and Bezzi [411] proposed an approach to classify security-related commits. To achieve the goal, they used 660 such commits from 152 open-source Java projects that are used in SAP software. Gharbi et al. [154] created a dataset containing 29K commits from 12 open source projects. Abdalkareem et al. [3] built a dataset to improve the detection CI skip commits i.e., commits where `\texttt{[ci skip]}` or `\texttt{[skip ci]}` is used to skip continuous integration pipeline to execute on the pushed commit. To build the dataset, the authors used BigQuery GitHub dataset to identify repositories where at least 10% of commits skipped the CI pipeline. Altarawy et al. [35] used three labeled data sets including one
that was created with 103 applications implemented in 19 different languages to find similar applications.

**Feature extraction:** Features in this category of studies belong to either source code features category or repository features. A subset of studies [445, 308, 504] relies on features extracted from source code token including language specific keywords and other syntactic information. Other studies [207, 334] collect repository metrics (such as number of changed statements, methods, hunks, and files) to classify commits. Ben-Nun et al. [57] leveraged both the underlying data- and control-flow of a program to learn code semantics performance prediction. Gharbi et al. [154] used TF-IDF to weight the tokens extracted from change messages. Ghadhab et al. [152] curated a set of 768 BERT-generated features, a set of 70 code change-based features and a set of 20 keyword-based features for training a model to classify commits. Similarly, Mariano et al. [325] and Mariano et al. [324] extracted a 71 features majorly belonging to source code changes and keyword occurrences categories. Meng et al. [333] and Li et al. [261] computed change metrics (such as number of lines added and removed) as well as natural language metrics extracted from commit messages. Abdalkareem et al. [3] employed 23 commit-level repository metrics. Sabetta and Bezi [411] analyzed changes in source code associated with each commit and extracted the terms that the developer used to name entities in the source code (e.g., names of classes). Similarly, LASCAD Altarawy et al. [35] extracted terms from the source code and preprocessed terms by removing English stop words and programming language keywords.

**ML model training:** A variety of ML approaches have been applied. Specifically, Ma et al. [308] used Support Vector Machine, Decision Tree, and Bayes Network for artifact classification. Meqdadi et al. [334] employed Naive Bayes, Ripper, as well as Decision Tree and Ugurel et al. [504] used Support Vector Machine to classify specific commits. Ben-Nun et al. [57] proposed an approach based on an RNN architecture and fixed inst2vec embeddings for code analysis tasks. Levin and Yehudai [258], Mariano et al. [325, 324] used Decision Tree and Random Forest for commits classification into maintenance activities. Gharbi et al. [154] applied Logistic Regression model to determine the commit classes for each new commit message. Ghabhab et al. [152] trained a DNN classifier to fine-tune the BERT model on the task of commit classification. Meng et al. [333] used a CNN-based model to classify code commits. Sabetta and Bezi [411] trained Random Forest, Naive Bayes, and Support Vector Machine to identify security-relevant commits. Altarawy et al. [35] developed LASCAD using Latent Dirichlet Allocation and hierarchical clustering to establish similarities among software projects.

3.6.3 Change analysis

Researchers have explored applications of ML techniques to identify or predict relevant code changes [484, 489]. We briefly describe the efforts in this domain w.r.t. three major steps—dataset preparation, feature extraction, and ML model training.

**Dataset preparation:** Tollin et al. [484] performed their study on two industrial projects. Tufano et al. [489] extracted 236K pairs of code snippets identified before and after the implementation of the changes provided in the pull requests. Kumar et al. [241] used eBay web-services as their subject systems. Uchôa et al. [503] used the data provided by the Code Review Open Platform (CROP), an open-source dataset that links code review data to software changes, to predict impactful changes in code review. Malhotra and Khanna [319] considered three open-source projects to investigate the relationship between code quality metrics and change proneness.

**Feature extraction:** Tollin et al. [484] extracted features related to the code quality from the issues of two industrial projects. Tufano et al. [489] used features from pull requests to investigate the ability of a NMT modes. Abbas et al. [2] and Malhotra and Khanna [319] computed well-known C&K metrics to investigate the relationship between change proneness and object-oriented metrics. Similarly, Kumar et al. [241] computed 21 code quality metrics to predict change-prone web-
services. Uchôa et al. [503] combines metrics from different sources—21 features related to source code, modification history of the files, and the textual description of the change, 20 features that characterize the developer’s experience, and 27 code smells detected by DesigniteJava [432].

**ML model training:** Tollin et al. [484] employed Decision Tree, Random Forest, and Naïve Bayes algorithms for their prediction task. Tufano et al. [489] used Encoder-Decoder architecture of a typical NMT model to learn the changes introduced in pull requests. Malhotra and Khanna [319] experimented with Multilayer Perceptron, and Random Forest to observe relationship between code metrics and change proneness. Abbas et al. [2] compared ten ML models including Random Forest, Decision Tree, Multilayer Perceptron, and Bayes Network. Similarly, Kumar et al. [241] used Support Vector Machine to predict change proneness in web-services. Uchôa et al. [503] used six ML models such as Support Vector Machine, Decision Tree, and Random Forest to investigate whether predicted impactful changes are helpful for code reviewers.

### 3.6.4 Entity identification/recommendation

This category represents studies that recommend source code entities (such as method and class names) [24, 322, 539, 210, 192] or identify entities such as design patterns [150] in code using ML [502, 17, 559, 133, 87]. Specifically, Linstead et al. [284] proposed a method to identify functional components in source code and to understand code evolution to analyze emergence of functional topics with time. Huang et al. [200] found commenting position in code using ML techniques. Uchiyama et al. [502] identified design patterns and Abuhamad et al. [5] recommended code authorship. Similar approaches include recommending method name [24, 210, 539], method signature [322], class name [24], and type inference [192]. We summarize these efforts classified in three steps of applying ML techniques below.

**Dataset preparation:** The majority of the studies employed GitHub projects for their experiments. Specifically, Linstead et al. [284] used two large, open source Java projects, Eclipse and ArgoUML in their experiments to apply unsupervised statistical topic models. Similarly, Hellendoorn et al. [192] downloaded 1,000 open-source TypeScript projects and extracted identifiers with corresponding type information. Abuhamad et al. [5] evaluated their approach over the entire Google Code Jam (gcj) dataset (from 2008 to 2016) and over real-world code samples (from 1987) extracted from public repositories on GitHub. Allamanis et al. [24] mined 20 software projects from GitHub to predict method and class names. Jiang et al. [210] used the Code2Seq dataset containing 3.8 million methods as their experimental data. Ali et al. [18] applied information retrieval techniques to automatically create traceability links in three subject systems.

A subset of studies focused on identifying design patterns using ML techniques. Uchiyama et al. [502] performed experimental evaluations with five programs to evaluate their approach on predicting design patterns. Alhusain et al. [17] applied a set of design patterns detection tools on 400 open source repositories; they selected all identified instances where at least two tools report a design pattern instance. Zanoni et al. [559] manually identified 2,794 design patterns instances from ten open-source repositories. Dwivedi et al. [133] analyzed JHotDraw and identified 59 instances of abstract factory and 160 instances of adapter pattern for their experiment. Similarly, Gopalakrishnan et al. [159] applied their approach to discover latent topics in source code on 116,000 open-source projects. They recommended architectural tactics based on the discovered topics. Furthermore, Mahmoud and Bradshaw [312] chose ten open-source projects to validate their topic modeling approach designed for source code.

**Feature extraction:** Several studies generated embeddings from their feature set. Specifically, Huang et al. [200] used embeddings generated from Word2vec capturing code semantics. Similarly, Jiang et al. [210] employed Code2vec embeddings and Allamanis et al. [24] used embeddings that contain semantic information about sub-tokens of a method name to identify similar embeddings utilized in similar contexts. Zhang et al. [567] utilized knowledge graph embeddings to extract interrelations of code for bug localization.
Other studies used source code or code metadata as features. Abuhamad et al. [5] extracted code authorship attributes from samples of code. Malik et al. [322] used function names, formal parameters, and corresponding comments as features. Ali et al. [18] extracted source code entity names, such as class, method, and variable names. Bavota et al. [56] retrieved 618 features from six open-source Java systems to apply Latent Dirichlet Allocation-based feature location technique. Similarly, De Lucia et al. [119] extracted class name, signature of methods, and attribute names from Java source code. They applied Latent Dirichlet Allocation to label source code artifacts. Gopalakrishnan et al. [159] processed tactics in the form of a set of textual descriptions and produced a set of weighted indicator terms. Mahmoud and Bradshaw [312] extracted code term co-occurrence, pair-wise term similarity, and clusters of terms features and applied their approach Semantic Topic Models (STM) on them.

In addition, Uchiyama et al. [502], Chaturvedi et al. [87], Dwivedi et al. [133], Alhusain et al. [17] used several source-code metrics as features to detect design patterns in software programs.

**ML model training**: The majority of studies in this category use RNN-based ML models. In particular, Huang et al. [200] and Hellendoorn et al. [192] used bidirectional RNN models. Similarly, Abuhamad et al. [5] and Malik et al. [322] also employed RNN models to identify code authorship and function signatures respectively. Zhang et al. [567] created a bug-localization tool, KGBugLocator utilizing knowledge graph embeddings and bi-directional attention models. Xu et al. [539] employed the GRU-based Encoder-Decoder model for method name prediction. Uchiyama et al. [502] used a hierarchical neural network as their classifier. Allamanis et al. [24] utilized neural language models for predicting method and class names.

Other studies used traditional ML techniques. Specifically, Chaturvedi et al. [87] compared four ML techniques (Linear Regression, Polynomial Regression, support vector regression, and neural network). Dwivedi et al. [133] used Decision Tree and Zanoni et al. [559] trained Naive Bayes, Decision Tree, Random Forest, and Support Vector Machine to detect design patterns using ML. Ali et al. [18] employed Latent Dirichlet Allocation to distinguish domain-level terms from implementation-level terms. Gopalakrishnan et al. [159] discovered latent topics using Latent Dirichlet Allocation in the large-scale corpus. The study used Decision Tree, Random Forest, and Linear Regression as classifiers to compute the likelihood that a given source file is associated with a given tactic.

### 3.7 Code review

Code Review is the process of systematically check the code written by a developer performed by one or more different developers. A very small set of studies explore the role of ML in the process of code review that we present in this section.

**Dataset preparation**: Lal and Pahwa [245] labeled check-in code samples as clean and buggy. On code samples, they carried out extensive pre-processing such as normalization and label encoding. Aiming to automate code review process, Tufano et al. [493] trained two DL architectures one for both contributor and for reviewer. They mined Gerrit and GitHub to prepare their dataset from 8,904 projects. Furthermore, Thongtanunam et al. [482] proposed AutoTransform to better handle new tokens using Byte-Pair Encoding (BPE) approach. They leveraged the dataset proposed by Tufano et al. [493] consisting 630,858 changed methodstotrainaTransformer-basedNMTmodel.

**Feature extraction**: Lal and Pahwa [245] used TF-IDF to convert the code samples into vectors after applying extensive pre-processing. Tufano et al. [493] used n-grams extracted from each commit to train their classifiers.

**ML model training**: Lal and Pahwa [245] used a Naive Bayes model to classify samples into buggy or clean. Tufano et al. [493] trained two DL architectures one for both contributor and for reviewer. The authors use n-grams extracted from each commit and implement their classifiers using Decision Tree, Naive Bayes, and Random Forest. In their revised work [494], the authors used Text-To-Text Transfer Transformer (T5) model and shown significant improvements in DL code review models.
3.8 Code search

Code search is an activity of searching a code snippet based on individual's need typically in Q&A sites such as StackOverflow [413, 450, 512]. The studies in this category define the following coarse-grained steps. In the first step, the techniques prepare a training set by collecting source code and often corresponding description or query. A feature extraction step then identifies and extracts relevant features from the input code and text. Next, these features are fed into ML models for training which is later used to execute test queries.

Dataset preparation: Shuai et al. [450] utilized commented code as input. Wan et al. [512] used source code in the the form of tokens, AST, and CFG. Sachdev et al. [413] employed a simple tokenizer to extract all tokens from source code by removing non–alphanumeric tokens. Ling et al. [282] mined software projects from GitHub for the training of their approach. Jiang et al. [208] used existing McGill corpus and Android corpus.

Feature extraction: Code search studies typically use embeddings representing the input code. Shuai et al. [450] performed embeddings on code, where source code elements (method name, API sequence, and tokens) are processed separately. They generated embeddings for code comments independently. Wan et al. [512] employed a multi-modal code representation, where they learnt the representation of each modality via LSTM, TREE-LSTM and GNN, respectively. Sachdev et al. [413] identified words from source code and transformed the extracted tokens into a natural language documents. Similarly, Ling et al. [282] used an unsupervised word embedding technique to construct a matching matrix to represent lexical similarities in software projects and used an RNN model to capture latent syntactic patterns for adaptive code search. Jiang et al. [208] used a fragment parser to parse a tutorial fragment in four steps (API discovery, pronoun and variable resolution, sentence identification, and sentence type identification).

ML model training: Shuai et al. [450] used a CNN-based ML model named CARLCS-CNN. The corresponding model learns interdependent representations for embedded code and query by a co-attention mechanism. Based on the embedded code and query, the co-attention mechanism learns a correlation matrix and leverages row/column-wise max-pooling on the matrix. Wan et al. [512] employed a multi-modal attention fusion. The model learns representations of different modality and assigns weights using an attention layer. Next, the attention vectors are fused into a single vector. Sachdev et al. [413] utilized word and documentation embeddings and performed code search using the learned embeddings. Similarly, Ling et al. [282] used an autoencoder network and a metric (believability) to measure the degree to which a sentence is approved or disapproved within a discussion in a issue-tracking system. Jiang et al. [208] used Latent Dirichlet Allocation to segregate all tutorial fragments into relevant clusters and identify relevant tutorial for an API.

Once an ML model is trained, code search can be initiated using a query and a code snippet. Shuai et al. [450] used the given query and code sample to measure the semantic similarity using cosine similarity. Wan et al. [512] ranked all the code snippets by their similarities with the input query. Similarly, Sachdev et al. [413] were able to answer almost 43% of the collected StackOverflow questions directly from code.

3.9 Refactoring

Refactoring transformations are intended to improve code quality (specifically maintainability), while preserving the program behavior (functional requirements) from users' perspective [471]. This section summarizes the studies that identify refactoring candidates or predict refactoring commits by analyzing source code and by applying ML techniques on code. A process pipeline typically adopted by the studies in this category can be viewed as a three step process. In the first step, the source code of the projects is used to prepare a dataset for training. Then, individual samples (i.e., either a method, class, or a file) is processed to extract relevant features. The extracted features are then fed to an ML model for training. Once trained, the model is used to predict whether an
Dataset preparation: The first set of studies created their own dataset for model training. For instance, Rodriguez et al. [407] and Amal et al. [37] created datasets where each sample is reviewed by a human to identify an applicable refactoring operation; the identified operation is carried out by automated means. Kosker et al. [234] employed four versions of the same repository, computed their complexity metrics, and classified their classes as refactored if their complexity metric values are reduced from the previous version. Nyamawe et al. [354] analyzed 43 open-source repositories with 13.5 thousand commits to prepare their dataset. Similarly, Aniche et al. [40] created a dataset comprising over two million refactorings from more than 11 thousand open-source repositories. Sagar et al. [414] identified 5004 commits randomly selected from all the commits obtained from 800 open-source repositories where RefactoringMiner [486] identified at least one refactoring. Along the similar lines, Li et al. [268] used RefactoringMiner and RefDiff tools to identify refactoring operations in the selected commits. Xu et al. [538], Krasniqi and Cleland-Huang [236] used manual analysis and tagging for identifying refactoring operations. Bavota et al. [55] obtained 2,329 classes from nine subject systems and applied topic modeling to identify latent topics and move them to an appropriate package. Similarly, Bavota et al. [56] identified all classes from six software systems and applied their proposed technique namely Methodbook to identify move method refactoring candidates using relational topic models. Finally, Kurbatova et al. [244] generated synthetic data by moving methods to other classes to prepare a dataset for feature envy smell. The rest of the studies in this category [239, 242, 43], used the tera-Promise dataset containing various metrics for open-source projects where the classes that need refactoring are tagged.

Feature extraction: A variety of features, belonging to product as well as process metrics, has been employed by the studies in this category. Some of the studies rely on code quality metrics. Specifically, Kosker et al. [234] computed cyclomatic complexity along with 25 other code quality metrics. Similarly, Kumar et al. [242] computed 25 different code quality metrics using the SourceMeter tool; these metrics include cyclomatic complexity, class class and clone complexity, ILOC, outgoing method invocations, and so on. Some of the studies [239, 43, 451, 524] calculated a large number of metrics. Specifically, Kumar and Sureka [239] computed 102 metrics and then applied PCA to reduce the number of features to 31, while Aribandi et al. [43] used 125 metrics. Sidhu et al. [451] used metrics capturing design characteristics of a model including inheritance, coupling and modularity, and size. Wang and Godfrey [524] computed a wide range of metrics related to clones such as number of clone fragments in a class, clone type (type1, type2, or type3), and lines of code in the cloned method.

Some other studies did not limit themselves to only code quality metrics. Particularly, Yue et al. [558] collected 34 features belonging to code, evolution history, diff between commits, and co-change. Similarly, Aniche et al. [40] extracted code quality metrics, process metrics, and code ownership metrics.

In addition, Nyamawe et al. [354], Nyamawe et al. [355] carried out standard NLP preprocessing and generated TF-IDF embeddings for each sample. Along the similar lines, Kurbatova et al. [244] used code2vec to generate embeddings for each method. Sagar et al. [414] extracted keywords from commit messages and used GloVe to obtain the corresponding embedding. Krasniqi and Cleland-Huang [236] tagged each commit message with their parts-of-speech and prepared a language model dependency tree to detect refactoring operations from commit messages. Bavota et al. [55] and Bavota et al. [56] extracted identifiers, comments, and string literals from source code. Bavota et al. [55] prepared structural coupling matrix and package decomposition matrix to identify move class candidates. Bavota et al. [56] applied relational topic models to derive semantic relationships between methods and define a probability distribution of topics (topic distribution model) among methods to refactor feature envy code smell.
Many studies [309, 112, 349, 135, 331, 146, 383, 238, 369, 36, 172, 107, 102, 338, 196, 422, 543, 573, 379, 430, 216, 280, 278] created their own datasets. Ma et al. [309], Ali Alatwi et al. [19], Cui et al. [112], and Gupta et al. [172] created datasets to train vulnerability detectors for Android applications. In particular, Ma et al. [309] decompiled and generated 9,462 malware classes and used them to train a decision tree classifier. Ali Alatwi et al. [19] collected 5,063 Android applications where 1,000 of them were marked as being vulnerable. They trained a random forest classifier on the collected data.
nign and the remaining as malware; Cui et al. [112] selected an open-source dataset comprised of 1,179 Android applications that have 4,416 different version (of the 1,179 applications) and labeled the selected dataset by using the Androrisk tool; and Gupta et al. [172] used two Android applications (Android-universal-image-loader and JHotDraw) which they have manually labeled based on the projects PMD reports (true if a vulnerability was reported in a PMD file and false otherwise). To create datasets of PHP projects, Medeiros et al. [331] collected 35 open-source PHP projects and intentionally injected 76 vulnerabilities in their dataset. Shar et al. [430] used phpminer to extract 15 datasets that include SQL injections, cross-site scripting, remote code execution, and file inclusion vulnerabilities, and labeled only 20% of their dataset to point out the precision of their approach. Ndichu et al. [349] collected 5,024 JavaScript code snippets from D3M, jsunpack, and 100 top websites where the half of the code snippets were benign and the other half malicious. In other cases, authors [543, 397, 379] collected large number of commit messages and mapped them to known vulnerabilities by using Google's Play Store, National Vulnerability Database (NVD), Synx, Node Security Project, and so on, while in limited cases authors [383] manually label their dataset. Hou et al. [196], Moskovitch et al. [338] and Santos et al. [422] created their datasets by collecting web-page samples from StopBadWare and VxHeavens. Lin et al. [280] constructed a dataset and manually labeled 1,471 vulnerable functions and 1,320 vulnerable files from nine open-source applications, named Asterisk, FFmpeg, HTTPD, LibPNG, LibTIFF, OpenSSL, Pidgin, VLC Player, and Xen. Lin et al. [278] have used more than 30,000 non-vulnerable functions and manually labeled 475 vulnerable functions for their experiments.

**Feature extraction:** Authors used static source code metrics, CFGS, ASTS, source code tokens, and word embeddings as features.

**Source code metrics:** A set of studies [331, 146, 36, 172, 107, 397, 112, 383, 403, 130, 232, 332, 6, 247, 467] used more than 20 static source code metrics (such as cyclomatic complexity, maximum depth of class in inheritance tree, number of statements, and number of blank lines).

**Data/control flow and AST:** Ma et al. [307], Kim et al. [231], Bilgin et al. [61], Kronjee et al. [238], Wang et al. [527], Du et al. [131], Medeiros et al. [332] used CFGS, ASTS, or data flow analysis as features. More specifically, Ma et al. [309] extracted the API calls from the CFGS of their dataset and collected information such as the usage of APIs (which APIs the application uses), the API frequencies (how many times the application uses APIs) and API sequence (the order the application uses APIs). Kim et al. [231] extracted ASTS and CFGS which they tokenized and fed into ML models, while Bilgin et al. [61] extracted ASTS and translated their representation of source code into a one-dimensional numerical array to feed them to a model. Kronjee et al. [238] used data-flow analysis to extract features, while Spreitzenbarth et al. [461] used static, dynamic analysis, and information collected from ltrace to collect features and train a linear vulnerability detection model. Lin et al. [278] created ASTS and from there they extracted code semantics as features.

**Repository and file metrics:** Perl et al. [379] collected GitHub repository meta-data (i.e., programming language, star count, fork count, and number of commits) in addition to source code metrics. Other authors [378, 135] used file-meta-data such as files’ creation and modification time, machine type, file size, and linker version.

**Code and Text tokens:** Chernis and Verma [102] used simple token features (character count, character diversity, entropy, maximum nesting depth, arrow count, ‘if’ count, ‘if’ complexity, ‘while’ count, and ‘for’ count) and complex features (character n-grams, word n-grams, and suffix trees). Hou et al. [196] collected 10 features such as length of the document, average length of word, word count, word count in a line, and number of NULL characters. The remaining studies [409, 369, 338, 422, 543, 412, 573, 430, 100, 346, 409, 327, 143, 570, 370, 48, 555, 280] tokenized parts of the source code or text-based information with various techniques such as the most frequent occurrences of operational codes, capture the meaning of critical tokens, or applied techniques to reduce the vocabulary size in order to retrieve the most important tokens. In some other cases, authors [269]
used statistical techniques to reduce the feature space to reduce the number of code tokens.

**Other features:** Ali Alatwi et al. [19], Ndichu et al. [349] and Milosevic et al. [337] extracted permission-related features. In other cases, authors [541] combined software metrics and N-grams as features to train models and others [448] created text-based images to extract features. Likewise, Sultana [466] extracted traceable patterns such as CompoundBox, Immutable, Implementor, Override, Sink, Stateless, FunctionObject, and LimitSel and used Understand tool to extract various software metrics. Wei et al. [531] extracted system calls and function call-related information to use as features, while Vishnu and Jevitha [509] extracted URL-based features like number of chars, duplicated characters, special characters, script tags, cookies, and re-directions. Padmanabhuni and Tan [362] extracted buffer usage patterns and defensive mechanisms statements constructs by analyzing files.

**Model training:** To train models, the selected studies used a variety of traditional ML and DL algorithms.

**Traditional ML techniques:** One set of studies [19, 349, 378, 409, 369, 338, 379, 430, 555, 467, 362, 247, 6, 556, 466, 509, 531, 130, 143, 332, 131, 346, 527, 100, 403] used traditional ML algorithms such as Naive Bayes, Decision Tree, Support Vector Machine, Linear Regression, Decision Tree, and Random Forest to train their models. Specifically, Ali Alatwi et al. [19], Russell et al. [409], Perl et al. [379] selected Support Vector Machine because it is not affected by over-fitting when having very high dimensional variable spaces. Along the similar lines, Ndichu et al. [349] used Support Vector Machine to train their model with linear kernel. Pereira et al. [378] used Decision Tree, Linear Regression, and Lasso to train their models, while [6] found that Random Forest is the best model for predicting cross-project vulnerabilities. Compared to the above studies, Shar et al. [430] used both supervised (i.e., Linear Regression and Random Forest) and semi-supervised (i.e., Co-trained Random Forest) algorithms to train their models since most of that datasets were not labeled. Yosifova et al. [555] used text-based features to train Naive Bayes, Support Vector Machine, and Random Forest models. Du et al. [130] created the LEOPARD framework that does not require prior knowledge about known vulnerabilities and used Random Forest, Naive Bayes, Support Vector Machine, and Decision Tree to point them out.

Other studies [331, 146, 383, 238, 36, 172, 107, 337, 102, 196, 422, 397, 112] used up to 32 different ML algorithms to train models and compared their performance. Specifically, Medeiros et al. [331] experimented with multiple variants of Decision Tree, Random Forest, Naive Bayes, K Nearest Neighbors, Linear Regression, Multilayer Perceptron, and Support Vector Machine models and identified Support Vector Machine as the best performing classifier for their experiment. Likewise, Milosevic et al. [337] and Rahman et al. [397] employed multiple ML algorithms, respectively, and found that Support Vector Machine offers the highest accuracy rate for training vulnerability detectors. In contrast to the above studies, Ferenc et al. [146] showed that K Nearest Neighbors offers the best performance for their dataset after experimenting with DNN, K Nearest Neighbors, Support Vector Machine, Linear Regression, Decision Tree, Random Forest, and Naive Bayes. In order to find out which is the best model for the swan tool, Piskachev et al. [383] evaluated the Support Vector Machine, Naive Bayes, Bayes Network, Decision Tree, Stump, and Ripper. Their results pointed out the Support Vector Machine as the best performing model to detect vulnerabilities. Similarly, Kronjee et al. [238], Cui et al. [112], and Gupta et al. [172] compared different ML algorithms and found Decision Tree and Random Forest as the best performing algorithms.

**DL techniques:** A large number of studies [543, 412, 231, 280, 48, 232, 327, 278, 448, 54] used ML methods such as CNN, RNN, and ANN to train models. In more details, Yang et al. [543] utilized the BP-ANN algorithm to train vulnerability detectors. For the project Achilles, Saccente et al. [412] used an array of LSTM models to train on data containing Java code snippets for a specific set of vulnerability types. In another study, Kim et al. [231] suggested a DL framework that makes use of RNN models to train vulnerability detectors. Specifically, the authors framework first feeds the code embed-
dings into a bi-LSTM model to capture the feature semantics, then an attention layer is used to get the vector weights, and finally, passed into a dense layer to output if a code is safe or vulnerable.

Compared to the studies that examined traditional ML or DL algorithms, Zheng et al. [573] examined both of them. They used Random Forest, K Nearest Neighbors, Support Vector Machine, Linear Regression among the traditional ML algorithms along with bi-LSTM, GRU, and CNN. There results indicate bi-LSTM as the best performing model. Lin et al. [280] developed a benchmarking framework that can use bi-LSTM, LSTM, bi-GRU, GRU, DNN and Text-CNN, but can be extended to use more deep learning models. Kim et al. [232] generating graphical semantics that reflect on code semantic features and use them for Graph Convolutional Network to automatically identify and learn semantic and extract features for vulnerability detection, while Shiqi et al. [448] created textual images and fed them to Deep Belief Networks to classify malware.

3.11 Summary

In this section, we briefly summarize the usage of ML in a software engineering task involving source code analysis. Figure 7 presents an overview of the pipeline that is typically used in a software engineering task that uses ML.

**Dataset preparation:** Preparing a dataset is the first major activity in the pipeline. The activity starts with identifying the source of required data, typically source code repositories. The activity involves selecting and downloading the required repositories, collecting supplementary data (such as GitHub issues), create individual samples sometimes by combining information, and annotate samples. Depending upon the specific software engineering task at hand, these steps are customized and extended.

The outcome of this activity is a dataset. Depending upon the context, the dataset may contain information such as annotated code samples, source code model (e.g., AST), and pairs of buggy code and fixed code.

**Feature extraction:** Performance of a ML model depends significantly on the provided kind and quality of features. Various techniques are applied on the prepared dataset to extract the required features that help the ML model perform well for the given task. Features may take variety of form and format; for source code analysis applications, typical features include source code metrics, source code tokens, their properties, and representation, changes in the code (code diff), vector representation of code and text, dependency graph, and vector representation of AST, CFG, or AST diff. Obviously, selection of the specific features depends on the downstream task.

**ML model training:** Selecting a ML model for a given task depends on many factors such as the nature of the problem, the properties of training and input samples, and the expected output.
Below, we provide an analysis of employed ML models based on these factors.

• One of the factors that influence the choice of ML models is the chosen features and their properties. Studies in the quality assessment category majorly relied on token-based features and code quality metrics. Such features allowed studies in this category to use traditional ML models. Some authors applied ML models such as DNN when higher-granularity constructs such as CFG and DFG are used as features.

• Similarly, the majority of the studies in testing category relied on code quality metrics. Therefore, they have fixed size, fixed meaning (for each column) vectors to feed to a ML model. With such inputs, traditional ML approaches, such as Random Forest and Support Vector Machine, work well. Other studies used a variation of AST of AST of the changes to generate the embeddings. DL models including DNN and RNN-based models are used to first train a model for embeddings. A typical ML classifier use the embeddings to classify samples in buggy or benign.

• Typical output of a code representation study is embeddings representing code in the vector form. The semantics of the produced embeddings significantly depend on the selected features. Studies in this domain identify this aspect and, hence, they are swiftly focused to extract features that capture the relevant semantics; for example, path-based features encode the order among the tokens. The chosen ML model plays another important role to generate effective embeddings. Given the success of RNN with text processing tasks, due to its capability to identify a sequence or pattern, RNN-based models dominate this category.

• Program repair is typically a sequence to sequence transformation i.e., a sequence of buggy code is the input and a sequence of fixed code is the output. Given the nature of the problem, it is not surprising to observe that the majority of the studies in this category used Encoder-Decoder-based models. RNN are considered a popular choice to realize Encoder-Decoder models due to its capability to remember long sequences.

4. Datasets and Tools

For RO3, this section provides a consolidated summary of available datasets and tools that are used by the studies considered in the survey. We carefully examined each selected study and noted the resources (i.e., datasets and tools). We define the following criteria to include a resource in our catalog.

• The referenced resource must have been used by at least one primary study.

• The referenced resource must be publicly available at the time of writing this article (Dec 2022).

• The resource provides bare-minimum usage instructions to build and execute (wherever applicable) and to use the artifact.

• The resource is useful either by providing an implementation of a ML technique, helping the user to generate information/data which is further used by a ML technique, or by providing a processed dataset that can be directly employed in a ML study.

Table 6 lists all the tools that we found in this exploration. Each resource is listed with it's category, name and link to access the resource, number of citations (as of Dec 2022), and the time when it was first introduced along with the time when the resource was last updated. We collected the metadata about the resources manually by searching the digital libraries, repositories, and authors' websites. The cases where we could not find the required information, are marked as ‘–’. We also provide a short description of the resource.

Table 6. A list of tools useful for applying machine learning to source code
<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>#Citation</th>
<th>Introd.</th>
<th>Updated</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Representation</td>
<td>ncc [57]</td>
<td>234</td>
<td>Dec 2018</td>
<td>Aug 2021</td>
<td>Learns representations of code semantics</td>
</tr>
<tr>
<td></td>
<td>Code2seq [31]</td>
<td>536</td>
<td>May 2019</td>
<td>Jul 2022</td>
<td>Generates sequences from structured representation of code</td>
</tr>
<tr>
<td></td>
<td>Vector representation for coding style [235]</td>
<td>3</td>
<td>Sep 2020</td>
<td>Jul 2022</td>
<td>Implements vector representation of individual coding style</td>
</tr>
<tr>
<td></td>
<td>CC2Vec [194]</td>
<td>69</td>
<td>Oct 2020</td>
<td>–</td>
<td>Implements distributed representation of code changes</td>
</tr>
<tr>
<td></td>
<td>Autoencode [490]</td>
<td>75</td>
<td>–</td>
<td>–</td>
<td>Encodes source code fragments into vector representations</td>
</tr>
<tr>
<td></td>
<td>Graph-based code modeling [28]</td>
<td>544</td>
<td>May 2018</td>
<td>May 2021</td>
<td>Generates code modeling with graphs</td>
</tr>
<tr>
<td></td>
<td>Vocabulary learning on code [115]</td>
<td>34</td>
<td>Jan 2019</td>
<td>–</td>
<td>Generates an augmented AST from Java source code</td>
</tr>
<tr>
<td></td>
<td>FRAPT [208]</td>
<td>43</td>
<td>Jul 2017</td>
<td>–</td>
<td>Searches relevant tutorial fragments for APIs</td>
</tr>
<tr>
<td></td>
<td>DeepTyper [192]</td>
<td>87</td>
<td>Oct 2018</td>
<td>Feb 2020</td>
<td>Annotates types for JavaScript and TypeScript</td>
</tr>
<tr>
<td></td>
<td>CallNN [285]</td>
<td>9</td>
<td>Oct 2019</td>
<td>–</td>
<td>Implements a code summarization approach by using call dependencies</td>
</tr>
<tr>
<td></td>
<td>Summarization_tf [443]</td>
<td>30</td>
<td>Jul 2019</td>
<td>–</td>
<td>Summarizes code with Extended TREE-LSTM</td>
</tr>
<tr>
<td></td>
<td>CoaCor [548]</td>
<td>36</td>
<td>Jul 2019</td>
<td>May 2020</td>
<td>Explores the role of rich annotation for code retrieval</td>
</tr>
<tr>
<td>Tool</td>
<td>Year1</td>
<td>Year2</td>
<td>Year3</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>DeepCom</td>
<td>Nov 2020</td>
<td>May 2021</td>
<td>Generates code comments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rencos</td>
<td>Oct 2020</td>
<td>–</td>
<td>Generates code summary by using both neural and retrieval-based techniques</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODES</td>
<td>Jul 2012</td>
<td>Jul 2016</td>
<td>Extracts method description from StackOverflow discussions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CFS</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Summarizes code fragments using SVM and NB</td>
<td></td>
</tr>
<tr>
<td>TASSAL</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Summarizes code using autofolding</td>
<td></td>
</tr>
<tr>
<td>ChangeScribe</td>
<td>Dec 2014</td>
<td>Dec 2015</td>
<td>Generates commit messages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeInsight</td>
<td>Nov 2015</td>
<td>May 2019</td>
<td>Recommends insightful comments for source code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code2Que</td>
<td>Jul 2020</td>
<td>Aug 2021</td>
<td>Suggests improvements in question titles from mined code in StackOverflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BI-TBCNN</td>
<td>Mar 2019</td>
<td>May 2019</td>
<td>Implements a bi-TBCNN model to classify algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepSim</td>
<td>Oct 2018</td>
<td>–</td>
<td>Implements a DL approach to measure code functional similarity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCDetector</td>
<td>Jul 2020</td>
<td>–</td>
<td>Proposes a fine-grained granularity of source code for functionality identification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASCAD</td>
<td>Aug 2018</td>
<td>–</td>
<td>Categorizes software into relevant categories</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunCom</td>
<td>May 2019</td>
<td>–</td>
<td>Summarizes code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SonarQube</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Analyzes code quality</td>
<td></td>
</tr>
<tr>
<td>SVF</td>
<td>Mar 2016</td>
<td>Jul 2022</td>
<td>Enables interprocedural dependency analysis for LLVM-based languages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Designite</td>
<td>Mar 2016</td>
<td>Jul 2023</td>
<td>Detects code smells and computes quality metrics in Java and C# code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Name</td>
<td>Year 1</td>
<td>Year 2</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------</td>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloneCognition</td>
<td>Nov 2018</td>
<td>May 2019</td>
<td>Proposes a ML framework to validate code clones</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMAD [52]</td>
<td>Mar 2020</td>
<td>Feb 2021</td>
<td>Implements smell detection (God class and Feature envy) using ML</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checkstyle</td>
<td>–</td>
<td>–</td>
<td>Checks for coding convention in Java code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindBugs</td>
<td>–</td>
<td>–</td>
<td>Implements a static analysis tool for Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMD</td>
<td>–</td>
<td>–</td>
<td>Finds common programming flaws in Java and six other languages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deep learning smells [437]</td>
<td>Jul 2021</td>
<td>Nov 2020</td>
<td>Implements DL (CNN, RNN, and autoencoder-based models) to identify four smells</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREEC [558]</td>
<td>Nov 2018</td>
<td>–</td>
<td>Recommends clones for refactoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML for software refactoring [40]</td>
<td>Sep 2020</td>
<td>–</td>
<td>Recommends refactoring by using ML</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTLPD [90]</td>
<td>Aug 2019</td>
<td>–</td>
<td>Implements a deep transfer learning framework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BugDetection</td>
<td>Oct 2019</td>
<td>May 2021</td>
<td>Trains models for defect prediction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepBugs [387]</td>
<td>Nov 2018</td>
<td>May 2021</td>
<td>Implements a framework for learning name-based bug detectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoCoNuT [305]</td>
<td>Jul 2020</td>
<td>Sep 2021</td>
<td>Repairs Java programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANX [552]</td>
<td>Oct 2018</td>
<td>–</td>
<td>Translates natural language text to formal meaning representations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeGen</td>
<td>Nov 2019</td>
<td>–</td>
<td>Generates code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AppFlow [197]</td>
<td>Oct 2018</td>
<td>–</td>
<td>Automates UI tests generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agilika [505]</td>
<td>Aug 2020</td>
<td>Mar 2022</td>
<td>Generates tests from execution traces</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The list of datasets found in our exploration is presented in Table 7. Similar to the Tools' table, Table 7 lists each resource with its category, name and link to access the resource, number of citations (as of July 2022), the time when it was first introduced along with the time when the resource was last updated, and a short description of the resource.

Table 7. A list of datasets useful for applying machine learning to source code

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>#Citation</th>
<th>Introd.</th>
<th>Updated</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GHTorrent [163]</td>
<td>728</td>
<td>Oct 2013</td>
<td>Sep 2020</td>
<td>Meta-data from GitHub repositories</td>
</tr>
<tr>
<td>Code Completion</td>
<td>Neural Code Completion</td>
<td>148</td>
<td>Nov 2017</td>
<td>Sep 2019</td>
<td>Dataset and code for code completion with neural attention and pointer networks</td>
</tr>
<tr>
<td>Dataset Name</td>
<td>Codebase/Description</td>
<td>Size/Date</td>
<td>Source/Generation</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------</td>
<td>-------------------------</td>
<td>-------------------</td>
<td>--------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Program Synthesis</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoNaLa corpus [553]</td>
<td>Python snippets and corresponding natural language description</td>
<td>201 Dec 2018-Oct 2021</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Program Comprehension</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program comprehension dataset [462]</td>
<td>Contains code for a program comprehension user survey</td>
<td>61 May 2018-Aug 2021</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CommitGen [212]</td>
<td>Commit messages and the diffs from 1,006 Java projects</td>
<td>116 - -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StaQC [547]</td>
<td>148K Python and 120K SQL question-code pairs from StackOverflow</td>
<td>80 Nov 2019-Aug 2021</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepCom [198]</td>
<td>Dataset for code completion</td>
<td>- May 2018-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>src-d datasets</strong></td>
<td>Various labeled datasets (commit messages, duplicates, DockerHub, and Nuget)</td>
<td>- - -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BigCloneBench [472]</td>
<td>Known clones in the IJ dataset source repository</td>
<td>272 Dec 2014-Mar 2021</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-label smells [169]</td>
<td>A dataset of 445 instances of two code smells and 82 metrics</td>
<td>28 May 2020--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deep learning smells [437]</td>
<td>A dataset of four smells in tokenized form from 1,072 C# and 100 Java repositories</td>
<td>27 Jul 2021-Nov 2020</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML for software refactoring [40]</td>
<td>Dataset for applying ML to recommend refactoring</td>
<td>31 Nov 2019-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QScored [431]</td>
<td>Code smell and metrics dataset for more than 86 thousand open-source repositories</td>
<td>11 Aug 2021-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Landfill [363]</td>
<td>Code smell dataset with public evaluation</td>
<td>34 May 2015-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeepItSimple [139]</td>
<td>A dataset of linguistic antipatterns of 1,753 instances of source code elements</td>
<td>16 Jul 2018-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>-----------------</td>
<td>---------------</td>
<td>-------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Size</td>
<td>8</td>
<td>858</td>
<td>434</td>
<td>59</td>
<td>155</td>
</tr>
<tr>
<td>Description</td>
<td>A dataset of four code smells</td>
<td>Various datasets including defect prediction and cost estimation</td>
<td>A bug prediction dataset containing 4,973M methods belonging to 92 different Java project versions</td>
<td>A bug prediction dataset containing 4,973M methods belonging to 92 different Java project versions</td>
<td>A JavaScript code corpus with 150K code snippets</td>
</tr>
</tbody>
</table>
5. Challenges and Perceived Deficiencies

The aim of this section is to focus on RO4 of the study by consolidating the perceived deficiencies, challenges, and opportunities in applying ML techniques to source code observed from the selected studies. We document challenges or deficiencies mentioned in the considered studies while studying and summarizing them. After the summarization phase was over, we consolidated all the documented notes and prepared a summary that we present below.

- **Standard datasets**: ML is by nature data hungry; specifically, supervised learning methods need a considerably large, cleaned, and annotated dataset. Though the size of available open software engineering artifacts is increasing day by day, the lack of high-quality datasets (i.e., clean and reliably annotated) is one of the biggest challenges in the domain [153, 501, 157, 243, 132, 90, 52, 34, 487, 459, 483, 474, 160, 419, 290, 513, 440, 216]. Therefore, there is a need for defining standardized datasets. Authors have cited low performance, poor generalizability, and over-fitting due to poor dataset quality as the results of the lack of standard validated high-quality datasets.

**Mitigation:** Although available datasets have increased, given a wide number of software engineering tasks and variations in these tasks as well as the need of application-specific datasets, the community still looks for application-specific, large, and high-quality datasets. To mitigate the issue, the community has focused on developing new datasets and making them publicly available by organizing a dedicated track, for example, the MSR data showcase track. Dataset search engines such as the Google dataset search [6], Zenodo [7], and Kaggle datasets [8] could be used to search available datasets. Researchers may also propose generic datasets that can serve multiple application domains or at least different variations of a software engineering task. In addition, recent advancements in ML techniques such as active learning [389, 428, 405] may reduce the need of large datasets. Besides, the way the data is used for model validation must be improved. For example, Jimenez et al. [216] showed that previous studies on vulnerability prediction trained predictive models by using perfect labelling information (i.e., including future labels, as yet undiscovered vulnerabilities) and showed that such an unrealistic labelling assumption can profoundly affect the scientific conclusions of a study as the prediction performance worsen dramatically when one fully accounts for realistically available labelling. Such issues can be avoided by proposing standards for datasets laying out the minimum expectations from any public dataset.

- **Reproducibility and replicability**: Reproducibility and replicability of any ML implementation can be compromised by the factors discussed below.

  - **Insufficient information**: Aspects such as the ML model, their hyper-parameters, data size and ratio (of benign and faulty samples, for instance) are required to understand and replicate the study. During our exploration, we found numerous studies that do not present even the bare-minimum pieces of information to replicate and reproduce their results. Likewise, Di Nucci et al. [127] carried out a detailed replication study and re-

---

7. https://zenodo.org/
ported that the replicated results were lower by up to 90% compared to what was reported in the original study.

- Handling of data imbalance: It is very common to have imbalanced datasets in software engineering applications. Authors use techniques such as under-sampling and over-sampling to overcome the challenge for training. However, test datasets must retain the original sample ratio as found in the real world [127]; carrying out a performance evaluation based on a balanced dataset is flawed. Obviously, the model will perform significantly inferior when it is put at work in a real-world context. We noted many studies [8, 360, 169, 149, 148, 481, 114] that used balanced samples and often did not provide the size and ratio of the training and testing dataset. Such improper handling of data imbalance contributes to poor reproducibility.

Mitigation: The importance of reproducibility and replicability has been emphasized and understood by the software engineering community [286]. It has lead to a concrete artifact evaluation mechanism adopted by leading software engineering conferences. For example, FSE artifact evaluation divides artifacts into five categories—functional, reusable, available, results reproduced, and results replicated. Such thorough evaluation encouraging software engineering authors to produce high-quality documentation along with easily replicate experiment results using their developed artifacts. In addition, efforts (such as model engineering process [50]) are being made to support ML research reproducible and replicable. Finally, identifying practices (such as assumptions related to hardware or dependencies) that may hinder reproducibility improve reproducibility.

- Maturity in ML development: Development of ML systems are inherently different from traditional software development [513]. Phases of ML development are very exploratory in nature and highly domain and problem dependent [513]. Identifying the most appropriate ML model, their appropriate parameters, and configuration is largely driven by trial and error manner [513, 45, 440]. Such an ad hoc and immature software development environment poses a huge challenge to the community.

A related challenge is lack of tools and techniques for various phases and tasks involved in ML software development. It includes effective tools for testing ML programs, ensuring that the dataset are pre-processed adequately, debugging, and effective data management [513, 373, 155]. In addition, quality aspects such as explainability and trust-worthiness are new desired quality aspects especially applicable for ML code where current practices and knowledge is inadequate [155].

Mitigation: The ad-hoc trial and error ML development can be addressed by improved tools and techniques. Even though the variety of ML development environments including managed services such as AWS Sagemaker and Google Notebooks attempt to make ML development easier, they essentially do not offer much help in reducing the ad-hoc nature of the development. A significant research push from the community would make ML development relatively systematic and organized.

Recent advancements in the form of available tools not only help a developer to comprehend the process but also let them effectively manage code, data, and experimental results. Examples of such tools and methods include DarkViz [420] for DL model visualization, MLflow\(^\text{10}\) for managing the ML lifecycle, and DeepFault [136] for identifying faults in DL programs. Such efforts are expected to address the challenge.

Software Engineering for Machine Learning (SE4ML) brings another perspective to this issue by bringing best practices from software engineering to ML development. Efforts in this direction not only can make ML specific code maintainable and reliable but also can contribute back to reproducibility and replicability.

\(^9\)https://2021.esec-fse.org/track/fse-2021-artifacts
\(^{10}\)https://mlflow.org/
• **Data privacy and bias:** Data hungry ML models are considered as good as the data they are consuming. Data collection and preparation without data diversity leads to bias and unfairness. Although we are witnessing more efforts to understand these sensitive aspects [566, 70], the present set of methods and practices lack the support to deal with data privacy issues at large as well as data diversity and fairness [70, 155].

*Mitigation:* Data standards and best practices focusing on data privacy could be considered as an evaluation criterion to mitigate issues concerning data privacy and bias. In addition, mitigation of the issue is also linked with appropriate data pre-processing. Adoption of effective anonymization techniques and data quality assurance practices will further help us deal with the concern.

• **Effective feature engineering:** Features represent the problem-specific knowledge in pieces extracted from the data; the effectiveness of any ML model depends on the features fed into it. Many studies identified the importance of effective feature engineering and the challenges in gathering the same [487, 440, 373, 513, 203]. Specifically, software engineering researchers have notified that identifying and extracting relevant features beyond code quality metrics is non-trivial. For example, Ivers et al. [203] discusses that identifying features that establishes a relationship among different code elements is a significant challenge for ML implementations applied on source code analysis. Sharma et al. [437] have shown in their study that smell detection using ML techniques perform poorly especially for design smells where multiple code elements and their properties has to be observed.

*Mitigation:* Recent advancements in the field of large language models (LLMs) trained on huge corpus of code and text have significantly eased the task for researchers. For example, tasks such as generating code embeddings and fine-tuning are supported natively by the LLMs. However, encoding code features specific to downstream tasks is required often and making the task easier requires a significant push from the research community.

• **Skill gap:** Wan et al. [513] identified that ML software development requires an extended set of skills beyond software development including ML techniques, statistics, and mathematics apart from the application domain. Similarly, Hall and Bowes [181] also reports a serious lack of ML expertise in academic software engineering efforts. Other authors [373] have emphasized the importance of domain knowledge to design effective ML models.

*Mitigation:* Raising awareness and training sessions customized for the audience is considered the mitigation strategy for skill gap. Software engineering conferences organize tutorials that typically helps new researchers in the field. Availability of various hands-on courses and lecture series from known universities also help bringing the gap.

• **Hardware resources:** Given the need of large training datasets and many hidden layers, often ML training requires high-end processing units (such as GPUs and memory) [513, 155]. A usersurvey study [513] highlights the need to special hardware for ML training. Such requirements poses a challenge to researchers constrained with limited hardware resources.

*Mitigation:* ML development is resource hungry. Certain ML models (such as models based on RNN) consume excessive hardware resources. The need for a large-scale hardware infrastructure is increasing with the increase in size of the captured features and the training samples. To address the challenge, infrastructure at institution and country level are maintained in some countries; however, a generic and widely-applicable solution is needed for more globally-inclusive research. Additionally, efforts in the direction of proposed pretrained models, various data pruning techniques, and effective preprocessing techniques are expected to reduce the need of large infrastructure requirements.

6. **Threats to validity**

The first internal threats to validity relates to the concern of covering all the relevant articles in the selected domain. It is prohibitively time consuming to search each machine learning technique
during the literature search. To mitigate the concern, we defined our scope i.e., studies that use ML techniques to solve a software engineering problem by analyzing source code. We also carefully defined inclusion and exclusion criteria for selecting relevant studies. We carry out an extensive manual search process on commonly used digital libraries with the help of a comprehensive set of search terms. Furthermore, we identified a set of frequently occurring keywords in the articles obtained initially for each category individually and carried out another round of literature search with the help of newly identified keywords to enrich the search results.

Another threat to validity is the validity of data extraction and their interpretation applicable to the generated summary and metadata for each selected study. We mitigated this threat by dividing the task of summarization to all the authors and cross verifying the generated information. During the manual summarization phase, metadata of each paper was reviewed by, at least, two authors.

External validity concerns the generalizability and reproducibility of the produced results and observations. We provide a spreadsheet [438] containing all the metadata for all the articles selected in each of the phases of article selection. In addition, inspired by previous surveys [27, 195], we have developed a website\(^\text{11}\) as a living documentation and literature survey to facilitate easy navigation, exploration, and extension. The website can be easily extended as the new studies emerge in the domain; we have made the repository\(^\text{12}\) open-source to allow the community to extend the living literature survey.

7. Conclusions

With the increasing presence of ML techniques in software engineering research, it has become challenging to have a comprehensive overview of its advancements. This survey aims to provide a detailed overview of the studies at the intersection of source code analysis and ML. We have selected 494 studies spanning since 2011 covering 12 software engineering categories. We present a comprehensive summary of the selected studies arranged in categories, subcategories, and their corresponding involved steps. Also, the survey consolidates useful resources (datasets and tools) that could ease the task for future studies. Finally, we present perceived challenges and opportunities in the field. The presented opportunities invite practitioners as well as researchers to propose new methods, tools, and techniques to make the integration of ML techniques for software engineering applications easy, flexible, and maintainable.

Looking ahead: In the recent past, we have witnessed game-changing advancements and all-around adoption of Large language models (LLMs) [572]. LLMs such as GPTx [68, 396] and BERT [123] learn generic language representation. They help ML models perform better with limited training (i.e., fine-tuning) for a targeted downstream task. Universal contextual representation learned from huge corpora (such as all available textbooks and publicly available articles on the internet) makes them suitable for various natural language tasks.

Similarly, language models for code, such as CodeBERT [145], CodeT5 [529], CodeGraphBERT [171], and Llama 2 [485] are gaining popularity rapidly among software engineering researchers. Such pre-trained models are trained with generic objectives with large corpora of code and natural language. The models learn the syntax, semantics, and fundamental relationships among the concepts and entities that make fine-tuning the model for a specific software engineering task easier (in terms of training time). These models are not only extensively used in software engineering research [300, 89, 294, 205, 381] already but also will be shaping the software engineering research for the years to come.

Acknowledgements

We would like to thank Mootez Saad and Abhinav Reddy Mandli for helping us categorize the ML techniques. We also thank anonymous reviewers who helped us significantly improve our

\(^{11}\)http://www.tusharma.in/ML4SCA
^{12}\)https://github.com/tushartushar/ML4SCA
manuscript. Maria Kechagia and Federica Sarro are supported by the ERC grant no. 741278 (EPIC).

References


[63] Frederick Boland and Paul Black. The juliet 1.1 c/c++ and java test suite. (45), 2012-10-01 2012. doi: https://doi.org/10.1109/MC.2012.345.


Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 


